

# Secure Embedded System (SeS)

U-boot

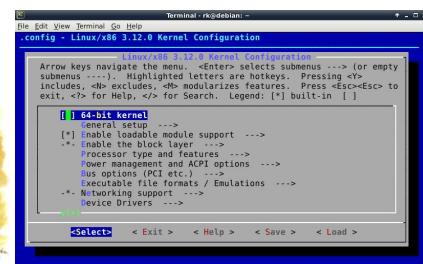


**U-Boot**

Valgrind



Compile kernel



Auteur : Spinelli Isaia  
Prof : Schuler Jean-Roland  
Date : 14.09.2020  
Salle : A4 – Lausanne  
Classe : SeS

# Table des matières

Introduction.....	- 2 -
Valgrind .....	- 2 -
Introduction.....	- 2 -
Question 1 .....	- 2 -
Programme 1.....	- 2 -
Programme 2.....	- 4 -
Programme 3.....	- 5 -
Limitations.....	- 7 -
Question 2 .....	- 8 -
Présentation du programme fourni .....	- 8 -
Correction du programme.....	- 8 -
Conclusion Valgrind.....	- 15 -
U-boot .....	- 16 -
Introduction.....	- 16 -
U-boot configuration.....	- 16 -
Manipulations.....	- 16 -
BOOT partition ext4.....	- 18 -
Manipulations.....	- 18 -
Change network initialization.....	- 20 -
Manipulations.....	- 20 -
Conclusion U-Boot.....	- 21 -
Compile Kernel .....	- 22 -
Le noyau Linux, développé par des contributeurs du monde entier, est un noyau libre et open-source, monolithique , modulaire et hautement configurable.....	- 22 -
Introduction.....	- 22 -
Configurer un noyau.....	- 22 -
Améliorer la sécurité du noyau lors du démarrage.....	- 23 -
L'option CONFIG_FORTIFY_SOURCE .....	- 24 -
Conclusion compile Kernel .....	- 25 -
Annexes .....	- 25 -

# Introduction

Ce rapport est composé de trois laboratoires distincts :

1. Valgrind
2. U-boot
3. Compile Kernel

## Valgrind

Valgrind est un outil de programmation libre. Il permet de détecter automatiquement de nombreux bugs de gestion de la mémoire, de threading et de profiler des programmes en détail. De nombreux modules tiers ont été écrits pour répondre à certaines demandes. Voici quelques exemples :

- *Massif* : Profileur du tas et de la pile. L'interface graphique massif-visualizer permet d'afficher les mesures issues de Massif.
- *Cachegrind* : Profileur de la mémoire cache et de prédiction de branchement. L'outil graphique KCacheGrind Permet de visualiser les mesures issues de Cachegrind.
- *Callgrind* : Analyseur de graphe d'appel de fonctions. KCacheGrind permet aussi d'exploiter les mesures de cet outil.
- *Memcheck* : Détecteur d'erreur mémoire.

## Introduction

Durant ce chapitre l'outil Valgrind est utilisé afin de rechercher des erreurs et corriger des programmes. Le but est de se familiariser avec l'outil et le maîtriser.

## Question 1

Le premier exercice consiste à créer trois petits programmes en C contenant des erreurs afin d'utiliser l'outil Valgrind pour trouver ces erreurs.

### Programme 1

Le 1<sup>er</sup> programme comporte une erreur de type « heap overflow » :

```

1 // Spinelli Isaia
2 // gcc -Wall -g -o Ex1_1 Ex1_1.c
3 // valgrind --tool=memcheck ./Ex1_1
4
5
6 #include "stdio.h"
7 #include "stdlib.h"
8 #include <string.h>
9
10 int heapOverflow(void);
11
12 int main(int argc, char *argv[]) {
13
14     printf("Ex1_1 !\n");
15     heapOverflow();
16     return EXIT_SUCCESS;
17 }
18
19 int heapOverflow(void)
20 {
21
22     char* caractere = (char*) malloc(1);
23     caractere[1] = 'A'; // heap Overflow
24
25     free(caractere);
26     caractere = NULL;
27
28     return 0;
29 }
30

```

Ce programme très simple contient une erreur de type heap overflow à la ligne 23. Afin de trouver cette erreur, Valgrind a été utilisé avec l'outil « memcheck » :

```

[lmi@localhost Laboratoire]$ valgrind --tool=memcheck ./Ex1_1a
==3338== Memcheck, a memory error detector
==3338== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3338== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==3338== Command: ./Ex1_1a
==3338==
Ex1_1 !
==3338== Invalid write of size 1
==3338==   at 0x401189: heapOverflow (Ex1_1a.c:23)
==3338==   by 0x401163: main (Ex1_1a.c:15)
==3338== Address 0x4a2e481 is 0 bytes after a block of size 1 alloc'd
==3338==   at 0x483A809: malloc (vg_replace_malloc.c:307)
==3338==   by 0x40117C: heapOverflow (Ex1_1a.c:22)
==3338==   by 0x401163: main (Ex1_1a.c:15)
==3338==
==3338== HEAP SUMMARY:
==3338==   in use at exit: 0 bytes in 0 blocks
==3338==   total heap usage: 2 allocs, 2 frees, 1,025 bytes allocated
==3338==
==3338== All heap blocks were freed -- no leaks are possible
==3338==
==3338== For lists of detected and suppressed errors, rerun with: -s
==3338== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

La sortie indique correctement qu'il y a une erreur. Une écriture invalide (overflow) de la taille d'un byte a été repéré à la ligne 23.

## Programme 2

Le 2<sup>eme</sup> programme comporte une erreur de type « memory leak » :

```
1 // Spinelli Isaias
2 // gcc -Wall -g -o Ex1_2 Ex1_2.c
3 // valgrind --leak-check=full ./Ex1_2
4
5
6 #include "stdio.h"
7 #include "stdlib.h"
8 #include <string.h>
9
10 int memoryLeak(void);
11
12 int main(int argc, char *argv[])
13 {
14     printf("Ex1_2 !\n");
15     memoryLeak();
16     return EXIT_SUCCESS;
17 }
18
19 int memoryLeak(void)
20 {
21     char* caractere = (char*) malloc(1);
22     caractere[0] = 'A';
23
24     //free(caractere); // memory leak
25     caractere = NULL;
26
27     return 0;
28 }
```

Ce programme est très ressemblant au premier, deux différences ont été apportées :

1. L'erreur de type heap overflow a été corrigée à la ligne 23.
2. Une erreur de type memory leak a été introduite en commentant la ligne 25.

Afin de permettre à Valgrind de détecter ce type d'erreur, l'option « --leak-check=full » de memcheck a été ajoutée.

