DETROYAT Alexis
VINCENT Côme
BODART Maxime

MOCA
Rapport
Semaine 8

Semaine 8

Utilisation de AFL avec les PC de l'UFR + AFL++

Exercice 1 : Prise en main du fuzzer AFL

Question 1 :

Si on utilise un **x > 99** ou un **x<0** qui correspondent à des valeurs sortant du tableau

Question 2 :

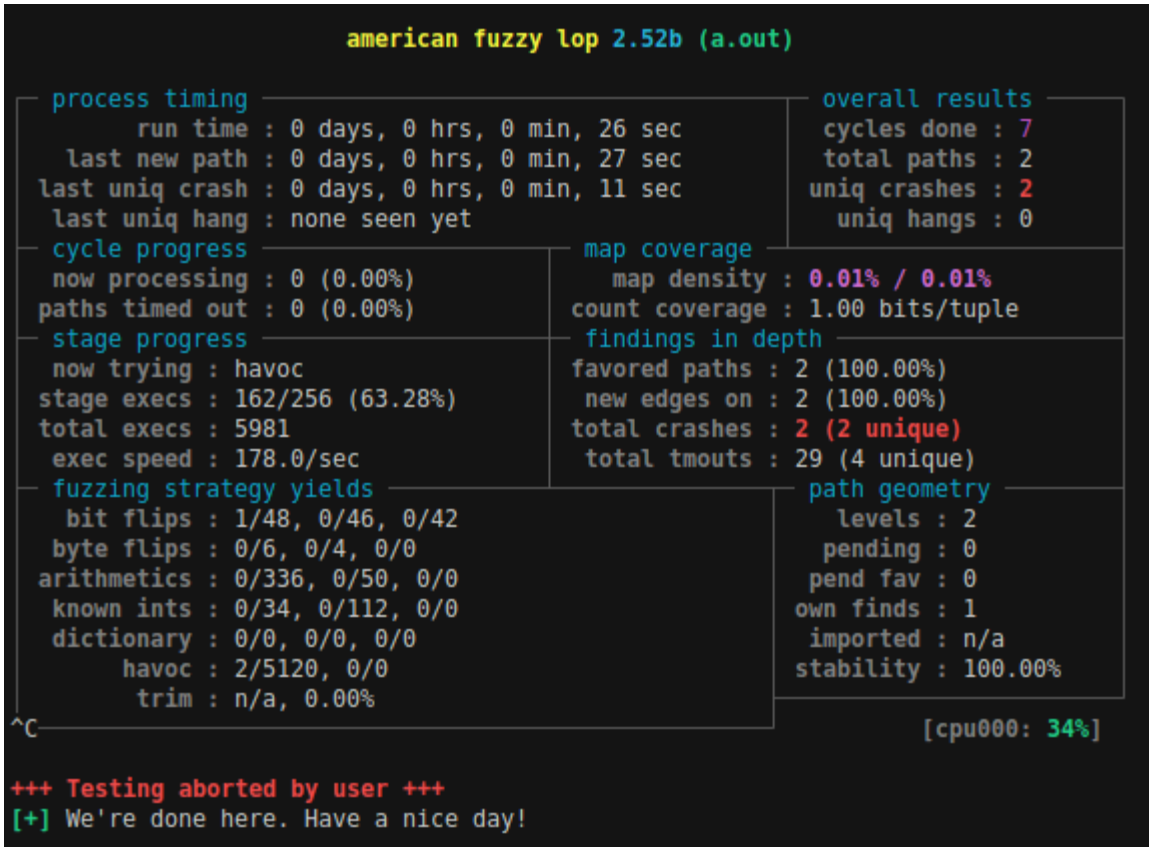
On a créé un fichier **test1.txt** dans le repertoire **in** qui correspond à la valeur **42**

Question 3 :

Après compilation avec **gcc exemple1.c**, on fait **./a.out in/test1.txt**, qui permet donc de lancer exemple1 sur le test qui correspond à la valeur 42, et nous remarquons qu'il n'y a pas d'erreurs.

Question 4 :

En lançant **./run_AFL.sh exemple1.c**, et attendant quelques secondes nous avons obtenu l'affichage ci-dessous :



Nous remarquons donc que nous avons obtenu 2 crashes uniques à partir du fichier **in/test1.txt**.

Question 5 :

Pour le premier crash, on obtient la valeur **4040** :

```
[11:34:22]bodartm$ od -l out/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,rep\:2
0000000          43758399540
0000005
```

et pour le deuxième crash on obtient la valeur **-1821689629**:

```
[11:29:01]bodartm$ hexdump -d out/crashes/id\:000001\,sig\:11\,src\:000000\,op\:havoc\,rep\:64
0000000   13621   13621   13621   13621   13621   13621   13621   13621
0000010   14141   13621   13621   09525   13621   13621   13621   13621
0000020   13621   13621   13621   13621   13621   13621   13621   13621
0000030   13621   11317   13621   13621   13621   13621   13621   13587
0000040   09525   13621   13621   13621   13621   13621   30005   13621
0000050   13621   13621   13621   13621   13621   13621   13633   13621
0000060   13621   13621   13621   13621   13621   13621   13621   13621
0000070   13621   13621   21557   13621   13621   13621   13621   25653
0000080
```

On remarque bien que ce sont des valeurs correspondant aux cas de la **question 1** qui génèrent donc bien des crashs.

