Chanfreau - Marcé

Compte rendu Architecture Matérielle

Objectif :

Le TP d'introduction aux architectures ARM-x86-x64 a pour objectif d'explorer les aspects principaux des basses couches des logiciels, fortement contraintes par l'architecture matérielle. Les trois architectures ciblées sont ARM, x86 et x64, représentatives de deux tendances historiques : RISC et CISC. Le TP se concentrera sur la manière dont le compilateur C organise les appels de fonctions et le stockage des données, ainsi que sur certains aspects de la sécurité qui en dépendent. Pour cela, la stratégie adoptée sera l'observation du fonctionnement du programme "en live" au moyen d'un debugger, ce qui suppose une bonne compréhension du langage d'assemblage.

Préparation :

Téléchargement des ressources : **tp-3a-fisa-arm-x86-x64-ressources.tar**
Téléchargement du compilateur : **gcc-arm-none-eabi-9-2019-q4-major-x86_64-linux**
Extraction du compilateur Cortex-M: **make gcc-arm**

Prise en main :

Qemu

Affichez les machines supportées par qemu : **qemu-system-arm -M help = lm3s6965evb Stellaris LM3S6965EVB**

Compiler les fichiers: **qemu-system-arm -M lm3s811evb -nographic -no-reboot -kernel arm.bin -s -S**

Gdb

Ouverture de gdb: **./arm-gdb OU gdb**
Lire les symboles du fichier source : **file arm.elf OU file x86.bin**
Connexion manuelle à qemu : **target remote localhost:1234**

Script pour tout lancer (ARM) : ./setup-arm-gdb-qemu.sh

Commandes:

- Découpage fenêtre en 3 : layout split
- Brakpoint : break [Nom_fichier]:ligne ou break [Nom_fichier]:function
- Exécute le code : continue
- Lancer le programme : run
- Avance l'execution : step
- Etat des registres : info registers
- Supprimer breakpoint : delete [nb_BP]
- Afficher la valeur d'un registre : print $[nom du registre]
- Sélectionner le code : focus src
- Sélectionner la commande : focus cmd
- Sélectionner l'assembleur : focus asm
- Quitter la vue 3 fenêtre : ctrl-x+a

EXP01: variables locales

Question 1

Pour les 3 architectures proposées, étudiez où sont stockées les variables locales aa, bb, cc, dd, ee de la fonction sub01.

Question 2

Étudiez de plus comment l'espace de stockage pour ces variables est réservé et comment il est libéré.

Architecture x86 :

```
(gdb) print &aa
$1 = (int *) 0xffffcc5c
(gdb) print &bb
$2 = (int *) 0xffffcc60
(gdb) print&cc
$3 = (int *) 0xffffcc64
(gdb) print &dd
$4 = (int *) 0xffffcc68
(gdb) print &ee
$5 = (int *) 0xffffcc6c
(gdb)
```

Les variables locales aa, bb, cc, dd, ee de la fonction sub01 sont stockés dans la pile de la mémoire vive.

En architecture x86, l'adresse de la pile commence à la fin de la plage d'adresse allouée pour les données.

La plage d'adresse débute à 0xFFFFFFFF

Architecture x86_64 :

```
Breakpoint 1, sub01 () at main.c:8
(gdb) print &sub01
$1 = (void (*)()) 0x555555555187 <sub01>
(gdb) print &aa
$2 = (int *) 0x7fffffffda5c
(gdb) print &bb
$3 = (int *) 0x7fffffffda60
(gdb) print &cc
$4 = (int *) 0x7fffffffda64
(gdb) print &dd
$5 = (int *) 0x7fffffffda68
(gdb) print &ee
$6 = (int *) 0x7fffffffda6c
```

Dans l'architecture x86_64 la pile est également située à la fin de la plage d'adresse allouée.

De plus, la pile grandit et vas vers des adresses plus basses lors de l'ajout de nouvelles données.

La fonction sub01 est stockée dans l'espace d'adressage du processus qui exécute le programme.

