Année Universitaire 2020-2021 session 1 de printemps

Parcours : TCY Master CSI **UE :** 4TCY801U

Épreuve : Examen Sécurité logicielle

Date : 7 mai **Heure :** 14h30 **Durée :** 3h

Modalités : Épreuve sans document, calculatrice autorisée

Épreuve de : Mr Samuel Thibault **4 pages**

université de BORDEAUX

Collège Sciences et technologie

Exercice 1. Un crash bizarre...

J'ai un serveur web qui l'autre jour a produit un segfault étrange. J'ai regardé l'état du processus avec `pframe`, et observé ceci :

```
0xffffc158            0xffffc120
0xffffc154            0xffffc120
0xffffc150         sp 0xffffc120
0xffffc14c            0xffffc120
0xffffc148            0xffffc120
0xffffc144            0xffffc120
0xffffc140            0x90909090
0xffffc13c            0x90909090
0xffffc138            0x90909090
0xffffc134            0x80cd0000
0xffffc130            0x0001b800
0xffffc12c            0x000042bb
0xffffc128            0x90909090
0xffffc124            0x90909090
0xffffc120 ip         0x90909090
```

Q1.1 Décrivez au maximum ce qui est affiché (pas les causes/conséquences, juste ce qui est affiché)

Q1.2 Pourquoi le programme a-t-il produit un segfault ?

Q1.3 Pourquoi je devrais m'inquiéter, quel genre de chose est survenue précisément ?

Q1.4 Pourquoi la valeur `0xffffc120` est répétée plusieurs fois ? Laquelle est la plus importante ?

Q1.5 Pourquoi `80` est affiché à gauche de `cd` plutôt qu'à droite ?

Q1.6 Pourquoi les `00` auraient pu poser problème ?

Q1.7 Donnez un exemple de code C montrant le type de bug de mon serveur web qui a pu permettre cela.

TSVP...

Exercice 2. Lecture d'assembleur

Un programme, dont je n'ai pas le code source, contient la fonction suivante :

```
00000000 <f>:
   0x0: mov    0x4(%esp),%ecx
   0x4: mov    0x8(%esp),%edx
   0x8: test   %edx,%edx
   0xa: jle    22 <f+0x22>
   0xc: mov    %ecx,%eax
   0xe: lea    (%ecx,%edx,4),%ecx
  0x11: mov    $0x0,%edx
  0x16: add    (%eax),%edx
  0x18: add    $0x4,%eax
  0x1b: cmp    %ecx,%eax
  0x1d: jne    16 <f+0x16>
  0x1f: mov    %edx,%eax
  0x21: ret
  0x22: mov    $0x0,%edx
  0x27: jmp    1f <f+0x1f>
```

Q2.1 Quelles instructions récupèrent les arguments de la fonction ?

Q2.2 Repérez le corps de la boucle, que fait-elle ?

Q2.3 Quel registre contient la borne de la boucle, comment est-elle calculée ?

Q2.4 Que fait la fonction, en fait ?

Q2.5 Dessinez l'état de la pile au moment de l'entrée de la fonction.

Q2.6 À quoi servent les instructions aux adresses 0x22 et 0x27 ? Comment le compilateur aurait-il pu mieux optimiser ?

TSVP...

Exercice 3. Attaquons la liaison statique !
Cet exercice est inspiré de l'article PoC||GTFO 18 :06 « RelroS : Read Only Relocations for Static ELF »

Questions de cours

Q3.1 Faites un dessin rappelant l'agencement d'un processus en mémoire, en faisant apparaître les bibliothèques, les données, la pile, le tas, le texte.

Q3.2 Précisez sur le dessin les droits r/w/x des différentes zones mémoire.

Q3.3 Pourquoi est-il utile grâce à NX d'enlever le droit `x` sur la pile, qu'est-ce que cela empêche ?

Q3.4 Expliquez comment fonctionne le canary du stack smashing protection et de quoi il protège, en vous aidant par exemple de dessins.

Regardons un peu...

J'ai le programme suivant :

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
        printf("%p %p\n", &argc, main);
}
```

Que je compile et exécute ainsi :

```
$ gcc test.c -o test
$ ./test
0x7ffd34beb2dc 0x563d64ece135
$ ./test
0x7ffe70867aec 0x56418b260135
$ nm ./test | grep " main$"
0000000000001135 T main
```

Le résultat est complètement différent d'une exécution à l'autre...

Q3.5 Pour la première adresse affichée par chaque exécution, dans quelle partie du processus se situe-t-elle ?

Q3.6 Toujours pour cette première adresse, pourquoi est-ce utile qu'elle soit différente d'une exécution à l'autre, qu'est-ce que cela empêche ?

Q3.7 Pour la deuxième valeur affichée, qu'est-ce que le compilateur fait pour qu'elle puisse changer d'une exécution à l'autre, et qu'est-ce que le système fait alors ?

Q3.8 Toujours pour cette deuxième valeur, pourquoi fait-on cela, qu'est-ce que cela empêche ?

Désormais je compile avec le flag `-static` :

```
$ gcc test.c -o test -static
$ ./test
0x7ffffc66ae7c 0x401c2d
$ ./test
0x7ffd4aae145c 0x401c2d
$ nm ./test | grep " main$"
0000000000401c2d T main
```

Apparemment ce n'est plus tout à fait pareil...
Le flag `-static` active la liaison statique : au lieu que le code des bibliothèques soit chargé dynamiquement à l'exécution, séparément du programme (liaison dynamique habituelle), il est inclus *dans* le programme à la compilation, à côté de la fonction `main` (liaison statique).

Q3.9 Refaites le dessin de la première question pour cette nouvelle situation.

Nous avons vu en cours sur la GOT (Global Offset Table) que d'habitude (avec la liaison dynamique) on détermine les adresses des fonctions des bibliothèques de manière paresseuse, au moment de leur premier appel. On note alors dans la GOT cette adresse pour que les appels suivants se fassent directement.

Q3.10 Pourquoi, avec la liaison dynamique, est-on obligé de déterminer ces adresses de manière dynamique plutôt que dès la compilation du programme, dans quelle mesure est-ce lié au changement de la deuxième valeur imprimée par notre programme ?

Q3.11 Pourquoi avec la liaison statique il n'y a plus besoin de cela a priori ?

Il se trouve que même avec la liaison statique il reste quelques fonctions pour lesquelles on détermine leur adresse de manière paresseuse : les fonctions avec l'attribut `ifunc`. Ce sont les fonctions pour lesquelles on dispose de plusieurs implémentations selon les capacités du processeur (sse, sse2, avx, etc.). Lorsque le programme appelle la fonction pour la première fois, un bout de code observe d'abord sur quel processeur on tourne, et écrit alors dans la GOT l'adresse de l'implémentation qui profite le mieux des capacités du processeur. Les appels suivants se feront alors directement. Par exemple, les fonctions `strchr`, `strcmp`, `strcpy`, etc. profitent de cet attribut.

Q3.12 Décrivez comment une attaque peut profiter de cela : quel genre de faille dans le code du programme permet d'en profiter, et l'ordre dans lequel les opérations se produisent.

Q3.13 Indiquez quelle stratégie on pourrait utiliser pour à la fois continuer à utiliser automatiquement l'implémentation qui profite le mieux du processeur, tout en éliminant le vecteur de vulnérabilité.

`ret`