Question 6 :

Pour le premier crash, il y a bien une erreur (erreur de segmentation) :

```
[11:32:21][2]bodartm$ ./a.out out/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,rep\:2
Erreur de segmentation (core dumped)
```

Pour le deuxième crash il y a aussi une erreur (erreur de segmentation) :

```
im2ag-217-06:~/L3/S6/MOCA/TP7/TP-ALF
[11:36:05]bodartm$ ./a.out out/crashes/id\:000001\,sig\:11\,src\:000000\,op\:havoc\,rep\:
Erreur de segmentation (core dumped)
```

Avec gdb, on peut remarquer que l'un crash au moment de T[x] = 0 c'est-à-dire dans la branche **if (x > 42),** et l'autre au moment de T[x] = 1, c'est-à-dire dans le else donc pour **if (x <= 42)**.

premier crash avec **gdb** :

```
Program received signal SIGBUS, Bus error.
0x0000555555555213 in main (argc=2, argv=0x7fffffffdb88) at exemple1.c:14
14                      T[x] = 0 ;
(gdb) backtrace
#0  0x0000555555555213 in main (argc=2, argv=0x7fffffffdb88) at exemple1.c:14
(gdb) x/x x
0xfc8:  Cannot access memory at address 0xfc8
(gdb) print x
$1 = 4040
```

deuxième crash avec **gdb** :

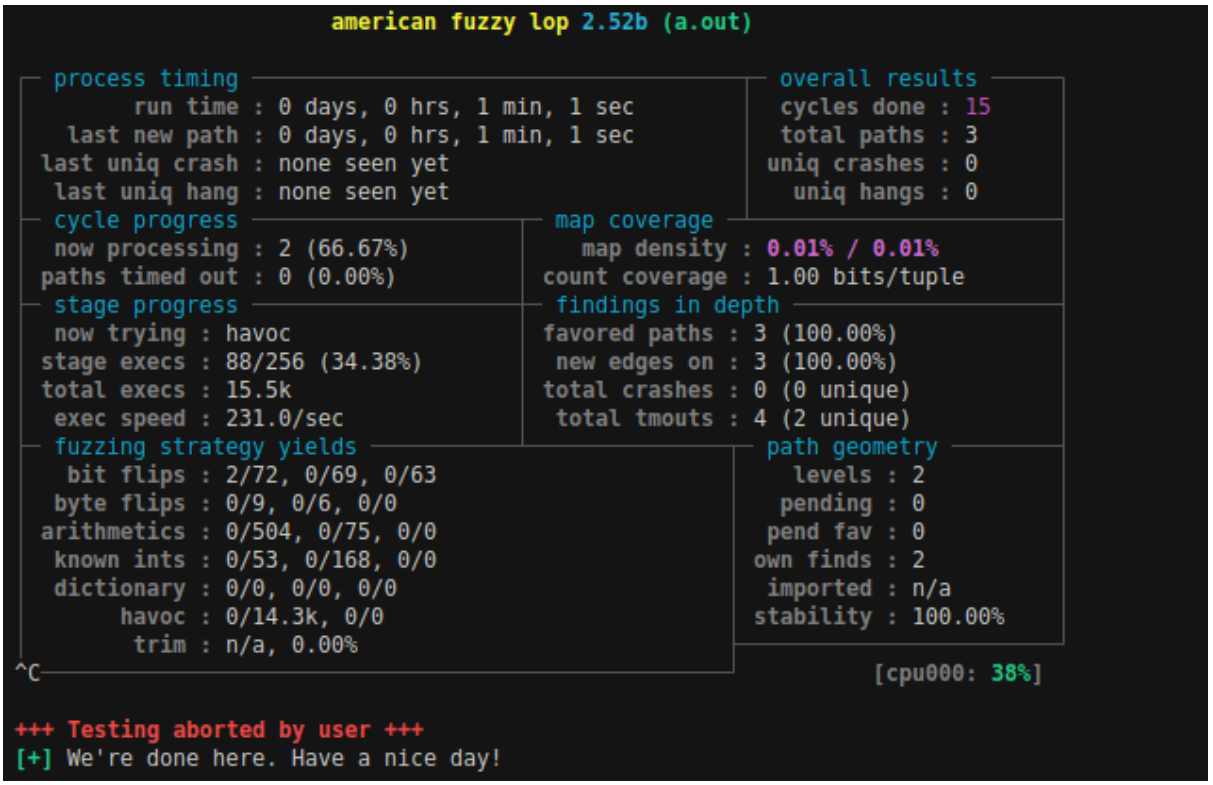
```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555228 in main (argc=2, argv=0x7fffffffdb88) at exemple1.c:16
16                      T[x] = 1 ;
(gdb) print x
$1 = -1821689629
```

Question 7 :

Le problème vient du fait qu'on lise à **T[100]** alors que les indices de **T** se trouvent dans l'intervalle **[0, 99]** : on provoque yun débordement de tableau.

Question 8 :

