MASTER OF SCIENCE IN ENGINEERING

Secure Embedded Systems

Laboratoire I

U-Boot & Linux Kernel

Savy Cyrille - Mueller Michael

2015

HES-SO//Master
Orientation TIC

Professeur	: Schuler Jean-Roland
Branche	: SES
Salle de laboratoire	: 5
Dernière mise à jour	: 23 avril 2015

Table des matières

Chapitre 1

Introduction

La sécurité dans les systèmes embarqués, qui sont présents de plus en plus souvent dans un nombre de domaine grandissant, est primordiale pour éviter les abus de personnes malintentionnées (voir éviter les erreurs humaines possibles...).

Le but de ce laboratoire était de découvrir l'environnement U-Boot sur la plateforme "Odroid" basée sur un processeur "Samsung", l'Exynos5422 ! C'est une plateforme beaucoup plus performante que celles utilisée précédement au cours de CSEL, qui elle est basée sur un processeur Freescale AFP27.

Dans ce laboratoire, nous avons pu appliquer certaines méthodes de software hardening en changeant quelques commandes de compilation, notamment pour ôter les symboles de "debug".

Chapitre 2

Bootloader U-Boot

2.1 MMC partitionning

2.1.1 Automated MMC partitionning

Le tout premier laboratoire était destiné à créer un petit script qui doit permettre d'automatiser la procédure d'installation de la MMC. Voilà directement notre proposition de script :

```
#!/bin/bash

diskId=$1
specCmd=$2

echo  "------------------ ODROID XU-3 MMC partition utility ----------------"
echo  "|         Written by : Michael Mueller                              |"
echo  "|         Modified by : Cyrille Savy                                |"
echo  "--------------------------------------------------------------------"


# Verification que l'utilisateur est bien root

uid=$(whoami)

if [ "$diskId" == "" ]
then
        echo "./flash_disk.sh <dev_name> [erase] !"
        exit -1
else    echo  "Writing on device : $diskId"
fi


if [ "$uid" == "root" ]
then
        if [ "$specCmd" == "erase_all" ]
        then
                echo "Erasing all..."
                dd if=/dev/zero of=$diskId bs=512 seek=1 count=2097152
                sync
        else echo "Writing partition only..."
        fi

        echo "Creating msdos MBR..."
        # First sector: msdos
        parted $diskId mklabel msdos

        echo "Creating bootfs..."
        # create bootfs 64MB
        parted $diskId mkpart primary ext4 131072s 262143s

        echo "Creating rootfs..."
        # create rootfs 256MB
        parted $diskId mkpart primary ext4 262144s 786431s
```

```
        echo "Creating usrfs..."
        # create usrfs  256MB
        parted $diskId mkpart primary ext4 786432s 1310719s

        echo "Formatting bootfs..."
        # format with label
        mkfs.ext4 $diskId"1" -L bootfs

        echo "Formatting rootfs..."
        # format with label
        mkfs.ext4 $diskId"2" -L rootfs

        echo "Formatting usrfs..."
        # format with label
        mkfs.ext4 $diskId"3" -L usrfs
        sync

        echo "Copying third-party firmware..."
        #copy firmware & bl1.bin, bl2.bin,tzsw.bin
        dd if=~/workspace/xu3/buildroot/output/images/xu3-bl1.bin of=$diskId bs=512
            seek=1
        dd if=~/workspace/xu3/buildroot/output/images/xu3-bl2.bin of=$diskId bs=512
            seek=31
        dd if=~/workspace/xu3/buildroot/output/images/xu3-tzsw.bin of=$diskId bs=512
            seek=719
        sync

        echo "Copy U-BOOT..."
        #copy u-boot
        dd if=~/workspace/xu3/buildroot/output/images/u-boot.bin of=$diskId bs=512 seek
            =63
        sync

        echo "Copy kernel and dtb..."
        mkdir /mnt/bootfs
        mount -t ext4 $diskId"1" /mnt/bootfs
        #copy kernel & flattened device tree
        cp ~/workspace/xu3/buildroot/output/images/uImage /mnt/bootfs/
        cp ~/workspace/xu3/buildroot/output/images/exynos5422-odroidxu3.dtb /mnt/bootfs
            /
        sync
        #unmount filesystem
        umount /mnt/bootfs

        echo "Copy rootfs..."
        #copy rootfs
        dd if=~/workspace/xu3/buildroot/output/images/rootfs.ext4 of=$diskId"2" bs=512
        #dd if=~/workspace/xu3/buildroot/output/images/rootfs.ext4 of=$diskId bs=512
            seek=262144
        sync

        echo ""
        echo "DONE WITH SUCCESS! "
        echo ""
        exit 0

else echo "root permission required !";exit -1
fi
```

Voici quelques explication pour les grandes lignes de ce scripts : dans premier temps on vérifie que l'on a passé un "device" en paramètre (par exemple "/dev/sdb"). Ensuite vérifie que l'utilisateur possède les droits suffisant ("root").

2.1.2 MMC File positionning

Dans cette partie du laboratoire nous avons du essayer de déplacer les divers fichiers sur la carte SD. Nous avons commencé par déplacer "bl1.bin" qui est un premier bootloader. Voilà les commandes utilisées (effacement de la mémoire et réécriture dans une autre zone) :

```
# BL1
dd if=/dev/zero of=$diskId bs=512 seek=1 count=31
dd if=~/workspace/xu3/buildroot/output/images/xu3-bl1.bin of=$diskId bs=512 seek=17849
    # déplacé
```

La carte ne fait rien ! Évidemment, si le processeur ne trouve pas le "bootloader", il ne peut charger aucun logiciel. Nous avons ensuite essayer de déplacer "bl2.bin". Voilà les commandes utilisées (effacement de la mémoire et réécriture dans une autre zone) :

```
# BL2
dd if=/dev/zero of=$diskId bs=512 seek=31 count=29
dd if=~/workspace/xu3/buildroot/output/images/xu3-bl2.bin of=$diskId bs=512 seek=17849
    # déplacé
```

La carte ne fait rien non plus... On peut partir du principe que le premier bootloader à été chargé en mémoire et exécuté et que le deuxième n'étant pas présent n'as pas démarré. Donc le système ne fonctionne pas. Nous avons essayé de déplacer la "trustzone". Voilà les commandes utilisées (effacement de la mémoire et réécriture dans une autre zone) :

```
#TZ
dd if=/dev/zero of=$diskId bs=512 seek=719 count=512
dd if=~/workspace/xu3/buildroot/output/images/xu3-tzsw.bin of=$diskId bs=512 seek=17849
    # déplacé
```

Cette fois u-boot démarre. Est-ce normal ou est-ce que la "trust-zone" n'est pas utilisée, nous ne savons pas... Peut-être que nous avons fait des erreurs de manipulations. Essayons encore de déplacer u-boot. Voilà les commandes utilisées (effacement de la mémoire et réécriture dans une autre zone) :

```
#Uboot
dd if=/dev/zero of=$diskId bs=512 seek=63 count=552
dd if=~/workspace/xu3/buildroot/output/images/u-boot.bin of=$diskId bs=512 seek=17849 #
    déplacé
```

U-boot ne démarre pas (on peut en déduire que c'est logique...) ! Par contre le ventilateur s'est mis à tourner, ce qui porte à penser que "bl2.bin" à bien démarré, mais n'as pas trouvé "u-boot".

```
#Kernel
dd if=/dev/zero of=$diskId bs=512 seek=1263 count=6412
dd if=~/workspace/xu3/buildroot/output/images/uImage of=$diskId bs=512 seek=17849 # dé
    placé
```

```
U-Boot 2012.07 (Feb 03 2015 - 23:05:29) for Exynos5422

CPU: Exynos5422 Rev0.1 [Samsung SOC on SMP Platform Base on ARM CortexA7]
APLL = 800MHz, KPLL = 800MHz
MPLL = 532MHz, BPLL = 825MHz

Board: HardKernel ODROID
DRAM:  2 GiB
WARNING: Caches not enabled

TrustZone Enabled BSP
BL1 version: [illegible]
VDD_KFC: 0x44
LDO19: 0xf2

Checking Boot Mode ... SDMMC
MMC:   S5P_MSHC2: 0, S5P_MSHC0: 1
MMC Device 0: 7.4 GiB
MMC Device 1: [ERROR] response error : 00000006 cmd 8
[ERROR] response error : 00000006 cmd 55
[ERROR] response error : 00000006 cmd 2
there are pending interrupts 0x00010001
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   No ethernet found.
Press 'Enter' or 'Space' to stop autoboot:  0

MMC read: dev # 0, block # 17647, count 256 ... 256 blocks read: OK

MMC read: dev # 0, block # 1263, count 16384 ... 16384 blocks read: OK
Wrong Image Format for bootm command
ERROR: can't get kernel image!
ODROIDXU3>
```