La plage de la pile va de 0x7FFFFFFFFFFF à 0x7FFF00000000

ARM Cortex-M3 :

```
$7 = (void (*)()) 0x4d4 <sub01>
(gdb) print &aa
$8 = (int *) 0x20000fec
(gdb) print &bb
$9 = (int *) 0x20000fe8
(gdb) print &cc
$10 = (int *) 0x20000fe4
(gdb) print &dd
$11 = (int *) 0x20000fe0
(gdb) print &ee
$12 = (int *) 0x20000fdc
```

Les variables sont stockées dans la pile, soit dans l'espace d'adresse RAM allant de 0x2000 0000 à 0x3FFFFFFF.

La fonction sub01 est chargé à l'adresse 0x4d4 ce qui correspond à la ROM qui a une plage d'adresse de 0x0 à 0x1FFFFFFF.

L’espace de stockage pour les variables locales est réservé sur la pile à des emplacements consécutifs. Pendant l'exécution de la fonction, les variables peuvent être lues et modifiées en utilisant des adresses relatives à l'adresse de base de la pile.
Lorsque la fonction se termine, l'espace alloué pour les variables locales est libéré.

Question 3

En quoi le stockage des variables locales a-t-il été amélioré ?

L’optimisation make CFLAGS=-O1 :

- Utilisation des registres pour stocker les variables locales ➔ plus rapide pour accéder à la mémoire
- Empilement de plusieurs variables dans un seul espace mémoire ➔ améliorer la rapidité
- Réorganiser les variables locales pour minimiser les accès mémoires ➔ améliorer les performances

L’optimisation 1 permet au compilateur de sauter certaines instructions qui n’ont pas d’impact majeur sur le code et notamment sur la sortie.
Ainsi, toutes les variables non utilisées ne sont pas stockées dans la mémoire pour éviter d’encombrer de l’espace de stockage inutilement.
Typiquement dans notre programme, les variables aa, bb, cc et dd ne sont pas stockées car elles n’ont aucun impact sur l’activité du code, à l’inverse de ee qui est projetée sur un périphérique avec expose en sortie.

EXP02: arguments d’une fonction

Question 4

Comment la fonction seed() retourne-t-elle sa valeur ?

```
>0x56556202 <seed>                    endbr32
 0x56556206 <seed+4>                  push   %ebp
 0x56556207 <seed+5>                  mov    %esp,%ebp
 0x56556209 <seed+7>                  call   0x5655621b <__x86.get_pc_thunk.ax>
 0x5655620e <seed+12>                 add    $0x2dca,%eaxnk.ax>
 0x56556213 <seed+17>                 mov    0x8(%ebp),%eax
>0x56556216 <seed+20>                 add    $0x1,%eax
 0x56556219 <seed+23>                 pop    %ebp
 0x5655621a <seed+24>                 ret
 0x5655621b <__x86.get_pc_thunk.ax>   mov    (%esp),%eax
 0x5655621e <__x86.get_pc_thunk.ax+3> ret
```

La fonction seed() retourne sa valeur en la plaçant dans le register EAX (mov 0x8(%ebp),%eax).

endbr 32 : Sécurise l’exécution contre les attaques de débordements de tampon.
push %ebp : Sauvegarde de la valuer du registre ebp sur la pile.
mov %esp,%ebp : Copie la valeur du registre esp dans le registre ebp
add $0x2dca,%eax : Ajour de l’offset 0x2dca à la valeur stocké dans le registre eax

mov 0x8(%ebp),%eax : Copie la valeur de l'argument i dans le registre eax
add $0x1,%eax : Ajoute 1 à la valeur stockée dans le registre eax
pop %ebp : Restaure la valeur du registre ebp à partir de la pile.
ret : Retourne à l'instruction appelante

Question 5

Comment les arguments aa, bb, cc, dd, ee sont-ils passés à la fonction sub01() ?

La fonction sub01 utilise le registre eax qui contient la valeur de nos variables (voir fonction seed précédente) et le pointeur de base de la pile ebp. En effectuant un décalage en hexadécimal par rapport à cette référence, on stocke la valeur finale de nos variables (en sortie de sub01) après avoir effectué les opérations d'addition et de multiplication correspondant au code.