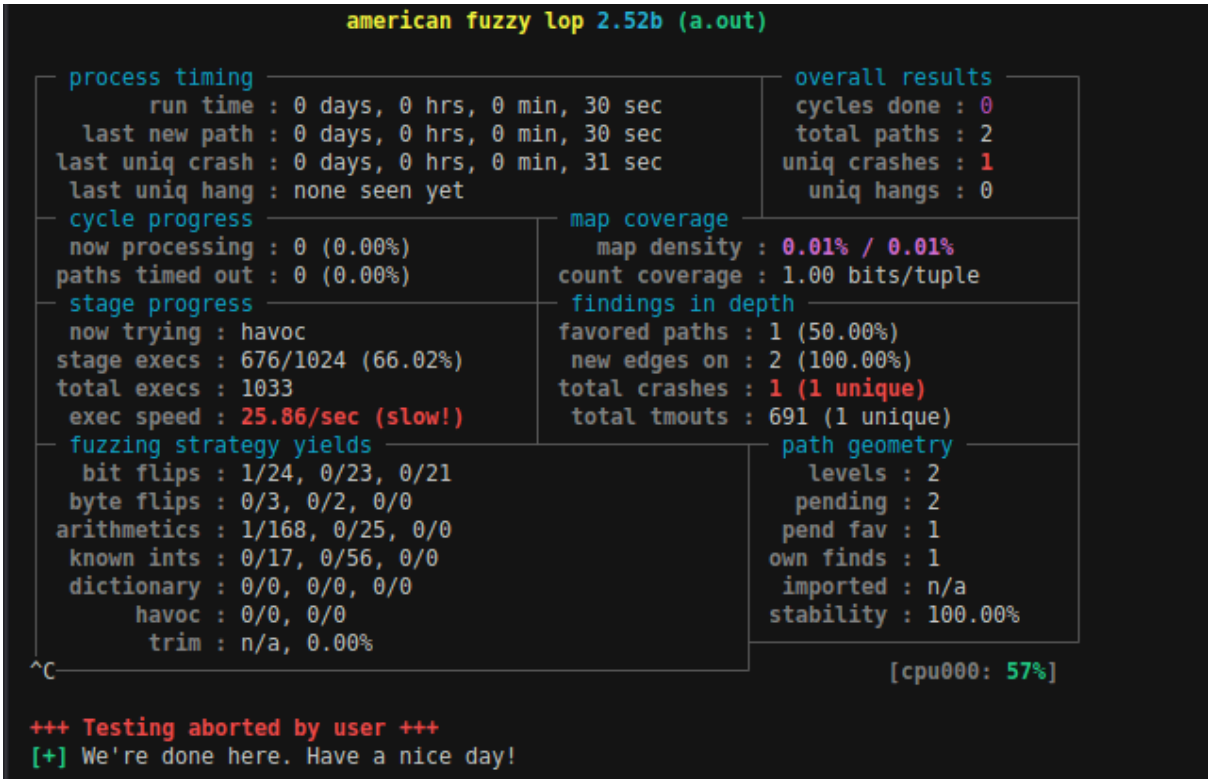
A partir du même fichier d'entrée de test que précédemment avec la valeur 42, on remarque au lancement du programme **exemple2** nous n'avons pas d'erreur avec les commandes **gcc exemple2.c** et **./a.out in/test1.txt**. Après lancement du script AFL avec la commande **./runAFL.sh exemple2.c**, nous n'avons aucun crash au bout d'une minute de lancement :



On ne trouve pas d'erreur car on ne rentre pas dans la branche du code liée au débordement de tableau.

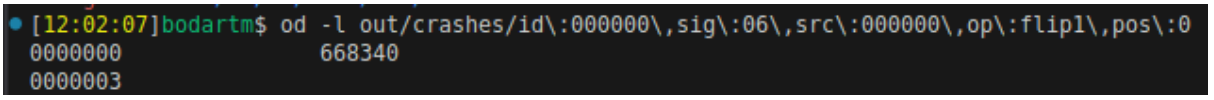
Question 9 :

Après exécution de **exemple2.c** avec l'**AddressSanitizer** grâce à la commande **./runAFL-Asan.sh exemple2.c** on obtient un crash unique :



Question 10 :

L'entrée permettant un crash est la valeur **668340**.



Mais si l'on recompile sans Asan et en lançant la commande **./a.out out/crashes/id\:000000\,sig\:06\,src\:000000\,op\:flip1\,pos\:0** qui lance l'exemple 2 sur le test qui avait causé le crash, nous n'obtenons aucune erreur. En effet, le débordement de tableau n'est pas assez important pour causer une erreur mémoire.

Question 11 :

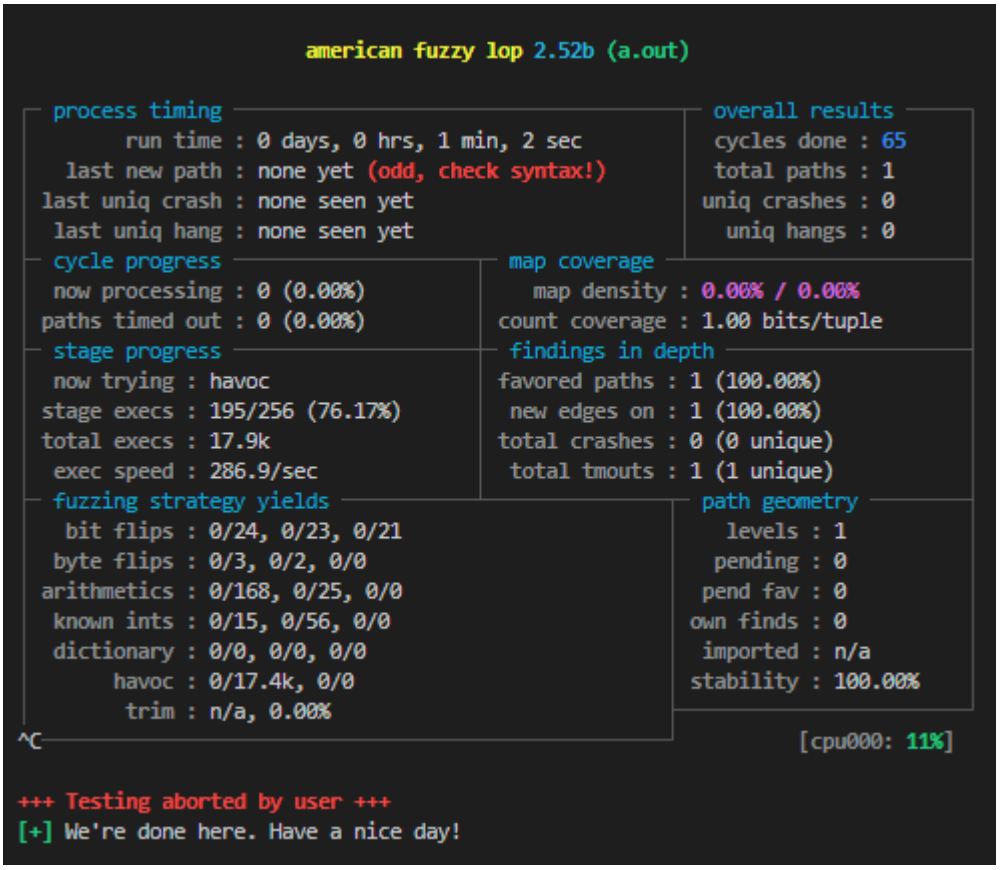
A partir du fichier **exemple3.c** nous remarquons que l'on fait un accès sur T à un indice sortant de l'intervalle de T, **T[200]** pour la condition **if (x == 42)**.