Cette fois-ci, logiquement, U-Boot démarre normalement. Mais au moment de charger le noyau linux, il nous dit qu'il n'arrive pas à le trouver (et oui, il n'est plus à l'adresse configurée dans les commandes de boot). Ce qui est rassurant, c'est que U-Boot n'essaie pas de lancer un code sans en vérifier la validité.

```
#DTB
dd if=/dev/zero of=$diskId bs=512 seek=17647 count=102 # dtb
dd if=~/workspace/xu3/buildroot/output/images/exynos5422-odroidxu3.dtb of=$diskId bs
    =512 seek=17849 #déplacé
```

```
U-Boot 2012.07 (Feb 03 2015 - 23:05:29) for Exynos5422

CPU: Exynos5422 Rev0.1 [Samsung SOC on SMP Platform Base on ARM CortexA7]
APLL = 800MHz, KPLL = 800MHz
MPLL = 532MHz, BPLL = 825MHz

Board: HardKernel ODROID
DRAM:  2 GiB
WARNING: Caches not enabled

TrustZone Enabled BSP
BL1 version: [illegible]
VDD_KFC: 0x44
LDO19: 0xf2

Checking Boot Mode ... SDMMC
MMC:   S5P_MSHC2: 0, S5P_MSHC0: 1
MMC Device 0: 7.4 GiB
MMC Device 1: [ERROR] response error : 00000006 cmd 8
[ERROR] response error : 00000006 cmd 55
[ERROR] response error : 00000006 cmd 2
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   No ethernet found.
Press 'Enter' or 'Space' to stop autoboot:  0

MMC read: dev # 0, block # 17647, count 256 ... there are pending interrupts 0x00000001
256 blocks read: OK

MMC read: dev # 0, block # 1263, count 16384 ... 16384 blocks read: OK
## Booting kernel from Legacy Image at 40007000 ...
   Image Name:   Linux-3.10.63
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3282416 Bytes = 3.1 MiB
   Load Address: 40008000
   Entry Point:  40008000
   Verifying Checksum ... OK
ERROR: Did not find a cmdline Flattened Device Tree
Could not find a valid device tree
ODROIDXU3>
```

Cette fois-ci, U-Boot démarre et arrive à charger charger le noyau linux, trouve que la checksum du noyau est correcte, mais il ne lance pas le noyau car il ne trouve pas un Flattened Device Tree correct.

2.1.3 U-Boot "bootcmds"

Cette petite section nous a permis de tester différentes commandes possible pour démarrer le noyau Linux.

2.1.3.1 bootelf

Nous avons testé cette commande sans succès (après avoir compilé et généré correctement l'image du noyau Linux). Voici les résulats que nous avons obtenus lors du démarrage :

```
Environment size: 2023/16380 bytes
ODROIDXU3> run addttyargs addmmcargs addipargs
ODROIDXU3> mmc read 0 ${fdts_addr}   0x44ef 0x0100

MMC read: dev # 0, block # 17647, count 256 ... there are pending interrupts 0x00000001
256 blocks read: OK
ODROIDXU3> mmc read 0 ${kernel_addr} 0x04ef 0x4000

MMC read: dev # 0, block # 1263, count 16384 ... 16384 blocks read: OK
ODROIDXU3> bootelf -p ${kernel_addr}
## Starting application at 0xc0008000 ...
prefetch abort
pc : [<be871840>]          lr : [<0000001a>]
sp : be765f38  ip : 00000030     fp : 00000000
r10: 00000000  r9 : 00000006     r8 : be765f30
r7 : be767fbc  r6 : 00000002     r5 : 00000000  r4 : c0008000
r3 : 0001000a  r2 : be767fbc     r1 : be767fbc  r0 : 00000002
Flags: nZCv  IRQs off  FIQs off  Mode HYP_32
Resetting CPU ...

emmc resetting ...
resetting ...


U-Boot 2012.07 (Feb 03 2015 - 23:05:29) for Exynos5422

CPU: Exynos5422 Rev0.1 [Samsung SOC on SMP Platform Base on ARM CortexA7]
APLL = 800MHz, KPLL = 800MHz
MPLL = 532MHz, BPLL = 825MHz
```

L'adresse de démarrage du noyau est mal récupérée et le processeur saute à une adresse non valide. Donc il faudrait modifier u-boot pour que cela fonctionne...

2.1.3.2 Bootz

Cette commande avait l'air de fonctionner, mais une fois le noyau décompressé et que "u-boot" lance le noyau, tout reste figé ! Nous avons trouvé la commande dans le code source sans pourvoir approfondir plus le problème. Il se trouve qu'un offset est mal calculé du à une entête du fichier exécutable.

2.1.3.3 Bootp

Nous n'avons malheureusement pas eu le temps de tester cette commande. Elle utilise le protocole TFTP pour copier et démarrer le noyau Linux.

2.1.3.4 Bootm CRC

La commande "bootm" permet de tester si le CRC est correct. Voici les tests que nous avons fait.
Si le CRC est correcte :

```
MMC read: dev # 0, block # 1263, count 16384 ... 16384 blocks read: OK
## Booting kernel from Legacy Image at 40007000 ...
   Image Name:   Linux-3.10.63
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3284192 Bytes = 3.1 MiB
   Load Address: 40008000
   Entry Point:  40008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 41f00000
   Booting using the fdt blob at 0x41f00000
   Loading Kernel Image ... OK
OK
```

En modifiant un bit au hasard, la commande ne devrait pas démarrer :

```
MMC read: dev # 0, block # 1263, count 16384 ... 16384 blocks read: OK
## Booting kernel from Legacy Image at 40007000 ...
   Image Name:   Linux-3.10.63
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3284192 Bytes = 3.1 MiB
   Load Address: 40008000
   Entry Point:  40008000
   Verifying Checksum ... Bad Data CRC
ERROR: can't get kernel image!
```

La commande "bootm" utilise l'algorithme CRC32 pour vérifier l'intégrité. Ce n'est pas la solution idéale à cause des collisions possibles. Il faudrait plutôt une fonction de hashage de type SHA2, voir SHA3 pour les futuriste.
Si on essaye de lancer notre noyau modifié sans contrôle :

```
## Starting application at 0x40007000 ...
data abort
pc : [<be8705fc>]          lr : [<0000001a>]
sp : be765f38  ip : 00000030     fp : 00000000
r10: be766478  r9 : 00000002     r8 : be765f30
r7 : be8acd70  r6 : 40007000     r5 : 00000002  r4 : be76645c
r3 : 40007000  r2 : be76645c     r1 : be76645c  r0 : 00000001
Flags: nZCv  IRQs off  FIQs off  Mode HYP_32
Resetting CPU ...

emmc resetting ...
resetting ...
```

Ça plante bien évidemment ! Donc, au niveau du "bootloader" c'est important de tester l'intégrité du noyau. L'idéal serait de pouvoir tester à l'aide de signature asymétrique (utilisations de certificats...)

2.2 U-Boot hardening

Dans cette partie nous allons nous concentrer sur la sécurité d'u-boot. Nous allons voir différentes techniques qui vont nous permettre de consolider le software du bootloader.

2.2.1 GCC compilation command line

Une technique est d'ajouter quelques variables de compilation à notre environnement "buildroot".

2.2.1.1 Debug symbol stripping

Afin de rendre le "reverse-engineering" un peu plus compliqué, on peut ôter divers symboles de "debug" en supprimant l'option de compilation "-g". Comme demandé, nous l'avons supprimé dans "buildroot" (modification de "config.mk" du dossier "uboot-eiafr"). Voici les options de compilation actuelle :

```
DBGFLAGS= -g # -DDEBUG
OPTFLAGS= -Os #-fomit-frame-pointer

OBJCFLAGS += --gap-fill=0xff

gccincdir := $(shell $(CC) -print-file-name=include)

CPPFLAGS := $(DBGFLAGS) $(OPTFLAGS) $(RELFLAGS)          \
        -D__KERNEL__
```

Et après modification (nous avons aussi ajouté la protection de la pile. Voir le sous-chapitre suivant pour les explications) :

```
# We don't actually use $(ARFLAGS) anywhere anymore, so catch people
# who are porting old code to latest mainline but not updating $(AR).
ARFLAGS = $(error update your Makefile to use cmd_link_o_target and not AR)
RELFLAGS= $(PLATFORM_RELFLAGS)
DBGFLAGS= -g # -DDEBUG
OPTFLAGS= -Os #-fomit-frame-pointer
PROTFLAGS= -fstack-protector-all # Add stack protection canaries

OBJCFLAGS += --gap-fill=0xff

gccincdir := $(shell $(CC) -print-file-name=include)

CPPFLAGS := $(PROTFLAGS) $(OPTFLAGS) $(RELFLAGS)              \
        -D__KERNEL__
```

Par cette simple manière, nous avons déjà protégé considérablement notre bootloader. Il y a encore une grande quantité de librairies standard qui ne sont pas compilée avec ces deux options. Les symboles de debug permettent de trouver les failles d'un système plus facilement et la non-protection de la pile permet leur exploitations... Donc à ne surtout pas oublier dans les nouvelles applications !