Question 6

En quoi l'appel à la fonction sub01() a-t-il été amélioré ?

L'optimisation make CFLAGS=-O2 :

- Utilisation des registres pour stocker les variables
- Variables les plus fréquemment utilisées stocké dans la mémoire cache.
- Vectorisation pour réaliser plusieurs tâches à la fois

Avec cette nouvelle optimisation, la fonction seed() est remplacée par de l'arithmétique pure et l'exécution de la fonction sub01 est beaucoup plus rapide. Les étapes d'addition et multiplication des registres ne sont même plus effectués par le compilateur.

EXP03: débordement de pile

Question 7

Vérifiez pour chaque architecture si la valeur soumise à **expose()** est bien la valeur prévue pour le dernier élément.

ARM:

La fonction expose fait un printf de l'argument placé en argument.
Sur l'ARM, la valeur du dernier élément du tableau est bien 37068. Cette observation signifie qu'il n'y a pas présence d'un mécanisme de protection d'intégrité de la pile dans ce processeur.

```
chanfreau@insa-20485:~/INSA/Architecture matérielle/TP1/EXP03$ qemu-system-arm -M lm3s811evb -nographic
-no-reboot -kernel arm.bin -s -S
37068=0x000090cc
```

x86 et x86-64 :

En revanche, pour les architecture x86 et x86_64 l'exécution du programme retourne une erreur. Cela signifie qu'il y a présence d'un mécanisme de protection de l'intégrité de la pile dans ces processeurs. Cette configuration permet de gérer les débordements de pile.

```
chanfreau@insa-20485:~/INSA/Architecture matérielle/TP1/EXP03$ ./x86.bin
*** stack smashing detected ***: terminated
Abandon (core dumped)
chanfreau@insa-20485:~/INSA/Architecture matérielle/TP1/EXP03$ ./x86_64.bin
*** stack smashing detected ***: terminated
Abandon (core dumped)
```

On observe tout de même que la dernière valeur du tableau correspondant bien à celle attendue grâce à un print effectué juste avant interruption du programme.

```
(gdb) print i
$4 = 37068
(gdb) print &i
$5 = (int *) 0xffffcc54
```

Question 8

Pour chaque architecture, étudiez si le déroulement du programme est correct. Sinon, à partir de quelle instruction y a-t-il une déviation ?

Question 9

Quelle est la cause de l'incident ?

ARM :

Après l'exécution du programme, celui-ci ne se termine pas et continue l'exécution d'une boucle infinie.

```
(gdb) continue
Continuing.
```

```
0x9438  movs   r0, r0
0x943a  movs   r0, r0
0x943c  movs   r0, r0
0x943e  movs   r0, r0
0x9440  movs   r0, r0
0x9442  movs   r0, r0
```

En étudiant le code d'assemblage, nous nous apercevons que la zone mémoire allouée dans la pile pour le tableau a été dépassée par l'écriture de nos valeurs. Ainsi, nous avons malencontreusement écrasé une partie de la pile contenant potentiellement des informations importantes telles que l'adresse de retour de la fonction. La ligne suivante en surbrillance montre que la partie suivante du code exécuté se trouve à une adresse dont nous avons écrasé la valeur. La conséquence est que le code effectue un saut à l'adresse 0x941d correspondant à une partie de la ROM «vide » (composée de 0 uniquement) et qui s'exécute indéfiniment.

```
0x338 <sub01+20>        bne.n  0x32c <sub01+8>
0x33a <sub01+22>        add    r0, sp, #4
0x33c <sub01+24>        bl     0x308 <sub02>
0x340 <sub01+28>        add    sp, #44 ; 0x2c
0x342 <sub01+30>        ldr.w  pc, [sp], #4
0x346 <main>            push   {r3, lr}
0x348 <main+2>          mov.w  r0, #4096       ; 0x1000
```

```
(gdb) print /x *0x20000FF4
$11 = 0x941d
```

x86 :

Pour le processeur x86, le programme se termine par un saut dans le gestionnaire d'erreur (__stack_chk_fail), ce qui a pour conséquence de terminer l'exécution. Plus précisément, l'erreur est due a une tentative de lecture dans une zone non prévue à cet effet (extérieure au tableau dans notre cas d'étude).
L'erreur se produit à partir de la ligne xor %gs : 0x14,%ecx qui nous renvoie par la suite à l'adresse 0x5655628d (call : gestionnaire d'erreur).