On ne peut pas reprendre le fichier **test1.txt** des exemples précédents car il possède la valeur **42**, qui correspond au cas où le programme devrait planter alors que AFL demande des entrées de tests correctes qui ne devraient pas causer de crash.

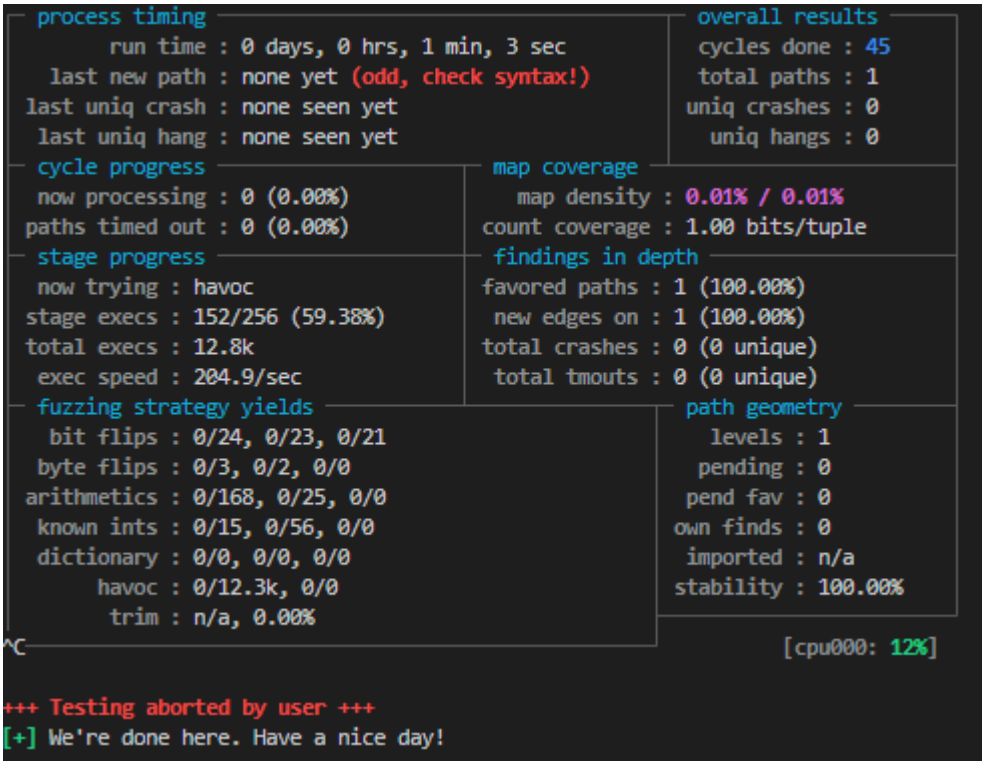
Donc on a créé un nouveau test correspondant à la valeur **20** qui sera le fichier **test2.txt**, pour lequel il ne devrait y avoir aucune erreur possible. On a également retiré le fichier **test1.txt** du répertoire in pour qu'**AFL** ne le prenne pas en compte.

Nous avons ensuite lancé **test2.txt** normalement pour vérifier son bon fonctionnement, avec les commandes **gcc exemple3.c** et **./a.out in/test2.txt**. Nous avons remarqué que cela n'affichait bien aucune erreur.

En utilisant **AFL** via la commande **./runAFL.sh exemple3.c**, et après environ une minute d'exécution, nous n'avons obtenu aucune erreur :



Cela était un résultat attendu car on retrouve le même cas que **exemple2.c**.
Ainsi nous avons décidé de lancer **exemple3.c** avec **AddressSanitizer** via la commande **./runAFL-Asan.sh exemple3.c**. Malgré tout, nous n'avons obtenu aucune erreur :



Cela s'explique par le comportement "mutationnel" d'**AFL** qui génère des valeurs aléatoires selon l'input précédent, en commençant par les valeurs données dans **in** (dans notre cas a partir de **20**). Mais ici, cela devrait provoquer un bug seulement pour une valeur particulière, **42** alors il est assez difficile de tomber sur ce cas. Donc il faudrait laisser tourner AFL plus longtemps pour espérer tomber dessus, mais il y seulement une probabilité de **1/(2^32)** d'avoir ce cas, ce qui est très faible.

Exercice 2 : Créer et retrouver des vulnérabilités dans du code

Etant donné qu'un large éventail de cas typiques de comportements indéfinis (division par 0, déréférencement ou libération de pointeurs nuls, débordement de tableau, de pile, de tas, récursion infinie …) est déjà abordé dans les autres exercices du TP, et par manque de temps, nous avons préféré nous concentrer sur ces autres exercices ainsi qu'à la mise en oeuvre d'AFL sur le projet.

Exercice 3 : Un exemple plus gros

Question 1 :

Après compilation avec la commande **gcc -g -o imgRead imgRead.c** et l'exécution sans argument avec **./imgRead**, on obtient le message **no input file**. On remarque donc que le programme **imgRead** a d'abord besoin d'un fichier d'entrée pour fonctionner.