2.2.1.2 Using canaries

Comme vu juste précédemment, nous avons ajouté une option de compilation qui permet de protéger la pile : "-fstack–protector-all".
Cette option permet de protéger de manière efficace contre les vulnérabilités de type "bufferoverflow" (généralement suite à une vulnérabilité de type "integeroverflow").

Cette vulnérabilité est simple : si la taille d'une écriture dans un tableau sur la pile n'est pas contrôlée, il y a un risque que l'on réécrive l'adresse de retour de la fonction. C'est par cette méthode que l'on peut injecter un code malin et prendre le contrôle d'un ordinateur.
Pour parer à ce problème, on a rajouté entre l'adresse de retour et les variables locale (qui sont sur la pile), une valeur aléatoire qui est contrôlée à la fin de la fonction. On les appelle des "canaris", nom historique reçu des mineurs de charbon qui utilisaient des canaris pour détecter le "grisou" (gaz très explosif...) présent dans les mines.
Voici un exemple de programme compilé sans et avec cette option. C'est le code "C" identique pour les deux programmes :

```
1  void func_a(int var)
2  {
3          int loc = var;
4          return;
5  }
6
7  int main(void)
8  {
9          func_a(1);
10         return 0;
11 }
```

On voit un bête appel de fonction avec une valeur et une variable local. Voici l'assembleur généré pour la fonction "func_a" :

```
1  080483fc <main>:
2          push   %ebp
3          mov    %esp,%ebp
4          sub    $0x4,%esp
5
6          // Appel de "func_a"
7          movl   $0x1,(%esp)
8          call   80483ed <func_a>
9          ...
10 080483ed <func_a>:
11         // Appel de fonction standard
12         push   %ebp
13         mov    %esp,%ebp
14         sub    $0x10,%esp
15
16         // int loc = var
17         mov    0x8(%ebp),%eax
18         mov    %eax,-0x4(%ebp)
19
20         //Retour de la fonction
21         nop
22         leave
23         ret
```

On voit que la fonction est simple, et qu'aucun contrôle de la pile n'est fait.

Voyons maintenant avec la protection de la pile ce qu'on obtient :

```
0804846e <main>:
        push   %ebp
        mov    %esp,%ebp
        and    $0xfffffff0,%esp
        sub    $0x20,%esp

        // On met le canari sur la pile
        mov    %gs:0x14,%eax // GS = Segment register.
        mov    %eax,0x1c(%esp)

        // EAX = 0
        xor    %eax,%eax

        // Appel de la fonction "func_a"
        movl   $0x1,(%esp)
        call   804843d <func_a>
        ...

0804843d <func_a>:
        push   %ebp
        mov    %esp,%ebp
        sub    $0x28,%esp  // On réserve plus de place

        // int loc = var
        mov    0x8(%ebp),%eax
        mov    %eax,-0x1c(%ebp)  // loc est à l'adresse (%EBP) - 0x1C

        // On met le canari sur la pile
        mov    %gs:0x14,%eax
        mov    %eax,-0xc(%ebp) // Le canari est au dessus des variables.
        xor    %eax,%eax // Mise à zero d'EAX

        mov    -0x1c(%ebp),%eax // EAX = loc
        mov    %eax,-0x10(%ebp) // On met loc sur la pile juste avant le canari...
            Pourquoi ??
        nop

        // Check si le canari est toujours de la bonne valeur
        mov    -0xc(%ebp),%eax
        xor    %gs:0x14,%eax
        je     804846c <func_a+0x2f>
        call   8048310 <__stack_chk_fail@plt> //Appel de cette fonction si le canari
            est faux
        leave
        ret
```

Donc on remarque que le compilateur à rajouté pas mal de lignes pour vérifier que la pile n'as pas été dépassée/écrasée par une action malveillante !

Chapitre 3

Secure Kernel Configuration

3.1 Linux kernel configuration

Pour cette section nous avions du sécuriser le noyau Linux de la meilleure façon (bonnes pratiques). Presque toutes les options étaient déjà dans le bon état.

3.1.1 True Random Number Generator

Le premier point est d'activer le générateur aléatoire hardware (il faut un générateur qui passe le test du "next-bit"). C'est un point capitale pour toute l'utilisation de la cryptographie sur le système ! On peut obtenir le nombre de bits aléatoire disponibles (max 4096 bits) avec la commande :

```
cat /proc/sys/kernel/random/entropy_avail
1697
```

Les processus qui consomme sur "/dev/random" seront bloqué lorsque ce nombre de bits devient trop petits. Il existe un périphérique non-bloquant : "/dev/urandom". Toutefois, son utilisation peut devenir dangereuse lorsque l'entropie devient trop faible : il peut devenir prédictible ! Ce fut le cas de certaines bornes wifi qui avaient au démarrage une entropie faible et donc généraient des nombre prédictibles.

3.1.2 TCP SYN Cookies

Le deuxième point était d'activer la protection contre les attaques DoS "SYN Flood", à l'aide des "TCP SYN Cookies". Basée sur une solution "cryptographique" [1], l'option "TCP SYN Cookies" permet d'éviter une attaque de type "SYN Flood" (Denial of Service).
La grande question : pourquoi peut-on encore choisir de désactiver une protection à une attaque existante et connue ? Certaines personnes disent que le fait d'utiliser les "SYN cookies" provoque une baisse des performances du réseau [2]. Ce point serait à vérifier, et c'est peut-être pour cela que l'option est encore à choix.

3.1.3 VA space randomization

Venons en à un point capital pour sécuriser notre système : activer la répartition aléatoire des adresses virtuelle des processus. Cette technique permet de rendre, par exemple, l'adresse d'un

1. http ://cr.yp.to/syncookies.html
2. http ://ckdake.com/content/2007/disadvantages-of-tcp-syn-cookies.html *(informations anciennes)*

buffer sur la pile aléatoire. Cela signifie que l'exploitation d'une vulnérabilité de type "buffer overflow" deviendra considérablement plus compliqué (un shellcode possède l'adresse en dur, et si celle-ci change, cela devient impossible) !

3.1.4 Linux kernel Read-Only

Le point suivant est de rendre la section de code du noyau "Read-Only". Pourquoi faire cela ? Cela va éviter une changement (détectable) malicieux de l'exécutable du noyau. Pour ne pas oublier, le noyau est "root" et maître de tout le système ! Donc si quelqu'un arrive à prendre le contrôle du flux d'exécution, il possède tout les droits !

3.1.5 /dev/mem access restrictions

Le point suivant paraît logique, mais c'est bien d'en faire mention : il faut absolutment limiter l'accès au fichier "/dev/mem" au super-utilisateur. En travaillant avec ce fichier on à accès à toutes les zones mémoire (processus et noyau).

3.1.6 Debug symbols stripping

Ensuite, comme expliqué dans un chapitre précédent, c'est une bonne pratique d'ôter les symboles de debug pour rendre le "reverse-engineering" plus compliqué (si on veut par exemple étudier des modules "fait maison" linkés statiquement au noyau).

3.1.7 Stack protection

Le point suivant qui consistait à utiliser les canaris dans le code, n'a pas fonctionné au linkage. La fonction de contrôle des canaris n'as pas été trouvé. Elle manque peut-être dans une librairie où le "Makefile" est mal configuré... Ce point serait à approfondir, car ce serait idéal de pouvoir prévenir de cette manière d'actes malicieux (ou d'erreurs de programmation) !

3.1.8 Restrict unprivileged access to kernel (dmesg)

Il existe une option pour limiter l'accès des logs noyau au super-utilisateur. Est-ce que c'est vraiment un grand critère de sécurité ?
Toutes informations non-indispensable peuvent donner un avantages à un adversaire. En effet, quelques informations en trop peuvent permettre un un adversaire de déceler des failles (CVE non-patchées, numéros de versions, etc...). Donc c'est aussi une bonne pratique de limiter ces accès.