```

==3843== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[lmi@localhost Laboratoire]$ gcc -Wall -g -o Ex1_2a Ex1_2a.c
[lmi@localhost Laboratoire]$ valgrind --leak-check=full ./Ex1_2a
==3843== Memcheck, a memory error detector
==3843== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3843== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==3843== Command: ./Ex1_2a
==3843=
Ex1_2 !
==3843=
==3843== HEAP SUMMARY:
==3843==     in use at exit: 1 bytes in 1 blocks
==3843==   total heap usage: 2 allocs, 1 frees, 1,025 bytes allocated
==3843==
==3843== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3843==    at 0x483A809: malloc (vg_replace_malloc.c:307)
==3843==    by 0x40116C: memoryLeak (Ex1_2a.c:22)
==3843==    by 0x401153: main (Ex1_2a.c:15)
==3843=
==3843== LEAK SUMMARY:
==3843==  definitely lost: 1 bytes in 1 blocks
==3843==  indirectly lost: 0 bytes in 0 blocks
==3843==  possibly lost: 0 bytes in 0 blocks
==3843==  still reachable: 0 bytes in 0 blocks
==3843==            suppressed: 0 bytes in 0 blocks
==3843=
==3843== For lists of detected and suppressed errors, rerun with: -s
==3843== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

Encore une fois Valgrind a pu détecter l'erreur et indique qu'un byte a été perdu. De plus, il affiche la ligne lors de l'allocation du byte en question (ligne 22).

### Programme 3

Le 3<sup>eme</sup> programme comporte une erreur de type « Segmentation » :

```

1 // Spinelli Isaia
2 // gcc -Wall -g -o Ex1_3 Ex1_3.c
3 // valgrind ./Ex1_3
4
5
6 #include "stdio.h"
7 #include "stdlib.h"
8 #include <string.h>
9
10 int segmentation(void);
11
12 int main(int argc, char *argv[]) {
13
14     printf("Ex1_3 !\n");
15     segmentation();
16     return EXIT_SUCCESS;
17 }
18
19 int segmentation(void)
20 {
21
22     char* caractere = NULL;
23     caractere[0] = 'A'; // Segmentation fault !
24
25     return 0;
26 }
27

```

Ce programme contient une erreur de segmentation à la ligne 23 en écrivant dans une adresse NULL. On peut voir la sortie du Valgrind ci-dessous :

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o Ex1_3a Ex1_3a.c
[lmi@localhost Laboratoire]$ valgrind ./Ex1_3a
==4180== Memcheck, a memory error detector
==4180== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4180== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==4180== Command: ./Ex1_3a
==4180==
Ex1_3 !
==4180== Invalid write of size 1
==4180==   at 0x40115B: segmentation (Ex1_3a.c:23)
==4180==   by 0x401143: main (Ex1_3a.c:15)
==4180== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==4180==
==4180=
==4180=: Process terminating with default action of signal 11 (SIGSEGV): dumping core
==4180== Access not within mapped region at address 0x0
==4180==   at 0x40115B: segmentation (Ex1_3a.c:23)
==4180==   by 0x401143: main (Ex1_3a.c:15)
==4180== If you believe this happened as a result of a stack
==4180== overflow in your program's main thread (unlikely but
==4180== possible), you can try to increase the size of the
==4180== main thread stack using the --main-stacksize= flag.
==4180== The main thread stack size used in this run was 8388608.
==4180==
==4180== HEAP SUMMARY:
==4180==   in use at exit: 0 bytes in 0 blocks
==4180==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4180==
==4180== All heap blocks were freed -- no leaks are possible
==4180==
==4180== For lists of detected and suppressed errors, rerun with: -s
==4180== ERROR SUMMARY: 1 errors from 1 contexts suppressed: 0 from 0
```

Sans problème Valgrind a pu détecter l'erreur de segmentation tout en indiquant la ligne fautive.

## Limitations

Un quatrième programme a été réalisé afin de tester quelques limite de Valgrind :

```

1 // Spinelli Isaia
2 // gcc -Wall -g -o Ex1_limit Ex1_limit.c
3 // valgrind --tool=memcheck --leak-check=full ./Ex1_limit
4
5
6 #include "stdio.h"
7 #include "stdlib.h"
8 #include <string.h>
9
10
11 int main(int argc, char *argv[]) {
12
13     printf("Ex1_limit !\n");
14
15     if (1){
16         int array[3];
17         array[3] = 0; // dépassement d'un tableau alloué automatiquement
18
19         // ne test pas le else
20     } else {
21         char* caractere = NULL;
22         caractere[0] = 'A'; // Segmentation fault !
23     }
24
25
26     return EXIT_SUCCESS;
27 }
```

Ce dernier programme contient deux erreurs :

1. À la ligne 17, une erreur de dépassement d'un tableau alloué automatiquement.
2. À la ligne 22, la même erreur du programme précédent dans une condition toujours fausse.

Ci-dessous, la sortie indiquée par Valgrind :

```
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./Ex1_limit
==4910== Memcheck, a memory error detector
==4910== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4910== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==4910== Command: ./Ex1_limit
==4910==
Ex1_limit !
==4910==
==4910== HEAP SUMMARY:
==4910==     in use at exit: 0 bytes in 0 blocks
==4910==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4910==
==4910== All heap blocks were freed -- no leaks are possible
==4910==
==4910== For lists of detected and suppressed errors, rerun with: -s
==4910== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Cette fois, l'outil ne détecte aucune erreur. On peut donc en conclure que Valgrind est un outil très puissant mais ne permet pas de détecter un dépassement sur un tableau alloué automatique. De plus, il ne vérifie pas les lignes de code dans des conditions toujours fausses.

## Question 2

Pour cette partie, un programme nous a été fourni. Celui-ci est spécialement mauvais et contient donc plusieurs erreurs. Le but est de contrôler et corriger ce programme en utilisant Valgrind.

### Présentation du programme fourni

Ce programme implémente en C une structure nommée « vector\_t » contenant l'adresse d'un tableau et une taille. Plusieurs fonctions permettant la manipulation de ce type de structure ont été implémentées.