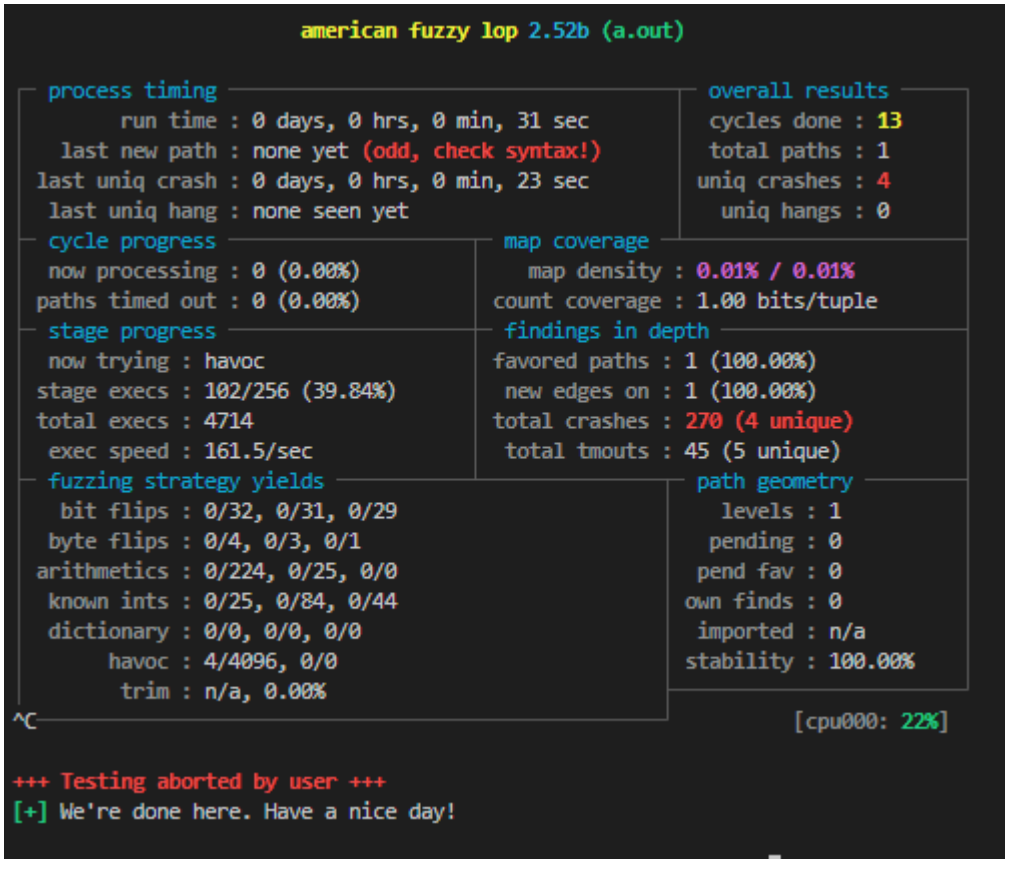
Question 2 :

Après exécution avec les fichier d'entrée **input1.txt** et **input2.txt** on ne remarque pas grand chose, il n'affiche rien et ne fait aucune erreur.

.

Question 3 :

Pour le lancement d'**AFL** avec le fichier d'entrée **input1.txt** pendant 30 secondes, on obtient 4 crashes uniques :



Question 4 :

Utilisation de **gdb** sur l'exécutable **imgRead** avec les fichiers d'entrée correspondant aux crashs identifiés par AFL. On obtient plusieurs erreurs :

- Une **division par 0**, car on voit **int size3= img.width/img.height;** alors que **img.height** est égal à **0**

```
(gdb)  run out/crashes/id\:000002\,sig\:06\,src\:000000\,op\:havoc\,rep\:128
Starting program: /home/b/bodartm/L3/S6/MOCA/TP7/TP-ALF/Vulnerable/imgRead out/crashes/id\:000002\,sig\:06\,src\:000000\,op\:havoc\,rep\:128

       Header  width   height  data

       D       226     0

Program received signal SIGFPE, Arithmetic exception.
0x00005555555552e6 in ProcessImage (filename=0x7fffffffdf62 "out/crashes/id:000002,sig:06,src:000000,op:havoc,rep:128") at imgRead.c:82
82                          int size3= img.width/img.height;
(gdb) p img.height
$1 = 0
```

- Plusieurs débordements de tableaux (sur tous les buffers) dus à des valeurs aberrantes de **img.width** et **img.height**, en particulier négatives, ce qui conduit à une allocation mémoire trop petites, étant donné que les buffers utilisent ces valeurs pour déterminer la taille du bloc à allouer

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555362 in ProcessImage (
    filename=0x7fffffffdee4 "out/crashes/id:000001,sig:06,src:000000,time:531,op:havoc,rep:128")
    at imgRead.c:59
59                              memcpy(buff1,img.data,sizeof(img.data));
(gdb) bt
#0  0x0000555555555362 in ProcessImage (
    filename=0x7fffffffdee4 "out/crashes/id:000001,sig:06,src:000000,time:531,op:havoc,rep:128")
    at imgRead.c:59
#1  0x0000555555555548 in main (argc=2, argv=0x7fffffffdbf8) at imgRead.c:133
(gdb) p/d size1
$1 = -2122594318
```

- Des erreurs d'écrasement de pile (stack smashing) causés par deux instructions qui écrivent un caractère bien au-delà de la taille maximale des buffers 3 et 4

```
(gdb) run out/crashes/id:000001,sig:06,src:000000,time:531,op:havoc,rep:128
Starting program: /home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead out/crashes/id:000001,sig:06,src:000000,time:531,op:havoc,rep:128

       Header  width   height  data

       bp@   ▯▯▯▯|    -2130739448     8145130
Size1:-2122594318

       Header  width   height  data

       @       418742016       31523063        1@
Size1:450265079

       Header  width   height  data

       ▯▯|     -1895825315     256
Size1:-1895825059
*** stack smashing detected ***: terminated