3.1.9 SELinux

SELinux est un modèle de sécurité supporté par le noyau Linux (évidemment) et est utilisé sur de nombreuses distributions. Pour l'utiliser, il faut activer la sécurité réseau et l'audit des appels systèmes disponibles. Nous avons la possibilité d'ajouter de nombreuses options de sécurité. Nous nous sommes limité à ceux présentés dans le cours : la sécurité des systèmes de fichier (sujet qui fera partie du prochain rapport de laboratoire.)

3.1.10 Disable IPv6 support

Cette partie permet de désactiver le support d'IPv6 qui n'est pas encore très sûrs sans les outils cryptographique (IPSec). Il existe de nombreuses attaques existantes, non-corrigée (notamment sur Microsoft Windows). C'est un sujet à débat, mais il est préférable de ne pas supporter IPv6 si nous ne l'utilisons pas !

3.2 TCP SYN cookies attack

3.2.1 Scapy script

Pour tenter de faire une attaque "SYN flood" contre la carte embarquée, on avait à faire un programme en Python avec le module Scapy. On a donc fait un script qui fabrique un paquet TCP de connexion (SYN) et l'envoie à l'adresse IP de la carte Odroid (carte connectée par Ethernet au PC) mais sans jamais renvoyer de ACK une fois que la réponse de la carte, SYN+ACK, est reçue, ce qui laisse des connexions semi-ouvertes.

```
#!/usr/bin/python2.7

from scapy.all import *

ip=IP(dst="192.168.0.4", id=1111, ttl=99)/TCP(sport=RandShort(), dport=[22, 80], seq
    =12345, ack=1000, window=1000, flags="S")/""

ls(ip)

srloop(ip, inter=0.3, retry=2, timeout=4)
```

Ensuite on a testé ce script en deux temps, une première fois avec les SYN Cookies désactivés sur la carte et une seconde fois en les activant.

Comme on pouvait s'y attendre, l'attaque n'a pas fonctionné lorsque les SYN Cookies étaient activés. Mais malheureusement, elle n'a pas fonctionné non plus lorsqu'ils étaient désactivés. Nous avons obtenu le résultat suivant dans les deux cas :

```
# netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 :::22                   :::*                    LISTEN
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node Path
unix  3      [ ]         DGRAM                    3389 /dev/log
unix  2      [ ]         DGRAM                    2241
```

3.2.2 Command "hping3"

Comme le script Scapy ne fonctionnait pas très bien, nous avons décidé de tester un autre outil : "hping3". Les résultats sont très concluant. Voici la commande utilisée pour générer l'attaque :

```
hping3 --flood -I eth0 -V --rand-source -S -p 22 192.168.0.11
```

Cette commande générer des attaque de type TCP SYN, avec un port de source aléatoire et l'envoie le plus rapidement possible. Il faut être rapide, car la carte embarquée l'est aussi (8 coeurs !). Voici la liste des ports ouverts sans la protection contre ce type d'attaques :

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 192.168.0.11:22         192.168.0.4:58727       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58732       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:56254       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:25217       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:17960       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:28399       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:64624       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:32836       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:11363       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:30715       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:30714       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58735       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:28398       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:64623       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:51550       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58736       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:34098       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58726       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:5320        SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:10561       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:5239        SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:30718       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58103       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:56185       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:46683       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:11365       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:56122       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:3161        SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:35130       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:10103       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:34099       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58939       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:56251       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:11364       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:56120       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:5321        SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:58730       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:10106       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:35129       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:42625       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:34131       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:38278       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:10565       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:5380        SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:28400       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:38279       SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:7806        SYN_RECV
tcp        0      0 192.168.0.11:22         192.168.0.4:28402       SYN_RECV
```

Et si on active la protection :

```
echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

Voilà le résultat :

```
[  885.225332] [c0] TCP: drop open request from 192.168.0.4/2718
[  885.231051] [c0] TCP: drop open request from 192.168.0.4/2719
[  885.236784] [c0] TCP: drop open request from 192.168.0.4/2720
[  885.242547] [c0] TCP: drop open request from 192.168.0.4/2721
[  885.248294] [c0] TCP: drop open request from 192.168.0.4/2723
[  885.254195] [c0] TCP: drop open request from 192.168.0.4/2728
[  885.259947] [c0] TCP: drop open request from 192.168.0.4/2729
[  885.265443] [c0] TCP: drop open request from 192.168.0.4/2730
[  885.271179] [c0] TCP: drop open request from 192.168.0.4/2731
[  885.277017] [c0] TCP: drop open request from 192.168.0.4/2734
[  890.218136] [c0] net_ratelimit: 236954 callbacks suppressed
[  890.222366] [c0] TCP: drop open request from 192.168.0.4/17814
[  890.228263] [c0] TCP: drop open request from 192.168.0.4/17815
[  890.234098] [c0] TCP: drop open request from 192.168.0.4/17816
[  890.239848] [c0] TCP: drop open request from 192.168.0.4/17817
[  890.245695] [c0] TCP: drop open request from 192.168.0.4/17818
[  890.251504] [c0] TCP: drop open request from 192.168.0.4/17819
[  890.257298] [c0] TCP: drop open request from 192.168.0.4/17820
[  890.263132] [c0] TCP: drop open request from 192.168.0.4/17821
[  890.268942] [c0] TCP: drop open request from 192.168.0.4/17822
[  890.274728] [c0] TCP: drop open request from 192.168.0.4/17823
[  895.233355] [c0] net_ratelimit: 237646 callbacks suppressed
[  895.237580] [c0] TCP: drop open request from 192.168.0.4/42946
[  895.243485] [c0] TCP: drop open request from 192.168.0.4/42947
[  895.249269] [c0] TCP: drop open request from 192.168.0.4/42948
[  895.255095] [c0] TCP: drop open request from 192.168.0.4/42949
[  895.260913] [c0] TCP: drop open request from 192.168.0.4/42950
[  895.266688] [c0] TCP: drop open request from 192.168.0.4/42951
[  895.272530] [c0] TCP: drop open request from 192.168.0.4/42952
[  895.278342] [c0] TCP: drop open request from 192.168.0.4/42953
[  895.284138] [c0] TCP: drop open request from 192.168.0.4/42954
[  895.289969] [c0] TCP: drop open request from 192.168.0.4/42955
[  900.250787] [c0] net_ratelimit: 237704 callbacks suppressed
[  900.255068] [c0] TCP: drop open request from 192.168.0.4/1645
[  900.260871] [c0] TCP: drop open request from 192.168.0.4/1646
[  900.266517] [c0] TCP: drop open request from 192.168.0.4/1647
[  900.272273] [c0] TCP: drop open request from 192.168.0.4/1648
[  900.277992] [c0] TCP: drop open request from 192.168.0.4/1649
[  900.283711] [c0] TCP: drop open request from 192.168.0.4/3704
[  900.289442] [c0] TCP: drop open request from 192.168.0.4/3705
[  900.295173] [c0] TCP: drop open request from 192.168.0.4/3706
[  900.300867] [c0] TCP: drop open request from 192.168.0.4/3707
[  900.306626] [c0] TCP: drop open request from 192.168.0.4/3708
[  905.264150] [c0] net_ratelimit: 237853 callbacks suppressed
[  905.268430] [c0] TCP: drop open request from 192.168.0.4/25087
[  905.274289] [c0] TCP: drop open request from 192.168.0.4/25088
[  905.280079] [c0] TCP: drop open request from 192.168.0.4/25089
[  905.285866] [c0] TCP: drop open request from 192.168.0.4/25090
[  905.291704] [c0] TCP: drop open request from 192.168.0.4/25091
[  905.297516] [c0] TCP: drop open request from 192.168.0.4/25092
```

On voit que le noyau rejette les connexions au fur et a mesure qu'elles arrivent et ne se terminent jamais.

Chapitre 4

Valgrind

4.1 Runtime memory analysis and vulnerability detection

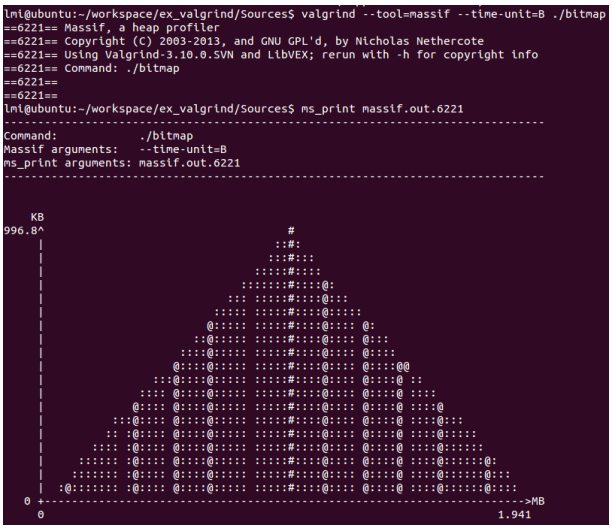
Les deux codes fournis comportent des erreurs de fonctionnement qu'il va falloir découvrir et corriger. Ces codes ont une syntaxe correcte et sont compilables, ce qui ne veut pas dire qu'ils sont corrects et il faut notamment vérifier la manière dont il gèrent la mémoire allouée dynamiquement (allocation, libération, accès) car des erreurs à ce niveau ne sont pas visibles lors de la compilation. Pour les découvrir, on utilise les outils Valgrind vu au cours : "memcheck", "sgcheck" et "massif".