Ci-dessous, le programme « main » qui crée un vecteur de 4 éléments, le remplit de N éléments, puis, l'affiche.

```
46 #define N 10 // Test vector length.
47 int main(int argc, char *argv[]) {
48     uint32_t i;
49     vector_t v = VectorCreate(4);
50
51     if (v == NULL)
52         return EXIT_FAILURE;
53
54     for (i = 0; i < N; ++i) { // Place some elements in the vector.
55         int *x = (int*)malloc(sizeof(int));
56         element_t old;
57         VectorSet(v, i, x, &old);
58     }
59
60     PrintIntVector(v);
61
62     return EXIT_SUCCESS;
63 }
```

### Correction du programme

Le système de correction du programme est basé sur une boucle répétitive constituée de quatre étapes :

1. Compilation du programme
2. Exécution de Valgrind sur le fichier exécutable
3. Recherche de l'erreur
4. Correction de l'erreur

Voici pour commencer la compilation et l'exécution de Valgrind sur la base du programme fourni :

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o exe2 exe2.c
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./exe2
==7391== Memcheck, a memory error detector
==7391== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7391== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==7391== Command: ./exe2
==7391==
==7391== Conditional jump or move depends on uninitialized value(s)
==7391==   at 0x4012B1: VectorSet (exe2.c:86)
==7391==     by 0x4011BE: main (exe2.c:57)
==7391==
==7391== Conditional jump or move depends on uninitialized value(s)
==7391==   at 0x40144B: ResizeArray (exe2.c:123)
==7391==     by 0x4012DC: VectorSet (exe2.c:89)
==7391==     by 0x4011BE: main (exe2.c:57)
==7391==
==7391== Conditional jump or move depends on uninitialized value(s)
==7391==   at 0x40147A: ResizeArray (exe2.c:127)
==7391==     by 0x4012DC: VectorSet (exe2.c:89)
==7391==     by 0x4011BE: main (exe2.c:57)
==7391==
==7391== Use of uninitialized value of size 8
==7391==   at 0x401468: ResizeArray (exe2.c:128)
==7391==     by 0x4012DC: VectorSet (exe2.c:89)
==7391==     by 0x4011BE: main (exe2.c:57)
==7391==
```

En haut de l'image on peut voir les commandes permettant la compilation ainsi que l'exécution de l'outil Valgrind.

Ensuite, une grande quantité d'erreurs toujours basées sur une valeur non initialisée. La première est détectée à la ligne 89.

Au bas du terminal, la quantité d'erreurs totale est indiquée (58) :

```
==7391== For details on detected and suppressed errors, rerun with: -s
==7391== ERROR SUMMARY: 58 errors from 17 contexts (suppressed: 0 from 0)
```

Afin de rechercher et comprendre l'erreur, voici quelques lignes du code au-dessus de la ligne 89 comme indiqué par Valgrind :

```

69 vector_t VectorCreate(size_t n) {
70   vector_t v = (vector_t)malloc(sizeof(struct vector_t));
71   v->arry = (element_t*)malloc(n*sizeof(element_t));
72   if (v == NULL || v->arry == NULL)
73     return NULL;
74
75   return v;
76 }
77
78 void VectorFree(vector_t v) {
79   assert(v != NULL);
80   free(v);
81 }
82
83 bool VectorSet(vector_t v, uint32_t index, element_t e, element_t *prev) {
84   assert(v != NULL);
85
86   if [index >= v->length] {
87     ...
88   }
89 }
```

On peut voir à la ligne 86 une comparaison entre un paramètre de la fonction « VectorSet » (index) et un attribut du vecteur v (length). On peut donc conclure que l'attribut « length » du vecteur n'est sûrement pas initialisé.

La fonction permettant de créer un vecteur « VectorCreate » est à la ligne 69. On peut confirmer le fait que la longueur n'est pas initialisée.

Voici la modification apportée :

```

...
69 vector_t VectorCreate(size_t n) {
70   vector_t v = (vector_t)malloc(sizeof(struct vector_t));
71   v->arry = (element_t*)malloc(n*sizeof(element_t));
72   v->length = n;
73   if (v == NULL || v->arry == NULL)
74     return NULL;
75
76   return v;
77 }
```

*Remarque : Pour faire correctement les choses, cette correction ne suffit pas. En effet, les n éléments insérés lors de la création du tableau doivent être initialisés. Dans notre cas, cela n'est pas impératif car juste après la création du vecteur, on remplit le vecteur avec des nouveaux éléments.*

Après une nouvelle compilation et exécution de l'outil Valgrind, on peut voir que les erreurs détectées sont différentes. De plus, le nombre total d'erreur est passé de 58 à 53.

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o exe2 exe2.c
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./exe2
==7412== Memcheck, a memory error detector
==7412== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7412== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==7412== Command: ./exe2
==7412==
==7412== Conditional jump or move depends on uninitialised value(s)
==7412==    at 0x48CD70B: __vfprintf_internal (in /usr/lib64/libc-2.31.so)
==7412==    by 0x48B940E: printf (in /usr/lib64/libc-2.31.so)
==7412==    by 0x4014F4: PrintIntVector (exe2.c:143)
==7412==    by 0x4011D4: main (exe2.c:60)
==7412==
==7412== Use of uninitialised value of size 8
==7412==    at 0x48B33EB: _itoa_word (in /usr/lib64/libc-2.31.so)
==7412==    by 0x48CD0F7: __vfprintf_internal (in /usr/lib64/libc-2.31.so)
==7412==    by 0x48B940E: printf (in /usr/lib64/libc-2.31.so)
==7412==    by 0x4014F4: PrintIntVector (exe2.c:143)
==7412==    by 0x4011D4: main (exe2.c:60)
==7412==

==7412== ERROR SUMMARY: 53 errors from 13 contexts (suppressed: 0 from 0)
```

La cause des erreurs est une fois de plus des valeurs non-initialisées. Cependant, les erreurs proviennent de la fonction « PrintIntVector ».

En recherchant un peu, on peut s'apercevoir que la boucle responsable de remplir le vecteur contient une erreur. En effet, la variable x insérée dans le vecteur est correctement allouée mais pas initialisée.

```
54 for (i = 0; i < N; ++i) { // Place some elements in the vector.
55     int *x = (int*)malloc(sizeof(int));
56     element_t old;
57     VectorSet(v, i, x, &old);
58 }
59
60 PrintIntVector(v);
61
```

Voici	donc	la	correction	apportée :
54 for (i = 0; i < N; ++i) { // Place some elements in the vector. 55     int *x = (int*)malloc(sizeof(int)); 56     *x = 0; 57     element_t old; 58     VectorSet(v, i, x, &old); 59 } 60 61 PrintIntVector(v); 62				

Après une nouvelle compilation et exécution de l'outil Valgrind, on peut voir que les erreurs détectées sont différentes. De plus, le nombre total d'erreur est passé de 53 à 3.