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
```

- Une erreur de **stack overflow** (débordement de pile) due à l'appel à **stack_operation()** qui réalise une récursion infinie en allouant un tableau à chaque itération (si **size4** est pair)

- Une erreur de **heap overflow** (débordement de tas) due à une boucle infinie qui alloue de la mémoire pour buff5 sans la libérer (si **size4** est impair)

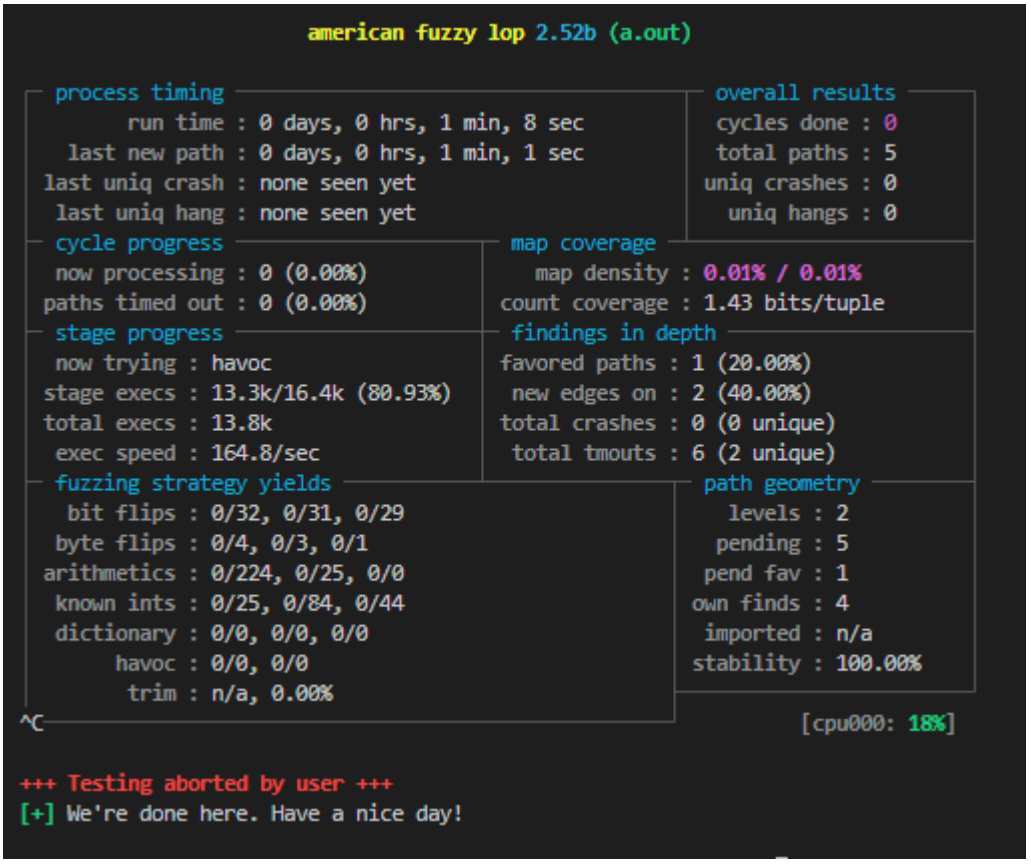
```
(gdb) run out/crashes/id:000000,sig:06,src:000000,time:422,op:havoc,rep:64
Starting program: /home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead out/crashes/id:000000,sig:06,src:000000,time:422,op:havoc,rep:64

        Header  width   height  data

        65536   169675274

mVUUUU
buffer overflow in buff1 : PASSED
free errors in buff1 : PASSED
buffer overflow in buff2 : PASSED
divide by zero for size3 : PASSED
buffer overflow in buff4 : PASSED
OOBR read past stack/heap buffer : PASSED
OOBW write past stack/heap buffer : PASSED

Program terminated with signal SIGKILL, Killed.
The program no longer exists.
```

- Un problème de buffer overflow identifié uniquement grâce à l'address sanitizer, lié à la déclaration des champs la **struct Image** avec des tableaux de taille trop petite

```
(gdb) run out/crashes/id:000000,sig:06,src:000000,time:418,op:havoc,rep:8
Starting program: /home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead out/crashes/id:000000,sig:06,src:000000,time:418,op:havoc,rep:8
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

        Header  width   height  data
=================================================================
==1705==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fffffffda58 at pc 0x7ffff7627dcb bp 0x7fffffffd880 sp 0x7fffffffcff8
READ of size 31 at 0x7fffffffda58 thread T0
    #0 0x7ffff7627dca in printf_common ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors_format.inc:546
    #1 0x7ffff7628dec in __interceptor_vprintf ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:1600
    #2 0x7ffff7628ee6 in __interceptor_printf ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:1658
    #3 0x555555556650 in ProcessImage /home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead.c:48
    #4 0x555555557019 in main /home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead.c:142
    #5 0x7ffff73bd082 in __libc_start_main ../csu/libc-start.c:308
    #6 0x55555555634d in _start (/home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead+0x234d)

Address 0x7fffffffda58 is located in stack of thread T0 at offset 56 in frame
    #0 0x5555555564c7 in ProcessImage /home/alexdet/MOCA/TP_Fuzzer/Vulnerable/imgRead.c:29