4.1.1 Code 1 : bitmap.c

Ce premier code encode une image dans une autre par stéganographie. Pour ce faire il copie les images dans une zone mémoire allouée dynamiquement, et quand il a terminé il libère cette zone mémoire.

On va d'abord voir si sa mémoire est bien gérée ou non avec l'outil massif :

```
valgrind --tool=massif --time-unit=B ./bitmap
```



Dans ce cas là on ne voit pas grand chose... Si ce n'est que la quantité de mémoire utilisée à la fin du programme est libérée (ou presque, peut-être quelques octets nous échappent, avec une échelle

qui va jusqu'à 1MB on ne voit pas une différence de quelques octets...)

On va donc essayer avec memcheck :

```
valgrind --tool=memcheck --leak-check=full ./bitmap
```

```
lml@ubuntu:~/workspace/ex_valgrind/Sources$ valgrind --tool=memcheck --leak-check=full ./bitmap
==6167== Memcheck, a memory error detector
==6167== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6167== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6167== Command: ./bitmap
==6167==
==6167== Invalid read of size 1
==6167==    at 0x804896C: steganographyEncrypt(char const*, char const*, char const*) (bitmap.C:163)
==6167==    by 0x8048631: main (bitmap.C:68)
==6167==  Address 0x4446686 is 2 bytes after a block of size 1,132 alloc'd
==6167==    at 0x402ADFC: operator new[](unsigned int) (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==6167==    by 0x80488C7: steganographyEncrypt(char const*, char const*, char const*) (bitmap.C:137)
==6167==    by 0x8048631: main (bitmap.C:68)
==6167==
==6167==
==6167== HEAP SUMMARY:
==6167==     in use at exit: 2,100 bytes in 1 blocks
==6167==   total heap usage: 536 allocs, 535 frees, 1,023,328 bytes allocated
==6167==
==6167== 2,100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6167==    at 0x402ADFC: operator new[](unsigned int) (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==6167==    by 0x8048748: steganographyEncrypt(char const*, char const*, char const*) (bitmap.C:109)
==6167==    by 0x8048631: main (bitmap.C:68)
==6167==
==6167== LEAK SUMMARY:
==6167==    definitely lost: 2,100 bytes in 1 blocks
==6167==    indirectly lost: 0 bytes in 0 blocks
==6167==      possibly lost: 0 bytes in 0 blocks
==6167==    still reachable: 0 bytes in 0 blocks
==6167==         suppressed: 0 bytes in 0 blocks
==6167==
==6167== For counts of detected and suppressed errors, rerun with: -v
==6167== ERROR SUMMARY: 241 errors from 2 contexts (suppressed: 0 from 0)
```

Cette fois on voit qu'il se passe des choses incorrectes durant l'exécution du programme. Notamment qu'à un moment il y a un accès mémoire 2 bytes plus loin que la fin d'un bloc alloué dynamiquement, ainsi que toute la mémoire n'est pas libérée à la fin (536 blocs alloués mais seulement 535 libérés, ce qui fait qu'un bloc de (d'une taille de 2.1kB) est définitivement perdu).

En cherchant un peu dans le code, on trouve assez facilement les deux erreurs.

La première est assez grossière et très visible. Elle mène à l'accès mémoire 2 bytes plus loin que la fin du bloc alloué. On voit ci-dessous (ligne 161 du fichier bitmap.c) qu'une boucle for itère jusqu'à un indice dépassant de 2 la largeur de l'image (+2 ajouté à l'indice), et donc accède à une zone mémoire dépassant la taille du bloc alloué.

```
bitmap.p_row = bitmap.p_buffer;
for (i=heightLoop; i; i--, bitmap.p_row++)
{
     fread  (hiddenText.p_buffer, hiddenText.nb_byte_line, 1, pFileHiddenSource);
     hiddenText.p_rgb = (RGB*)hiddenText.p_buffer;


     unsigned short j;
     for (j=0; j<(widthLoop+2); j=j+1, hiddenText.p_rgb++)
     {
         if ((hiddenText.p_rgb->Blue != WHITE) ||
             (hiddenText.p_rgb->Green!= WHITE) ||
             (hiddenText.p_rgb->Red  != WHITE))
         {
             modifyPixel ((*bitmap.p_row) + j);
         }
     }
```

La seconde est classique elle aussi, même si elle est moins visible. La zone mémoire allouée dynamiquement pour l'image est réservée sous la forme d'un tableau de tableaux (tableau 2D). Donc un premier tableau est alloué pour contenir tous les pointeurs vers les autres tableaux qui contiendront, eux, l'image elle-même. Seulement lors de la libération de la mémoire, les tableaux contenant l'image sont libérés, mais le tableau contenant les pointeurs est oublié, ce qui provoque la fuite mémoire.

```
fclose(pFileHiddenSource);


// Copie l'image qui est en memoire dans un fichier
for (i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.p_buffer; i; i--, bitmap.p_row++)
{
        fwrite (*bitmap.p_row, bitmap.nb_byte_line, 1, pFileDest);
}



// Libere toutes les memoires
for (i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.p_buffer; i; i--, bitmap.p_row++)
{
    delete [] *bitmap.p_row;
}

//on libère aussi le premier tableau de pointeurs!!!
delete [] bitmap.p_buffer;

fclose(pFileSource);
fclose(pFileDest);
```

Et maintenant si on relance la commande memcheck sur le code corrigé, elle ne trouve plus de problème lors de l'exécution du programme.

```
lmi@ubuntu:~/workspace/ex_valgrind/Sources$ valgrind --tool=memcheck --leak-check=full ./bitmap
==6907== Memcheck, a memory error detector
==6907== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6907== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6907== Command: ./bitmap
==6907==
==6907==
==6907== HEAP SUMMARY:
==6907==     in use at exit: 0 bytes in 0 blocks
==6907==   total heap usage: 536 allocs, 536 frees, 1,023,328 bytes allocated
==6907==
==6907== All heap blocks were freed -- no leaks are possible
==6907==
==6907== For counts of detected and suppressed errors, rerun with: -v
==6907== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

L'outil sgcheck quant à lui n'a rien donné sur ce programme. Il n'a pas trouvé de problème car il est fait pour détecter des dépassement de pile ou de tableaux (mais pas sur l'allocation dynamique apparemment), et il n'y avait donc pas de problème détectable pour lui.

```
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
==6374==
==6374== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

4.1.2 Code 2 : exe2.c

Le second code est extrêmement mal écrit ! Après une longue étude (:-P) du programme, il s'avère que c'est une implémentation de vecteurs en C. Si on compile le code et qu'on l'exécute avec les outils "Valgrind" pour tester les fuites mémoire, on obtient le résultat suivant :

```
==15093==
==15093== HEAP SUMMARY:
==15093==     in use at exit: 432 bytes in 17 blocks
==15093==   total heap usage: 18 allocs, 1 frees, 448 bytes allocated
==15093==
==15093== 48 (32 direct, 16 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 3
==15093==    at 0x4C2ABB0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15093==    by 0x400728: VectorCreate (exe2.c:82)
==15093==    by 0x400675: main (exe2.c:49)
==15093==
==15093== 384 (360 direct, 24 indirect) bytes in 6 blocks are definitely lost in loss record 3 of 3
==15093==    at 0x4C2ABB0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15093==    by 0x40097B: ResizeArray (exe2.c:150)
==15093==    by 0x40085D: VectorSet (exe2.c:122)
==15093==    by 0x4006B5: main (exe2.c:57)
==15093==
==15093== LEAK SUMMARY:
==15093==    definitely lost: 392 bytes in 7 blocks
==15093==    indirectly lost: 40 bytes in 10 blocks
==15093==      possibly lost: 0 bytes in 0 blocks
==15093==    still reachable: 0 bytes in 0 blocks
==15093==         suppressed: 0 bytes in 0 blocks
==15093==
==15093== For counts of detected and suppressed errors, rerun with: -v
==15093== Use --track-origins=yes to see where uninitialised values come from
==15093== ERROR SUMMARY: 62 errors from 14 contexts (suppressed: 0 from 0)
```