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o exe2 exe2.c
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./exe2
==7435== Memcheck, a memory error detector
==7435== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7435== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==7435== Command: ./exe2
==7435==
[0,0,0,0,0,0,0,0,0]
==7435==
==7435== HEAP SUMMARY:
==7435==     in use at exit: 448 bytes in 18 blocks
==7435==   total heap usage: 19 allocs, 1 frees, 1,472 bytes allocated
==7435==
==7435== 32 bytes in 1 blocks are definitely lost in loss record 1 of 5
==7435==    at 0x483A809: malloc (vg_replace_malloc.c:307)
==7435==    by 0x40120F: VectorCreate (exe2.c:72)
==7435==    by 0x40117E: main (exe2.c:49)
==7435==
==7435== 136 (16 direct, 120 indirect) bytes in 1 blocks are definitely lost in loss record 4 of 5
==7435==    at 0x483A809: malloc (vg_replace_malloc.c:307)
==7435==    by 0x4011FB: VectorCreate (exe2.c:71)
==7435==    by 0x40117E: main (exe2.c:49)
==7435==
==7435== 280 bytes in 5 blocks are definitely lost in loss record 5 of 5
==7435==    at 0x483A809: malloc (vg_replace_malloc.c:307)
==7435==    by 0x40140F: ResizeArray (exe2.c:119)
==7435==    by 0x4012F1: VectorSet (exe2.c:91)
==7435==    by 0x4011C8: main (exe2.c:58)
==7435==
==7435== LEAK SUMMARY:
==7435==    definitely lost: 328 bytes in 7 blocks
==7435==    indirectly lost: 120 bytes in 11 blocks
==7435==    possibly lost: 0 bytes in 0 blocks
==7435==    still reachable: 0 bytes in 0 blocks
==7435==    suppressed: 0 bytes in 0 blocks
==7435==
==7435== For lists of detected and suppressed errors, rerun with: -s
==7435== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Cette fois, la cause des erreurs est une fuite de mémoire. On peut voir qu'enormément de bytes sont perdus, principalement dû à la fonction « VectorCreate ».

On peut voir que la fonction « VectorFree » permettant de libérer toute la mémoire de vecteur n'est jamais appelée :

```
61     PrintIntVector(v);
62
63     return EXIT_SUCCESS;
64 }
```

Voici l'ajout de l'appel à la fonction :

```
...
61     PrintIntVector(v);
62
63     VectorFree(v);
64
65
66     return EXIT_SUCCESS;
```

Après une nouvelle compilation et exécution de l'outil Valgrind, on peut voir que les erreurs détectées sont ressemblantes mais que la quantité de byte perdu indirectement a diminuée alors que la quantité de byte perdu définitivement a augmentée. Cependant, le nombre total d'erreur est passé de 3 à 2.

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o exe2 exe2.c
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./exe2
==7457== Memcheck, a memory error detector
==7457== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7457== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==7457== Command: ./exe2
==7457==
[0,0,0,0,0,0,0,0,0]
==7457==
==7457== HEAP SUMMARY:
==7457==     in use at exit: 432 bytes in 17 blocks
==7457==   total heap usage: 19 allocs, 2 frees, 1,472 bytes allocated
==7457==
==7457== 48 (32 direct, 16 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 3
==7457==   at 0x483A809: malloc (vg_replace_malloc.c:307)
==7457==   by 0x40121B: VectorCreate (exe2.c:72)
==7457==   by 0x40117E: main (exe2.c:49)
==7457==
==7457== 384 (360 direct, 24 indirect) bytes in 6 blocks are definitely lost in loss record 3 of 3
==7457==   at 0x483A809: malloc (vg_replace_malloc.c:307)
==7457==   by 0x40141B: ResizeArray (exe2.c:119)
==7457==   by 0x4012FD: VectorSet (exe2.c:91)
==7457==   by 0x4011C8: main (exe2.c:58)
==7457==
==7457== LEAK SUMMARY:
==7457==   definitely lost: 392 bytes in 7 blocks
==7457==   indirectly lost: 40 bytes in 10 blocks
==7457==   possibly lost: 0 bytes in 0 blocks
==7457==   still reachable: 0 bytes in 0 blocks
==7457==   suppressed: 0 bytes in 0 blocks
==7457==
==7457== For lists of detected and suppressed errors, rerun with: -s
==7457== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

On peut vite en déduire que la fonction « VectorFree » n'a pas été implémenté par « Linus Torvalds » :

```
80 void VectorFree(vector_t v) {
81   assert(v != NULL);
82   free(v);
83 }
```

Voici les modifications apportées :

- Ligne 83-84 : Libérer la mémoire de tous les éléments du tableau
- Ligne 86 : Libérer la mémoire du tableau

```
80 void VectorFree(vector_t v) {
81   assert(v != NULL);
82
83   for (int i = 0; i < VectorLength(v); ++i)
84     free(v->arry[i]);
85
86   free(v->arry);
87   free(v);
88 }
```

Après une nouvelle compilation et exécution de l'outil Valgrind, on peut voir que les erreurs détectées sont encore causées par des fuites de mémoire. Le nombre de byte perdu à encore diminué mais il reste des erreurs.

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o exe2 exe2.c
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./exe2
==7474== Memcheck, a memory error detector
==7474== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7474== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==7474== Command: ./exe2
==7474==
[0,0,0,0,0,0,0,0,0,0]
==7474==
==7474== HEAP SUMMARY:
==7474==     in use at exit: 312 bytes in 6 blocks
==7474==   total heap usage: 19 allocs, 13 frees, 1,472 bytes allocated
==7474==
==7474== 32 bytes in 1 blocks are definitely lost in loss record 1 of 2
==7474==    at 0x483A809: malloc (vg_replace_malloc.c:307)
==7474==    by 0x40121B: VectorCreate (exe2.c:72)
==7474==    by 0x40117E: main (exe2.c:49)
==7474==
==7474== 280 bytes in 5 blocks are definitely lost in loss record 2 of 2
==7474==    at 0x483A809: malloc (vg_replace_malloc.c:307)
==7474==    by 0x401473: ResizeArray (exe2.c:124) [Adresse]
==7474==    by 0x401355: VectorSet (exe2.c:96)
==7474==    by 0x4011C8: main (exe2.c:58)
==7474==
==7474== LEAK SUMMARY:
==7474==    definitely lost: 312 bytes in 6 blocks
==7474==    indirectly lost: 0 bytes in 0 blocks
==7474==    possibly lost: 0 bytes in 0 blocks
==7474==    still reachable: 0 bytes in 0 blocks
==7474==    suppressed: 0 bytes in 0 blocks
==7474==
==7474== For lists of detected and suppressed errors, rerun with: -s
==7474== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

On peut voir que la fonction « ResizeArray » est aussi une source d'erreur. On peut donc y jeter un coup œil :

```
117 static element_t *ResizeArray(element_t *arry, size_t oldLen, size_t newLen) {
118     uint32_t i;
119     size_t copyLen = oldLen > newLen ? newLen : oldLen;
120     element_t *newArry;
121
122     assert(arry != NULL);
123
124     newArry = (element_t*)malloc(newLen*sizeof(element_t)); [Ligne]
125
126     if (newArry == NULL)
127         return NULL; // malloc error!!!
128
129     // Copy elements to new array
130     for (i = 0; i < copyLen; ++i)
131         newArry[i] = arry[i];
132
133     // Null initialize rest of new array.
134     for (i = copyLen; i < newLen; ++i)
135         newArry[i] = NULL;
136
137     return newArry;
138 }
```