  This frame has 2 object(s):
    [32, 56) 'img' (line 31) <== Memory access at offset 56 overflows this variable
    [96, 106) 'buff3' (line 87)
```

<u>Question 5 :</u>

Lancement d'**AFL** sur le programme corrigé, après 1 minute de lancement nous aucun crash :

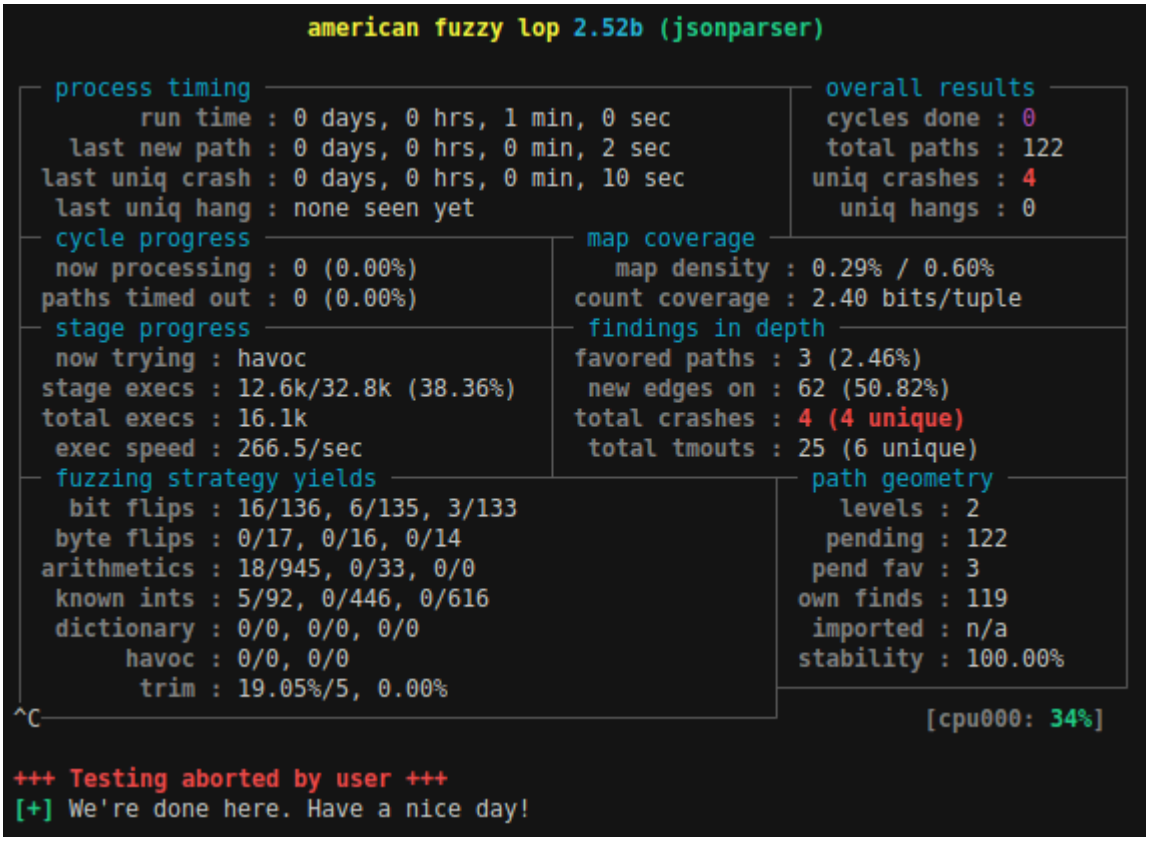


Exercice 4 : Fuzzer un parser Json

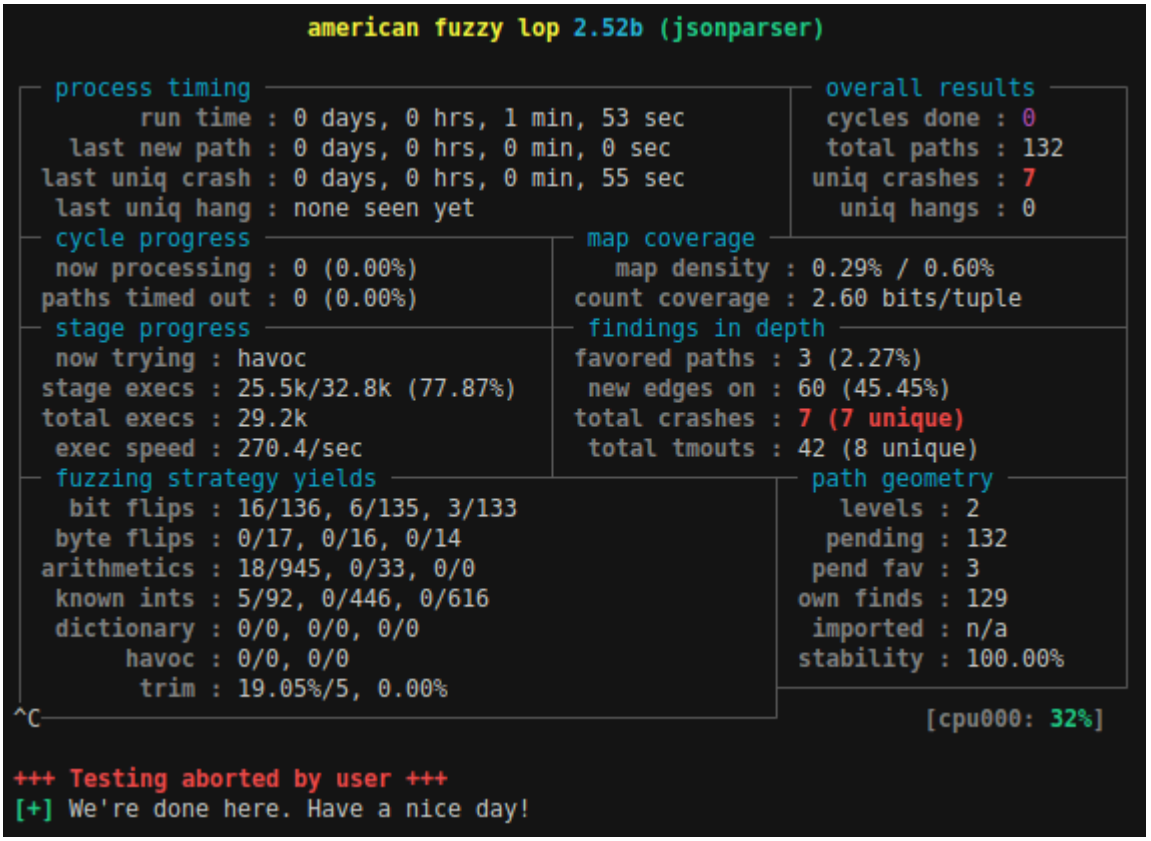
Question 1 :

Lors des tests des 2 fichiers exécutables **jsonparser** et **jsonparser_ASAN** avec les fichiers de tests du répertoire **regular_input**, on n'a aucune erreur pour **jsonparser** lors de l'exécution, mais des erreurs mémoire à chaque test pour **jsonparser_ASAN**.

On a réglé les erreurs sur le **main.c** car il manquait le **'\0'** à la fin du tableau avec son allocation. Après ça on lance **AFL** sur le **jsonparser** et on obtient 4 crashes uniques :



Maintenant, pour le cas de **jsonparser_ASAN** on obtient :



On remarque sur les crashes trouvés pour les 2 exécutable que l'on a 4 sources de plantage introduites dans le code (4 cas de "Fuzzgoat Vulnerability") :

- un premier cas où le tableau json est vide **[]** : une instruction de **free()** erronée est ajoutée et provoque des erreurs mémoire plus tard, lorsque le programme accèdera à nouveau au pointeur dont le bloc a été libéré ;
- un pointeur est maladroitement décrémenté dans le deuxième cas, ce qui provoque des problèmes de free() à plusieurs endroits, plus loin dans le code, car le nouveau pointeur est alors nul ;
- dans le troisième cas, un pointeur est inutilement décrémenté si une chaîne de caractères (l'un des champs du parser) est vide, causant là encore une erreur de **free()** plus loin ;
- enfin, dans le cas où on a une chaîne avec un seul caractère, on déclare un pointeur nul, qu'on déréférence, ce qui mène à une erreur de segmentation.

Nous les avons corrigés en respectant les indications en commentaires. Concrètement, ces cas d'erreurs menaient systématiquement à des erreurs de free dans gdb, comme le montre la capture suivante :