On remarque que 392 bytes ont étés perdus après allocations. Si on utilise l'outil massif on voit clairement que rien n'est libéré :

```
valgrind --tool=massif --time-unit=B ./exe2_orig
```

```
DarkOne@NoOne:~/Master/SeS/Labo_Valgrind$ ms_print massif.out.8001
--------------------------------------------------------------------------------
Command:            ./exe2_orig
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.8001
--------------------------------------------------------------------------------


    B
  864^                                                                      :
     |                                                                      :
     |                                                                      :
     |                                                                 :::::::::
     |                                                                 : :      :
     |                                                               @@@@@@@: :      :
     |                                                             ::@      : :      :
     |                                                      ::::::: @      : :      :
     |                                                   :::     : @      : :      :
     |                                                   : :     : @      : :      :
     |                                          ::::::::: :     : @      : :      :
     |                                    :::::: :      : :     : @      : :      :
     |                                 :::    : :      : :     : @      : :      :
     |                             ::::: :    : :      : :     : @      : :      :
     |                          :::   : :     : :      : :     : @      : :      :
     |                    ::@@@@: :   : :     : :      : :     : @      : :      :
     |                :::: @   : :   : :     : :      : :     : @      : :      :
     |           :::::  : @   : :   : :     : :      : :     : @      : :      :
     |       ::::: : :  : @   : :   : :     : :      : :     : @      : :      :
     |     ::: : : : :  : @   : :   : :     : :      : :     : @      : :      :
   0 +--------------------------------------------------------------------------->B
     0                                                                        864
```

Commençons par la modification de la fonction "main" :

```
int main(int argc, char *argv[]) {
  uint32_t i;
  vector_t v = VectorCreate(4);

  if (v == NULL)
    return EXIT_FAILURE;

  for (i = 0; i < N; ++i) { // Place some elements in the vector.
    int *x = (int*)malloc(sizeof(int));

    /* ----- Michael Hacks -------- */
    *x = 0;
    /* ----- ----- */

    element_t old;
    VectorSet(v, i, x, &old);
  }

  PrintIntVector(v);

  /* ----- Michael Hacks -------- */
  VectorFree(v);
  /* ----- ----- */

  return EXIT_SUCCESS;
}
```

Il manquait une initialisation des éléments du vecteur. Les valeurs des vecteurs étaient aléatoires (on lit les dernières valeur dans la RAM). De plus, la fonction de libération du vecteur n'était même pas appelée !

La prochaine fonction à améliorer est "VectorCreate" :

```
vector_t VectorCreate(size_t n) {

  int i = 0 ;

  vector_t v = (vector_t)malloc(sizeof(struct vector_t));

  if (v == NULL)
    return NULL;

  /* ----- Michael Hacks -------- */
  v->length = n;
  v->arry = (element_t*)malloc(n*sizeof(element_t));
  if(v->arry == NULL)
  {
    free(v);
    return NULL;
  }

  for(i=0;i < n ;i++)
  {
    v->arry[i] = NULL;
  }
  /* ----- ----- */

  return v;
}
```

La mémoire n'était pas libérée en cas de d'erreur d'allocation ultérieure (si l'allocation des éléments échoue, le vecteur alloué juste avant n'était pas libéré) et les éléments n'étaient pas initialisés.

La modification suivante est l'implémentation de la fonction "VectorFree" qui ne libérait pas une grande partie de la mémoire :

```
void VectorFree(vector_t v) {
  int i = 0;
  assert(v != NULL);

  /* ------------- */
  for(i=0;i<v->length;i++)
  {
    free(v->arry[i]);
  }
  free(v->arry);

  /* -------------- */

  free(v);
}
```

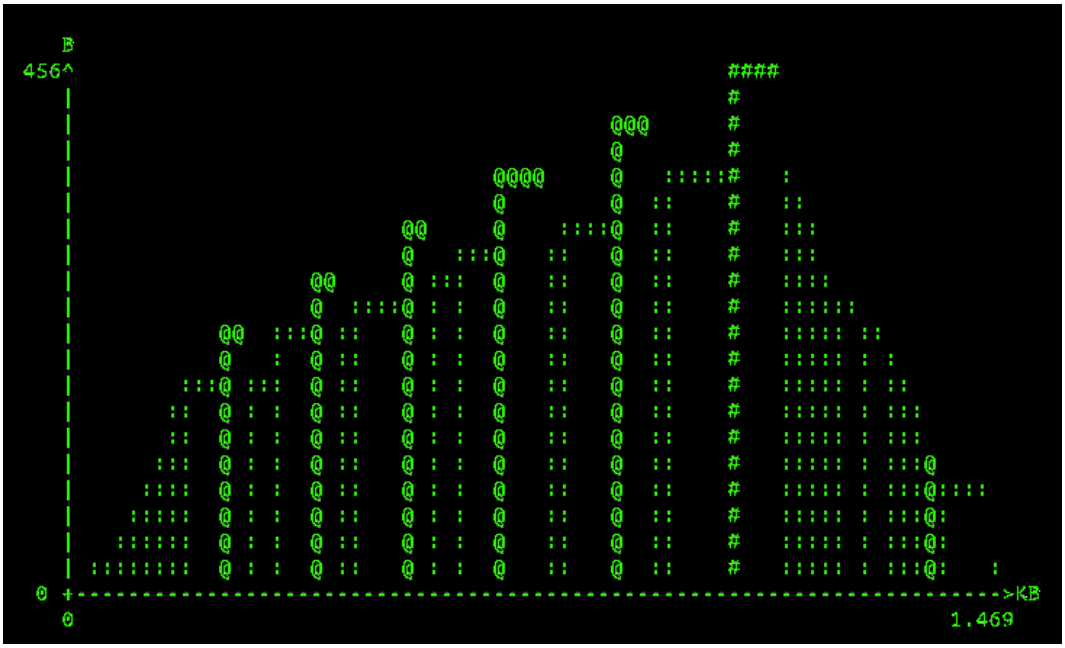
Le programme libérait bien le vecteur, mais pas ses éléments ! Donc presque rien n'était libéré.

Le dernier gros bug joufflu se trouvait dans la fonction "ResizeArray" :

```
static element_t *ResizeArray(element_t *arry, size_t oldLen, size_t newLen) {
  uint32_t i;
  size_t copyLen = oldLen > newLen ? newLen : oldLen;
  element_t *newArry;

  assert(arry != NULL);

  newArry = (element_t*)malloc(newLen*sizeof(element_t));

  if (newArry == NULL)
    return NULL; // malloc error!!!

  // Copy elements to new array
  for (i = 0; i < copyLen; ++i) {
    newArry[i] = arry[i];
  }
  /* ----- Michael Hacks -------- */
  free(arry);
  /* -------------- */

  // Null initialize rest of new array.
  for (i = copyLen; i < newLen; ++i)
    newArry[i] = NULL;

  return newArry;
}
```

La fonction alloue un nouveau tableau d'une taille "X" et l'initialise avec les valeur du tableau précédent. Par contre, l'ancien tableau n'était pas libéré !
Il restait encore quelques modifications pour initialiser comme il faut les valeurs.

Si on lance cette fois notre outil massif, on voit que la mémoire est libérée correctement :



Voici aussi le "memcheck" :

```
==15176==
==15176== HEAP SUMMARY:
==15176==     in use at exit: 0 bytes in 0 blocks
==15176==   total heap usage: 18 allocs, 18 frees, 448 bytes allocated
==15176==
==15176== All heap blocks were freed -- no leaks are possible
==15176==
==15176== For counts of detected and suppressed errors, rerun with: -v
==15176== Use --track-origins=yes to see where uninitialised values come from
==15176== ERROR SUMMARY: 60 errors from 12 contexts (suppressed: 0 from 0)
```

On voit que tout à bien été libéré. Il n'existe plus de fuites mémoire. Remarques : l'outil "Sgcheck" ne donnait pas d'erreur pour ce programme.

Chapitre 5

Conclusion

Ce laboratoire nous a permis de mettre en pratique la théorie vue en cours ainsi que de rafraîchir nos connaissances précédentes sur Linux embarqué. Les systèmes embarqué étant "limités" (cette affirmation est aujourd'hui à moitié vraie... Notre plate-forme est tout de même dotée d'un octo-coeur ARM et de beaucoup de mémoire vive. Mais ce n'est pas forcément le cas de tous les systèmes sur le marché) en performances nécessitent un traitement particulier quand à leur sécurité.