On voit que cette fonction alloue de la mémoire mais ne la libère jamais. De ce fait, l'ajout d'un simple appel à la fonction free devrait suffire :

```
137 free(arry);  
138 return newArry;  
139 }
```

Finalement, après une dernière compilation et exécution de l'outil Valgrind, une sortie sans erreur peut être observée :

```
[lmi@localhost Laboratoire]$ gcc -Wall -g -o exe2 exe2.c  
[lmi@localhost Laboratoire]$ valgrind --tool=memcheck --leak-check=full ./exe2  
==7552== Memcheck, a memory error detector  
==7552== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==7552== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info  
==7552== Command: ./exe2  
==7552==  
[0,0,0,0,0,0,0,0,0]  
==7552==  
==7552== HEAP SUMMARY:  
==7552==     in use at exit: 0 bytes in 0 blocks  
==7552==   total heap usage: 19 allocs, 19 frees, 1,472 bytes allocated  
==7552==  
==7552== All heap blocks were freed -- no leaks are possible  
==7552==  
==7552== For lists of detected and suppressed errors, rerun with: -s  
==7552== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## Conclusion Valgrind

Ce laboratoire m'a permis de me familiariser avec l'outil Valgrind. En effet, cet outil est très appréciable car le langage C est plutôt permissif et il est vite arrivé de faire des erreurs. Je suis convaincu qu'il me sera fort utile pour ma vie professionnel car il est simple d'utilisation et très efficace.

# U-boot

U-boot, de l'anglais « Universal-Boot » est un logiciel libre, utilisé comme chargeur d'amorçage, surtout sur les systèmes embarqués. U-Boot peut être divisé en deux étapes : la plate-forme chargerait un petit SPL (Secondary Program Loader), qui est une version allégée de U-Boot, et le SPL ferait la configuration matérielle initiale et le chargement de la version plus grande et complète de U-Boot.

## Introduction

L'objectif de ce laboratoire « U-boot » est de se familiariser avec ce logiciel et surtout d'apprendre à le manipuler afin de pouvoir effectuer des modifications quelconques de configuration. Par exemple, durant ce chapitre, nous allons voir trois différentes manipulations :

1. Changement de la configuration de u-boot (changement du prompt par défaut de u-boot).
2. Changement de la partition BOOT (vfat -> ext4).
3. Changement de l'initialisation du réseau.

## U-boot configuration

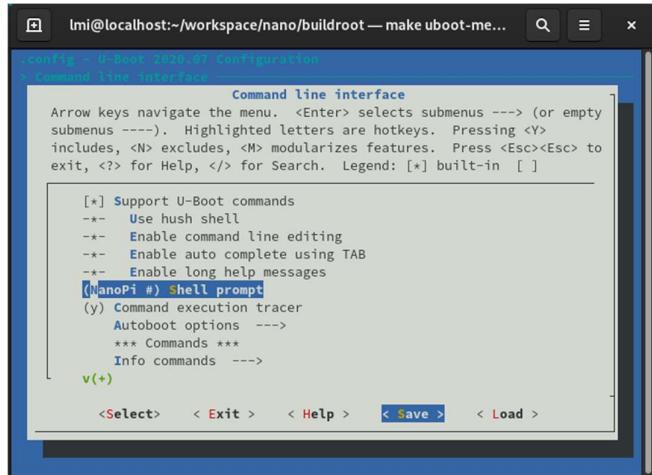
Dans ce chapitre nous allons voir comment changer le prompt par défaut de u-boot de « => » à « NanoPi # ».

### Manipulations

Pour commencer, il faut changer la configuration de u-boot :

« *make uboot-menuconfig* »

La configuration en question est dans le sous-menu « *Command line interface -> Shell prompt* » comme on peut le voir ci-dessous :



Il ne faut pas oublier de sauvegarder.

Ensuite, il est nécessaire de reconstruire u-boot grâce à la commande suivante :

« *make uboot-rebuild* »

« *make* »

Une fois la configuration modifiée, il faut copier les nouveaux fichiers (u-boot.itb et boot.scr) sur la carte SD :

```
[lmi@localhost buildroot]$ sudo dd if=~/workspace/nano/buildroot/output/images/u-boot.itb of=/dev/sdb bs=512 seek=80 status=progress
1212+1 records in
1212+1 records out
620556 bytes (621 kB, 606 KiB) copied, 0.184968 s, 3.4 MB/s
[lmi@localhost buildroot]$ sudo mount /dev/sdb1 /run/media/lmi/
mount: /run/media/lmi: /dev/sdb1 already mounted on /run/media/lmi/BOOT.
[lmi@localhost buildroot]$ sudo cp ~/workspace/nano/buildroot/output/images/boot.scr /run/media/lmi
[lmi@localhost buildroot]$ sudo umount /dev/sdb1
```

Finalement, il est possible de visualiser la modification en redémarrant la cible avec la carte SD. Il faut appuyer sur une touche avant le démarrage automatique du kernel. Voici le nouveau prompt U-boot :

```
U-Boot 2020.07 (Oct 05 2020 - 11:10:19 +0200) Allwinner Technology

CPU:  Allwinner H5 (SUN50I)
Model: FriendlyARM NanoPi NEO Plus2
DRAM: 512 MiB
MMC:  Device 'mmc@lc11000': seq 1 is in use by 'mmc@lc10000'
mmc@lc0f000: 0, mmc@lc10000: 2, mmc@lc11000: 1
Loading Environment from FAT... OK
In:   serial
Out:  serial
Err:  serial
Net:  phy interface?
eth0: ethernet@lc30000
starting USB...
Bus usb@lc1a000: USB EHCI 1.00
Bus usb@lc1a400: USB OHCI 1.0
Bus usb@lc1d000: USB EHCI 1.00
Bus usb@lc1d400: USB OHCI 1.0
scanning bus usb@lc1a000 for devices... 1 USB Device(s) found
scanning bus usb@lc1a400 for devices... 1 USB Device(s) found
scanning bus usb@lc1d000 for devices... 1 USB Device(s) found
scanning bus usb@lc1d400 for devices... 1 USB Device(s) found
      scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot: 0
NanoPi #
NanoPi #
```

## BOOT partition ext4

Par défaut, la partition BOOT est en format vfat. Dans cette section, nous allons changer ce format en ext4. Ce dernier est en général plus apprécié pour plusieurs raisons :

1. Ext4 est un système de fichiers journalisé.
2. La lecture et écriture est plus rapide qu'en vfat.
3. La sécurité est plus prise en charge sur ext4.

## Manipulations

Pour commencer, il faut modifier notre script qui génère la carte SD afin de modifier le format vfat en ext4. Voici les deux principales lignes à modifier :