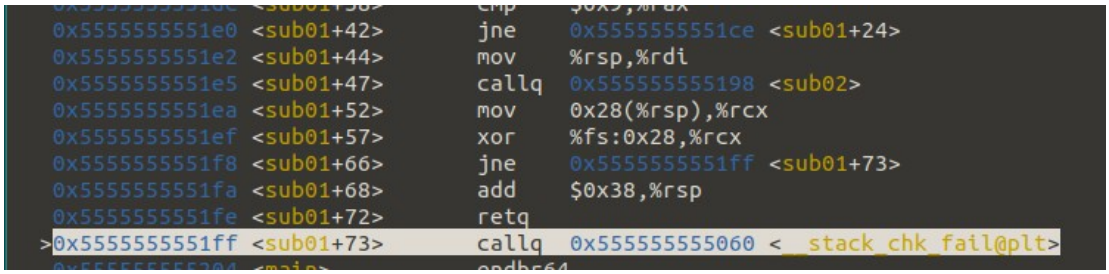
```
(gdb) run
Starting program: /home/chanfreau/INSA/Architecture mat rielle/TP1/EXP03/x86.bin
*** stack smashing detected ***: terminated

Program received signal SIGABRT, Aborted.
(gdb) f549 in __kernel_vsyscall ()
```

```
 0x56556261 <sub01+28>   add    %ecx,%edx
 0x56556263 <sub01+30>   mov    %edx,0x8(%esp,%eax,4)
 0x56556267 <sub01+34>   add    $0x1,%eax
 0x5655626a <sub01+37>   cmp    $0x9,%eax
 0x5655626d <sub01+40>   jne    0x5655625c <sub01+23>
 0x5655626f <sub01+42>   lea    0x8(%esp),%eax
 0x56556273 <sub01+46>   push   %eax
 0x56556274 <sub01+47>   call   0x56556225 <sub02>
 0x56556279 <sub01+52>   add    $0x4,%esp
 0x5655627c <sub01+55>   mov    0x2c(%esp),%ecx
 0x56556280 <sub01+59>   xor    %gs:0x14,%ecx
 0x56556287 <sub01+66>   jne    0x5655628d <sub01+72>
 0x56556289 <sub01+68>   add    $0x3c,%esp
 0x5655628c <sub01+71>   ret
>0x5655628d <sub01+72>   call   0x56556360 <__stack_chk_fail_local>
 0x56556292 <main>       endbr32
```

X86_64 :

Pour le processeur x86_64, le programme se termine également par un saut dans la gestionnaire d'erreur (__stack_chk_fail) et provoque la fin de son exécution. L'erreur est due au même débordement mémoire que pour le x86.
L'erreur se produit à partir de la ligne xor %fs : 0x28,%rcx qui nous renvoie à l'adresse 0x55555555551ff (call : gestionnaire d'erreur).

```
__stack_chk_fail () at stack_chk_fail.c:23
stack_chk_fail.c: Aucun fichier ou dossier de ce type.
```



En conclusion:

L'incident est causé par une optimisation incorrecte du compilateur. Lorsque le compilateur optimise le code, il effectue des modifications pour accélérer l'exécution, mais cela peut parfois entraîner des erreurs si ces modifications sont incorrectes.

Dans ce cas, l'optimisation a entraîné un accès incorrect à la mémoire en essayant d'accéder à des éléments au-delà de la limite du tableau. Cela a conduit à des résultats incorrects et à un comportement imprévisible du programme.