On peut facilement prendre ces systèmes et en extraire les données. Une partie de la sécurité peut être crée à la conception hardware du système (protection des ports JTAG, etc...), mais le reste doit être géré au niveau du software.

Nous avons vu quelques techniques de protections en modifiant les variables de compilation. L'utilisation de la cryptographie devrait être mieux appliquée au niveau du bootloader (notre version ne la support pas, mais elle existe dans les versions supérieure). Contrôler le noyau chargé en mémoire par un hash et/ou une signature (avec ou sans certificats) serait adéquat.
La protection du système de fichiers fait partie de la seconde partie du laboratoire.

Chapitre 6

LUKS

6.1 LUKS, cryptsetup, dmcrypt

6.1.1 Installation

Il faut installer le support dans le noyau avec :

```
make linux-menuconfig
```

et activer l'option :

```
device driver --> Multiple Devices drivers support (RAID and LVM) --> Device mapper
    support --> Crypt target support
```

Et ensuite il faut installer le support dans buildroot avec :

```
make menuconfig
```

et activer l'option :

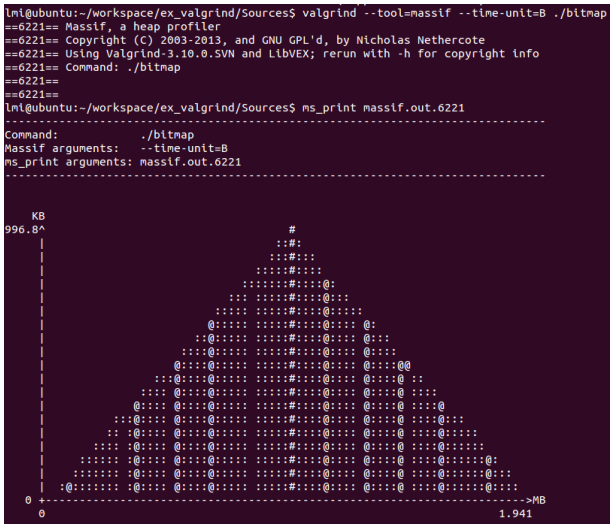
```
target packages --> hardware handling --> cryptsetup
```

6.1.2 Code 1 : bitmap.c

Ce premier code encode une image dans une autre par stéganographie. Pour ce faire il copie les images dans une zone mémoire allouée dynamiquement, et quand il a terminé il libère cette zone mémoire.

On va d'abord voir si sa mémoire est bien gérée ou non avec l'outil massif :

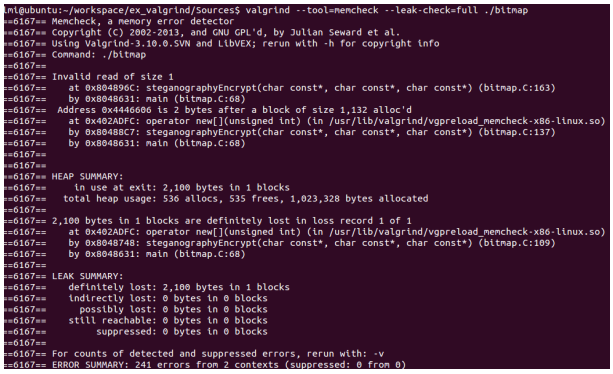
```
valgrind --tool=massif --time-unit=B ./bitmap
```



Dans ce cas là on ne voit pas grand chose... Si ce n'est que la quantité de mémoire utilisée à la fin du programme est libérée (ou presque, peut-être quelques octets nous échappent, avec une échelle qui va jusqu'à 1MB on ne voit pas une différence de quelques octets...)

On va donc essayer avec memcheck :

```
valgrind --tool=memcheck --leak-check=full ./bitmap
```



Cette fois on voit qu'il se passe des choses incorrectes durant l'exécution du programme. Notamment qu'à un moment il y a un accès mémoire 2 bytes plus loin que la fin d'un bloc alloué dynamiquement, ainsi que toute la mémoire n'est pas libérée à la fin (536 blocs alloués mais seulement 535 libérés, ce qui fait qu'un bloc de (d'une taille de 2.1kB) est définitivement perdu).

En cherchant un peu dans le code, on trouve assez facilement les deux erreurs.

La première est assez grossière et très visible. Elle mène à l'accès mémoire 2 bytes plus loin que la fin du bloc alloué. On voit ci-dessous (ligne 161 du fichier bitmap.c) qu'une boucle for itère jusqu'à un indice dépassant de 2 la largeur de l'image (+2 ajouté à l'indice), et donc accède à une zone mémoire dépassant la taille du bloc alloué.

```
bitmap.p_row = bitmap.p_buffer;
for (i=heightLoop; i; i--, bitmap.p_row++)
{
     fread  (hiddenText.p_buffer, hiddenText.nb_byte_line, 1, pFileHiddenSource);
     hiddenText.p_rgb = (RGB*)hiddenText.p_buffer;


     unsigned short j;
     for (j=0; j<(widthLoop+2); j=j+1, hiddenText.p_rgb++)
     {
         if ((hiddenText.p_rgb->Blue != WHITE) ||
             (hiddenText.p_rgb->Green!= WHITE) ||
             (hiddenText.p_rgb->Red  != WHITE))
         {
             modifyPixel ((*bitmap.p_row) + j);
         }
     }
```

La seconde est classique elle aussi, même si elle est moins visible. La zone mémoire allouée dynamiquement pour l'image est réservée sous la forme d'un tableau de tableaux (tableau 2D). Donc un premier tableau est alloué pour contenir tous les pointeurs vers les autres tableaux qui contiendront, eux, l'image elle-même. Seulement lors de la libération de la mémoire, les tableaux contenant l'image sont libérés, mais le tableau contenant les pointeurs est oublié, ce qui provoque la fuite mémoire.

```
fclose(pFileHiddenSource);


// Copie l'image qui est en memoire dans un fichier
for (i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.p_buffer; i; i--, bitmap.p_row++)
{
        fwrite (*bitmap.p_row, bitmap.nb_byte_line, 1, pFileDest);
}



// Libere toutes les memoires
for (i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.p_buffer; i; i--, bitmap.p_row++)
{
    delete [] *bitmap.p_row;
}

//on libère aussi le premier tableau de pointeurs!!!
delete [] bitmap.p_buffer;

fclose(pFileSource);
fclose(pFileDest);
```

Et maintenant si on relance la commande memcheck sur le code corrigé, elle ne trouve plus de problème lors de l'exécution du programme.

```
lmi@ubuntu:~/workspace/ex_valgrind/Sources$ valgrind --tool=memcheck --leak-check=full ./bitmap
==6907== Memcheck, a memory error detector
==6907== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6907== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6907== Command: ./bitmap
==6907==
==6907==
==6907== HEAP SUMMARY:
==6907==     in use at exit: 0 bytes in 0 blocks
==6907==   total heap usage: 536 allocs, 536 frees, 1,023,328 bytes allocated
==6907==
==6907== All heap blocks were freed -- no leaks are possible
==6907==
==6907== For counts of detected and suppressed errors, rerun with: -v
==6907== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

L'outil sgcheck quant à lui n'a rien donné sur ce programme. Il n'a pas trouvé de problème car il est fait pour détecter des dépassement de pile ou de tableaux (mais pas sur l'allocation dynamique apparemment), et il n'y avait donc pas de problème détectable pour lui.