```
echo "-----1st partition: 64MiB: (163840-32768)*512/1024 = 64MiB" #(sudo fdisk /dev/sdb - interacitf)"
sudo parted /dev/sdb mkpart primary ext4 32768s 163839s

echo "-----2nd partition: 16iB: 4358144-163840)*512/1024 = 16iB"
sudo parted /dev/sdb mkpart primary ext4 163840s 4358143s
sudo mkfs.ext4 /dev/sdb1
sudo mkfs.ext4 /dev/sdb2 -L rootfs
echo "--sync"
sync
```

Il est possible de voir entièrement le script d'initialisation de la carte SD en annexe [2].

Ensuite, il est nécessaire de modifier les commandes exécutées par u-boot via le script « scr » afin qu'il charge l'image linux et le FDT (flattened device tree) depuis un format ext4 et non plus vfat. Pour ce faire, il faut modifier le fichier « boot.cmd » qui dépend de la board utilisée :

```
[lmi@localhost buildroot]$ cat board/friendlyarm/nanopi-neo-plus2/boot.cmd
setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait

fatload mmc 0 $kernel_addr_r Image
fatload mmc 0 $fdt_addr_r nanopi-neo-plus2.dtb

booti $kernel_addr_r - $fdt_addr_r
[lmi@localhost buildroot]$ nano board/friendlyarm/nanopi-neo-plus2/boot.cmd
[lmi@localhost buildroot]$ cat board/friendlyarm/nanopi-neo-plus2/boot.cmd
setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait

ext4load mmc 0 $kernel_addr_r Image
ext4load mmc 0 $fdt_addr_r nanopi-neo-plus2.dtb

booti $kernel_addr_r - $fdt_addr_r
[lmi@localhost buildroot]$
```

Sur l'image précédente, il est possible de voir l'emplacement du fichier de commande de boot (boot.cmd). De plus, on peut voir le contenu du fichier avant et après la modification.

Une fois ce fichier modifié, il faut générer l'image de ce fichier pour U-boot via la commande « *mkimage* » :

```
[lmi@localhost buildroot]$ mkimage -C none -A arm64 -T script -d board/friendlyarm/nanopi-neo-plus2/boot.cmd ./output/images/boot.scr
Image Name:
Created: Mon Oct  5 12:48:06 2020
Image Type: AArch64 Linux Script (uncompressed)
Data Size: 207 Bytes = 0.20 KiB = 0.00 MiB
Load Address: 00000000
Entry Point: 00000000
Contents:
  Image 0: 199 Bytes = 0.19 KiB = 0.00 MiB
```

Maintenant que le script d'initialisation et que les commandes u-boot ont été modifiés, il est possible de générer la carte SD.

Ensuite, il est possible de confirmer la modification du format de la partition via U-boot en utilisant la commande « *ext4ls* » comme ceci :

```
NanoPi # ext4ls mmc 0:1
<DIR>      1024 .
<DIR>      1024 ..
<DIR>      12288 lost+found
30210560  Image
20426  nanopi-neo-plus2.dtb
271 boot.scr
```

On peut voir que tous les fichiers sont là dans le format ext4.

*Remarque : « mmc 0 » correspond à la carte SD. « :1 » correspond à la partition 1 de la carte SD.*

Finalement, en tapant la commande « *boot* », le noyau linux se lance correctement :

```
NanoPi # boot
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
Found U-Boot script /boot.scr
271 bytes read in 3 ms (87.9 KiB/s)
## Executing script at 4fc00000
30210560 bytes read in 1438 ms (20 MiB/s)
20426 bytes read in 4 ms (4.9 MiB/s)
## Flattened Device Tree blob at 4fa00000
  Booting using the fdt blob at 0x4fa00000
EHCI failed to shut down host controller.
  Loading Device Tree to 0000000049ff8000, end 0000000049ffffc9 ... OK

Starting kernel ...
```

## Change network initialization

Linux initialise le réseau avec le script « /etc/init.d/S40network », qui lit le fichier de configuration « /etc/network/interfaces ». Cependant, par défaut, ce fichier de configuration ne configure pas le réseau « eth0 » que nous utilisons.

Afin d'y remédier, il faut créer correctement le fichier de configuration et utiliser la fonctionnalité d'overlay afin que ce fichier soit toujours dans le rootfs de la cible.

### Manipulations

Pour commencer, il faut créer ce fichier dans le répertoire d'overlay associé à la board utilisée :

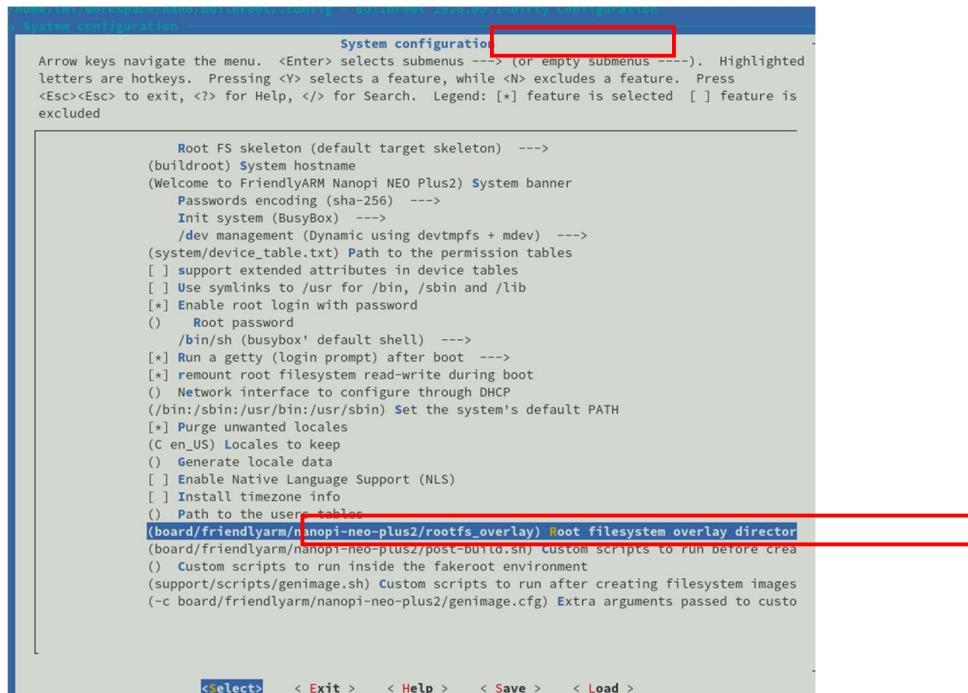
```
[lmi@localhost network]$ pwd
/home/lmi/workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs_overlay/etc/network
[lmi@localhost network]$ cat interfaces
# interface file auto-generated by buildroot

auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 192.168.0.14
    netmask 255.255.255.0
    gateway 192.168.0.4
```

Dans l'image précédente, on peut voir que le fichier a bien été créé et configuré comme souhaité afin d'initialiser automatiquement le réseau « eth0 ».

Ensuite, il faut s'assurer que Buildroot prenne en compte la fonctionnalité d'overlay et qu'il possède le bon chemin jusqu'au rootfs d'overlay. Pour ce faire, il faut modifier la configuration de Buildroot si cela n'est pas fait via la commande « make menuconfig » :



Après avoir recompilé Buildroot et regénéré la carte SD, il est possible de voir le fichier de configuration précédemment créé sur la cible :

```
# cat /etc/network/interfaces
# interface file auto-generated by buildroot

auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 192.168.0.14
    netmask 255.255.255.0
    gateway 192.168.0.4

#
```

De plus, il est possible de voir, directement après le démarrage de la cible, l'interface réseau « eth0 » correctement initialisée avec l'adresse IP configurée précédemment.