```
(gdb) run out/crashes/id:000000,sig:06,src:000000,time:1505,op:havoc,rep:8
Starting program: /home/alexdet/MOCA/TP_Fuzzer/JsonParser/jsonparser out/crashes/id:000000,sig:06,src:0
00000,time:1505,op:havoc,rep:8
""
------------------------------

string:
free(): invalid pointer

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff7c9d859 in __GI_abort () at abort.c:79
#2  0x00007ffff7d0826e in __libc_message (action=action@entry=do_abort,
    fmt=fmt@entry=0x7ffff7e32298 "%s\n") at ../sysdeps/posix/libc_fatal.c:155
#3  0x00007ffff7d102fc in malloc_printerr (str=str@entry=0x7ffff7e304c1 "free(): invalid pointer")
    at malloc.c:5347
#4  0x00007ffff7d11b2c in _int_free (av=<optimized out>, p=<optimized out>, have_lock=0)
    at malloc.c:4173
#5  0x0000555555557e57 in json_value_free_ex (settings=settings@entry=0x7fffffffd9e0,
    value=0x5555555628b0) at jsonparser.c:302
#6  0x000055555555e32e in json_value_free_ex (value=<optimized out>, settings=0x7fffffffd9e0)
    at jsonparser.c:222
#7  json_value_free (value=<optimized out>) at jsonparser.c:1080
#8  0x00005555555556d2 in main (argc=<optimized out>, argv=<optimized out>) at main.c:166
```

Exercice 5 : Utilisation de AFL sur votre projet

Pour faciliter le fuzzing de notre programme nous avons créé une nouvelle target qui se charge de compiler notre programme avec un des compilateurs afl (gcc ou clang, fast si possible) en fonction de l'installation. Le fuzzing lancé s'appuie sur un cas de base simple avec quelques mots, **hugo.txt**. On stocke le résultat du fuzzing dans un sous dossier de auto_test.

Pour lancer cette règle, il suffit d'exécuter **`make afltest`**. Pour une couverture plus grande et pour vérifier la robustesse de notre programme, on choisit d'ajouter l'option **-D** à AFL, qui fera des inversions de bits et octets, et testera aussi un dictionnaire d'entiers intéressants.

L'activation de cette exploration nous permet de trouver rapidement un cas de crash : notre programme n'est pas robuste aux caractères non ASCII. Afin de corriger ce problème, nous avons décidé de traiter les caractères non ASCII comme des séparateurs. Ainsi, nous avons ajouté deux procédures, **is_ascii** et **skip_separators** dans le fichier **word_tools.c** :

- la première permet, comme son nom l'indique de vérifier si un caractère est un caractère ASCII alphabétique;

- la seconde est une factorisation du code de **next_word** et permet de sauter les séparateurs (caractères non alphabétiques).

On a également choisi de compiler nos programmes avec ASAN lorsqu'on fait du fuzzing, pour détecter plus de crashs, notamment liés à des comportements indéfinis dans la mémoire. De cette manière, l'utilisation combinée d'ASAN et AFL nous a permis de trouver encore quelques fuites de mémoire au niveau de la sérialisation des dictionnaires.

En effet, nous avons constaté des collisions dans la table de hachage, pour certaines entrées telles que **orwell.txt** et **wells.txt**. Dans ce cas, la sérialisation des mots concernés échouait, et notre stratégie de libération des blocs mémoire (libération de **serialized_dico** dans un premier temps, afin de libérer les structures mot_data_t communes aux dictionnaires **dictionary** et **copieDico**) ne traitait pas ces mots. Nous avons donc décidé de libérer directement la structure d'**emplacement_t** associée à un mot en cas de collision (dans **serializeDico** de **serialization.c**).