```
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
--6374-- warning: evaluate_Dwarf3_Expr: unhandled DW_OP_ 0x93
==6374==
==6374== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

6.1.3 Code 2 : exe2.c

Le second code est extrêmement mal écrit ! Après une longue étude (:-P) du programme, il s'avère que c'est une implémentation de vecteurs en C. Si on compile le code et qu'on l'exécute avec les outils "Valgrind" pour tester les fuites mémoire, on obtient le résultat suivant :

```
==15093==
==15093== HEAP SUMMARY:
==15093==     in use at exit: 432 bytes in 17 blocks
==15093==   total heap usage: 18 allocs, 1 frees, 448 bytes allocated
==15093==
==15093== 48 (32 direct, 16 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 3
==15093==    at 0x4C2ABB0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15093==    by 0x400728: VectorCreate (exe2.c:82)
==15093==    by 0x400675: main (exe2.c:49)
==15093==
==15093== 384 (360 direct, 24 indirect) bytes in 6 blocks are definitely lost in loss record 3 of 3
==15093==    at 0x4C2ABB0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15093==    by 0x40097B: ResizeArray (exe2.c:150)
==15093==    by 0x40085D: VectorSet (exe2.c:122)
==15093==    by 0x4006B5: main (exe2.c:57)
==15093==
==15093== LEAK SUMMARY:
==15093==    definitely lost: 392 bytes in 7 blocks
==15093==    indirectly lost: 40 bytes in 10 blocks
==15093==      possibly lost: 0 bytes in 0 blocks
==15093==    still reachable: 0 bytes in 0 blocks
==15093==         suppressed: 0 bytes in 0 blocks
==15093==
==15093== For counts of detected and suppressed errors, rerun with: -v
==15093== Use --track-origins=yes to see where uninitialised values come from
==15093== ERROR SUMMARY: 62 errors from 14 contexts (suppressed: 0 from 0)
```

On remarque que 392 bytes ont étés perdus après allocations. Si on utilise l'outil massif on voit clairement que rien n'est libéré :

```
valgrind --tool=massif --time-unit=B ./exe2_orig
```

```
DarkOne@NoOne:~/Master/SeS/Labo_Valgrind$ ms_print massif.out.8001
--------------------------------------------------------------------------------
Command:            ./exe2_orig
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.8001
--------------------------------------------------------------------------------


    B
  864^                                                                      :
     |                                                                      :
     |                                                                      :
     |                                                                 :::::::::
     |                                                                 : :      :
     |                                                               @@@@@@@: :      :
     |                                                             ::@      : :      :
     |                                                      ::::::: @      : :      :
     |                                                   :::       : @      : :      :
     |                                                   : :       : @      : :      :
     |                                             ::::::::: :     : @      : :      :
     |                                          :::::: :     : :     : @      : :      :
     |                                       :::     : :     : :     : @      : :      :
     |                                    ::::: :    : :     : :     : @      : :      :
     |                                 :::    : :    : :     : :     : @      : :      :
     |                             ::@@@@: :  : :    : :     : :     : @      : :      :
     |                          :::: @    : :  : :    : :     : :     : @      : :      :
     |                    :::::  : @    : :  : :    : :     : :     : @      : :      :
     |              ::::: : :  : @    : :  : :    : :     : :     : @      : :      :
     |        ::: : : : :  : @    : :  : :    : :     : :     : @      : :      :
   0 +-------------------------------------------------------------------------->B
     0                                                                      864
```

Commençons par la modification de la fonction "main" :

```
int main(int argc, char *argv[]) {
  uint32_t i;
  vector_t v = VectorCreate(4);

  if (v == NULL)
    return EXIT_FAILURE;

  for (i = 0; i < N; ++i) { // Place some elements in the vector.
    int *x = (int*)malloc(sizeof(int));

    /* ----- Michael Hacks -------- */
    *x = 0;
    /* ----- ----- */

    element_t old;
    VectorSet(v, i, x, &old);
  }

  PrintIntVector(v);

  /* ----- Michael Hacks -------- */
  VectorFree(v);
  /* ----- ----- */

  return EXIT_SUCCESS;
}
```

Il manquait une initialisation des éléments du vecteur. Les valeurs des vecteurs étaient aléatoires (on lit les dernières valeur dans la RAM). De plus, la fonction de libération du vecteur n'était même pas appelée !

La prochaine fonction à améliorer est "VectorCreate" :

```
vector_t VectorCreate(size_t n) {

  int i = 0 ;

  vector_t v = (vector_t)malloc(sizeof(struct vector_t));

  if (v == NULL)
    return NULL;

  /* ----- Michael Hacks -------- */
  v->length = n;
  v->arry = (element_t*)malloc(n*sizeof(element_t));
  if(v->arry == NULL)
  {
    free(v);
    return NULL;
  }

  for(i=0;i < n ;i++)
  {
    v->arry[i] = NULL;
  }
  /* ----- ----- */

  return v;
}
```

La mémoire n'était pas libérée en cas de d'erreur d'allocation ultérieure (si l'allocation des éléments échoue, le vecteur alloué juste avant n'était pas libéré) et les éléments n'étaient pas initialisés.

La modification suivante est l'implémentation de la fonction "VectorFree" qui ne libérait pas une grande partie de la mémoire :

```
void VectorFree(vector_t v) {
  int i = 0;
  assert(v != NULL);

  /* ------------- */
  for(i=0;i<v->length;i++)
  {
    free(v->arry[i]);
  }
  free(v->arry);

  /* -------------- */

  free(v);
}
```

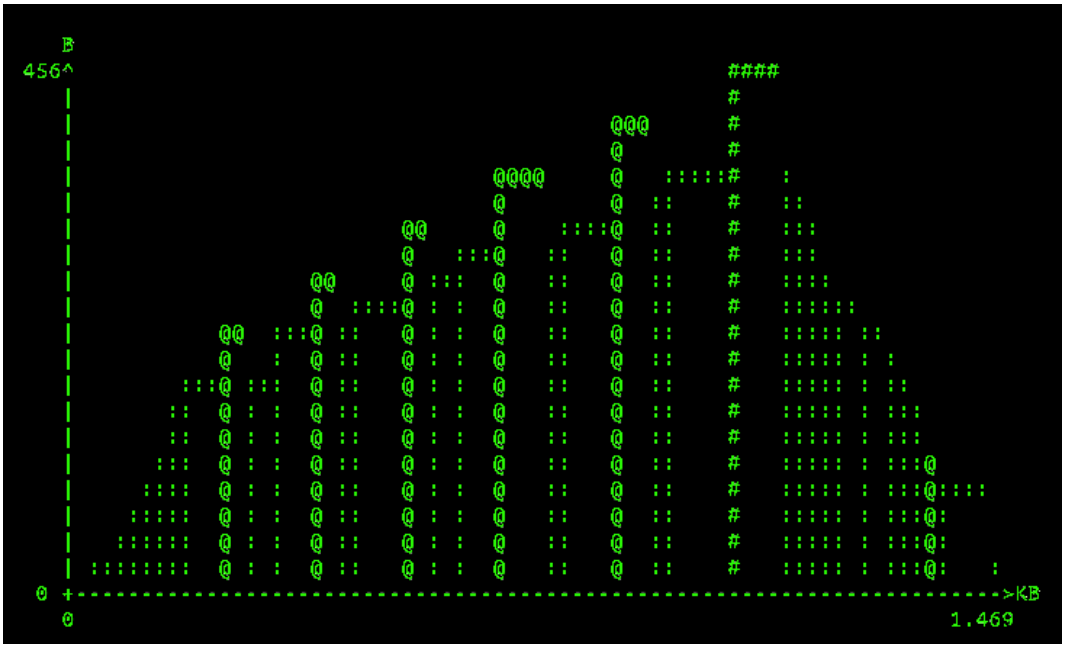
Le programme libérait bien le vecteur, mais pas ses éléments ! Donc presque rien n'était libéré.

Le dernier gros bug joufflu se trouvait dans la fonction "ResizeArray" :

```
static element_t *ResizeArray(element_t *arry, size_t oldLen, size_t newLen) {
  uint32_t i;
  size_t copyLen = oldLen > newLen ? newLen : oldLen;
  element_t *newArry;

  assert(arry != NULL);

  newArry = (element_t*)malloc(newLen*sizeof(element_t));

  if (newArry == NULL)
    return NULL; // malloc error!!!

  // Copy elements to new array
  for (i = 0; i < copyLen; ++i) {
    newArry[i] = arry[i];
  }
  /* ----- Michael Hacks -------- */
  free(arry);
  /* -------------- */

  // Null initialize rest of new array.
  for (i = copyLen; i < newLen; ++i)
    newArry[i] = NULL;

  return newArry;
}
```

La fonction alloue un nouveau tableau d'une taille "X" et l'initialise avec les valeur du tableau précédent. Par contre, l'ancien tableau n'était pas libéré !
Il restait encore quelques modifications pour initialiser comme il faut les valeurs.

Si on lance cette fois notre outil massif, on voit que la mémoire est libérée correctement :



Voici aussi le "memcheck" :

```
==15176==
==15176== HEAP SUMMARY:
==15176==     in use at exit: 0 bytes in 0 blocks
==15176==   total heap usage: 18 allocs, 18 frees, 448 bytes allocated
==15176==
==15176== All heap blocks were freed -- no leaks are possible
==15176==
==15176== For counts of detected and suppressed errors, rerun with: -v
==15176== Use --track-origins=yes to see where uninitialised values come from
==15176== ERROR SUMMARY: 60 errors from 12 contexts (suppressed: 0 from 0)
```

On voit que tout à bien été libéré. Il n'existe plus de fuites mémoire. Remarques : l'outil "Sgcheck" ne donnait pas d'erreur pour ce programme.