```
Welcome to FriendlyARM Nanopi NEO Plus2
buildroot login: root
# [ 7.005275] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 7.011620] dwmac-sun8i 1c30000.ethernet eth0: Link is Up - 1Gbps/Full - flox
ifconfig
eth0      Link encap:Ethernet HWaddr 02:01:E0:1A:C3:77
          inet addr:192.168.0.14 Bcast:0.0.0.0 Mask:255.255.255.0
          inet6 addr: fe80::1:e0ff:fe1a:c377/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:0 (0.0 B) TX bytes:516 (516.0 B)
```

## Conclusion U-Boot

Ce laboratoire m'a permis de me familiariser avec la configuration de U-boot ainsi que de Buildroot. De plus, j'ai appris comment et pourquoi changer le format d'une partition. Finalement, j'ai découvert et pratiqué la fonctionnalité d'overlay. Tous ces outils et concepts sont très utiles pour le développement sur systèmes embarqués.

# Compile Kernel

Le noyau Linux, développé par des contributeurs du monde entier, est un noyau libre et open-source, monolithique, modulaire et hautement configurable.

## Introduction

L'objectif de ce laboratoire est de se familiariser avec les configurations du noyau Linux. Nous allons voir comment activer ou désactiver des configurations du noyau. De plus, nous allons voir comment améliorer la sécurité du noyau de différentes manières.

## Configurer un noyau

La première étape consiste à activer et désactiver des configurations de Linux. Pour ce faire, il faut se placer dans le répertoire de Buildroot (/nano/buildroot) puis exécuter la commande « make linux-menuconfig ». Le principe est le même que pour la configuration de U-boot vu dans le chapitre précédent.

Personnellement j'ai décidé de désactiver deux options qui pourront être testées après leur modification :

1. Désactiver l'accès au .config via /proc/config.gz
2. Désactiver le périphérique /dev/port.

J'ai commencé par tester si ces options étaient correctement activées de base :

```
# ls /proc/config.gz      # ls /dev/port
/proc/config.gz          /dev/port
```

On peut voir que la configuration (config.gz) est bien accessible depuis /proc et que le device « port » est présent.

De plus, pour le prochain laboratoire, j'ai activé l'option qui permet de supporter un Initramf/initrd. Voici les trois options modifiées dans le menu de configuration :

The screenshot shows a terminal window with the following content:

```
[ ] Enable access to .config through /proc/config.gz
< > Enable kernel headers through /sys/kernel/kheaders.tz
[ ] RAW driver (/dev/raw/rawN)
[ ] /dev/port character device
[!] Initial RAM filesystem and RAM disk (initramfs/initrd) support
```

Une fois nos configurations modifiées et sauvegardées, il faut rebuild le noyau linux en exécutant la commande « make linux-rebuild ». Ensuite, on peut mettre à jour la nouvelle image noyau sur la carte SD.

Il est maintenant possible de confirmer les modifications en redémarrant avec le nouveau noyau :

```
# ls /proc/con*      # ls /dev/port
/proc/consoles      ls: /dev/port: No such file or directory
```

On peut voir que le fichier config.gz n'est plus présent ainsi que le device « port ».

## Améliorer la sécurité du noyau lors du démarrage

Ce chapitre permet principalement d'améliorer la sécurité d'ipv4 en ajoutant des paramètres et en les appliquant au démarrage.

Pour commencer il faut créer le fichier de paramétrage dans le dossier « rootfs\_overlay » afin d'ajouter ce fichier automatiquement dans le rootfs.

```
[lmi@localhost etc]$ pwd
/home/lmi/workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs_overlay/etc
[lmi@localhost etc]$ touch sysctl.conf
[lmi@localhost etc]$ nano sysctl.conf
[lmi@localhost etc]$ cat sysctl.conf
kernel.randomize_va_space=2
net.ipv4.conf.lo.rp_filter = 1
net.ipv4.conf.eth0.rp_filter = 1
net.ipv4.conf.lo.accept_source_route= 0
net.ipv4.conf.eth0.accept_source_route= 0
net.ipv4.conf.lo.accept_redirects=0
net.ipv4.conf.eth0.accept_redirects=0
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.icmp_ignore_bogus_error_responses = 1
net.ipv4.conf.lo.log_martians=1
net.ipv4.conf.eth0.log_martians=1
net.ipv4.tcp_synccookies=1
```

Dans l'image ci-dessus on peut voir le chemin du dossier « rootfs\_overlay » ainsi que la création et l'édition du fichier de paramétrage (/etc/sysctl.conf).

Afin d'appliquer tous ces paramètres au noyau dès le démarrage du système, il est nécessaire de créer un script initial dans le dossier « /etc/init.d/ » :

```
[lmi@localhost etc]$ mkdir init.d
[lmi@localhost etc]$ cd init.d/
[lmi@localhost init.d]$ touch S00KernelParameter
[lmi@localhost init.d]$ nano S00KernelParameter
[lmi@localhost init.d]$ cat S00KernelParameter
#!/bin/sh

echo "Test initial script"

sysctl -p
```

Il faut exécuter la commande « sysctl -p » qui va se charger de lire le fichier de configuration précédemment créé et d'appliquer ceux-ci.

Remarque : Il est important d'ajouter les droits d'exécution (chmod) sur le fichier « S00S00KernelParameter ». Sinon le script ne va pas s'exécuter.

```
[lmi@localhost init.d]$ ls -la
total 12
drwxrwxr-x. 2 lmi lmi 4096 Oct 19 14:00 .
drwxr-xr-x. 4 lmi lmi 4096 Oct 19 14:00 ..
-rw-rw-rwx. 1 lmi lmi 52 Oct 19 14:05 S00KernelParameter
```

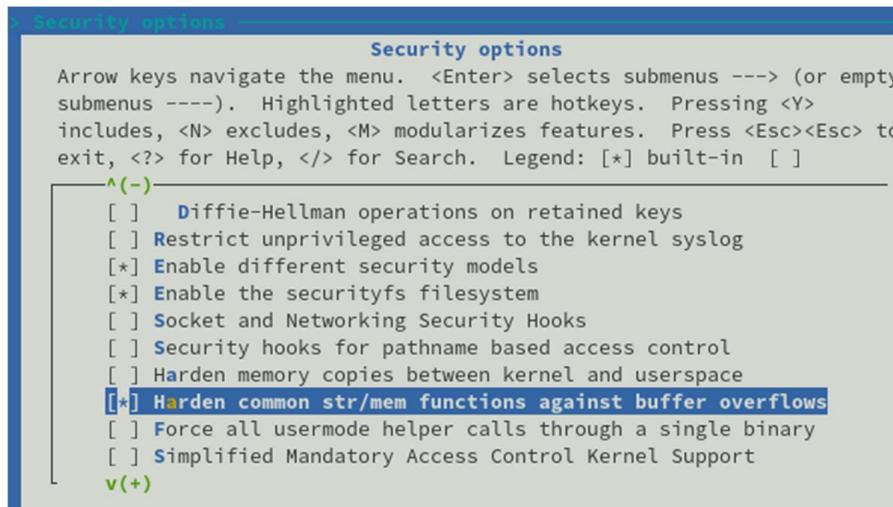
Après avoir ajouté ces fichiers, il faut exécuter la commande « make » dans le dossier Buildroot afin de prendre en compte les nouveaux fichiers d'overlay. Lorsque la compilation est finie on peut mettre à jour la carte SD à l'aide du script créé lors du laboratoire d'initialisation.

En allumant la board avec la carte SD à jour on peut voir les log lors du démarrage :

```
[ 2.510240] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
mount: mounting 192.168.0.1:/home/lmi/workspace on /usr/workspace failed: Network is
Test initial script
kernel.randomize_va_space = 2
net.ipv4.conf.lo.rp_filter = 1
net.ipv4.conf.eth0.rp_filter = 1
net.ipv4.conf.lo.accept_source_route = 0
net.ipv4.conf.eth0.accept_source_route = 0
net.ipv4.conf.lo.accept_redirects = 0
net.ipv4.conf.eth0.accept_redirects = 0
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.icmp_ignore_bogus_error_responses = 1
net.ipv4.conf.lo.log_martians = 1
net.ipv4.conf.eth0.log_martians = 1
sysctl: cannot stat '/proc/sys/net/ipv4/tcp_syncookies': No such file or directory
Starting syslogd: OK
```

## L'option CONFIG\_FORTIFY\_SOURCE

Cette option permet d'améliorer la sécurité des fonctions des librairies str/mem contre les buffer overflows. Afin d'en profiter, il est nécessaire d'activer l'option la concernant comme on peut le voir ci-dessous :



Une fois cette modification effectuée, il faut recompiler le noyau linux « make linux-rebuild » et mettre à jour la carte SD.

Afin d'essayer de comprendre la différence d'implémentation avec cette configuration, j'ai utilisé la commande grep pour voir quelle influence pouvait avoir cette option :

```
[lmj@localhost linux-5.8.6]$ grep -ri CONFIG_FORTIFY_SOURCE
arch/x86/lib/memcpy_32.c:#if defined(CONFIG_X86_USE_3DNOW) && !defined(CONFIG_FORTIFY_SOURCE)
arch/x86/include/asm/string_32.h:#ifndef CONFIG_FORTIFY_SOURCE
arch/x86/include/asm/string_32.h:#endif /* !CONFIG_FORTIFY_SOURCE */
arch/x86/include/asm/string_32.h:#ifndef CONFIG_FORTIFY_SOURCE
arch/x86/include/asm/string_32.h:#ifndef CONFIG_FORTIFY_SOURCE
arch/x86/include/asm/string_32.h:#endif /* !CONFIG_FORTIFY_SOURCE */
arch/Kconfig:      build and run with CONFIG_FORTIFY_SOURCE.
arch/arm/configs/aspeed_g4_defconfig:CONFIG_FORTIFY_SOURCE=y
arch/arm/configs/aspeed_g5_defconfig:CONFIG_FORTIFY_SOURCE=y
arch/powerpc/configs/sk1root_defconfig:CONFIG_FORTIFY_SOURCE=y
arch/s390/configs/debug_defconfig:CONFIG_FORTIFY_SOURCE=y
arch/s390/include/asm/string.h:#if !defined(IN_ARCH_STRING_C) && (!defined(CONFIG_FORTIFY_SOURCE) || defined(__NO_FORTIFY))
Documentation/translations/it_IT/process/deprecated.rst:'CONFIG_FORTIFY_SOURCE=y' e svariate opzioni del compilatore aiutano
Documentation/process/deprecated.rst:'CONFIG_FORTIFY_SOURCE=y' and various compiler flags help reduce the
lib/kconfig.ubsan:      by CONFIG_FORTIFY_SOURCE).
.config.old:CONFIG_FORTIFY_SOURCE=y
.config:CONFIG_FORTIFY_SOURCE=y
include/generated/autoconf.h:#define CONFIG_FORTIFY_SOURCE 1
include/linux/string.h:#if !defined(__NO_FORTIFY) && defined(__OPTIMIZE__) && defined(CONFIG_FORTIFY_SOURCE)
include/config/auto.conf:CONFIG_FORTIFY_SOURCE=y
```

On peut voir dans le rectangle vert que le fichier « string.h » contient bien une influence :

En dessous de ce test il y a l'implémentation de plusieurs fonctions avec un mot clef « FORTIFY\_INLINE ».

On peut voir ci-dessous la fonction « strncpy » :

```
FORTIFY_INLINE char *strncpy(char *p, const char *q, kernel size_t size)
{
    size_t p_size = builtin object size(p, 0);
    if (!builtin constant p(size) && p_size < size)
        write overflow();
    if (p_size < size)
        fortify panic( func );
    return underlying strncpy(p, q, size);
}
```

On peut constater que cette fonction teste les entrées avant d'appeler une autre fonction « strncpy », qui doit faire le vrai traitement. Si elle détecte une erreur, une fonction d'erreur est appelée et la fonction de traitement n'est pas appelée.

## Conclusion compile Kernel

Ce laboratoire m'a permis de me familiariser avec la configuration de Linux. De plus, j'ai appris comment et pourquoi ajouter des configurations améliorant la sécurité. De plus, j'ai pu mettre en pratique l'initialisation de configuration automatique grâce au script de démarrage. Finalement, j'ai pu prendre conscience des tests supplémentaires ajoutés aux fonctions str/mem si la configuration « CONFIG\_FORTIFY\_SOURCE » est active.

## Annexes

1. Exercice 2 du laboratoire Valgrind avec les corrections.
2. Script d'initialisation de la carte SD avec la partition BOOT en format ext4.

Date : 02.11.20

Nom de l'étudiant : Spinelli Isaia