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...

Q1.1 On remarque une adresse répétée des tas de fois, il se trouve que c'est l'adresse vers laquelle ip pointe actuellement.

On remarque des 909090 qui font penser à des nops.

Est-ce qu'il y aurait des instructions ? En effect, on voit 80 cd, c'est CD 80 à l'envers, càd int 0x80, donc un appel système, ça sent le gaz.

Et des 9090 de nouveau.

Q1.2 Le programme a essayé d'exécuter l'instruction à l'adresse ip, mais probablement la pile n'est pas exécutable, d'où segfault.

Q1.3 Cela a tout l'air d'un débordement de tampon intentionel par un attaquant, avec du code qui a été injecté sur la pile, et l'adresse de retour qui a été écrasée avec l'adresse du code.

Q1.4 Elle est répétée pour que même si l'attaque est approximative, elle parvienne à écraser l'adresse de retour. Celle qui est importante, c'est 0xfffffc14c, que l'on vient de dépiler.

Q1.5 C'est parce que pframe affiche ce qui est en mémoire comme des nombres, interprétés en little-endian. Donc même si en mémoire c'est cd 80, interprété comme nombre little-endian cela apparait sous la forme 80 cd

Q1.6 Selon la fonction via laquelle l'attaquant injecte son code (printf, memcpy ou strcpy), cela aurait pu terminer la chaine copiée (strcpy) ou non (memcpy). Apparemment ce n'est pas le cas ici donc ça devait être un memcpy.

Q1.7

```
litligne() {
        char buf[16];
        gets(buf);
        return strdup(buf);
}
```

Exercice 2. Lecture d'assembleur

Q2.1 Ce sont les

```
mov 0x4(%esp),%ecx
mov 0x8(%esp),%edx
```

qui vont lire les arguments dans la pile, et les copient dans `%ecx` et `%edx`

Q2.2 On repère un `jne` entre 0x1d et 0x16 la boucle lit un entier 32bits en mémoire à l'adresse `%eax` et l'ajoute à `%edx`, puis avance `%eax` de 32 bits,

Q2.3 `%eax` est comparé à `%ecx` avant de faire le jne, c'est donc `%ecx` qui donne apparemment la fin de la boucle. elle a été calculée plus haut avec

`lea (%ecx,%edx,4),%ecx`

i.e.

`%ecx = %ecx + %edx * 4`

Il se trouve que `%ecx` et `%edx` sont à l'origine les paramètres de la fonction, donc a priori un début de tableau et le nombre d'éléments dans le tableau

Q2.4 Apparemment elle calcule simplement la somme des éléments d'un tableau

Q2.5

```
   nb elements
   ptr tab
sp @ret
```

Q2.6 C'est pour le cas où le tableau a 0 éléments, on retourne directement la valeur 0.

- `mov $0x0, %edx` aurait pu être remplacé par un `xor %edx, %edx`.
- Et puis plutôt que stocker dans `%edx`, autant stocker directement dans `%eax` et utiliser `ret` plutôt que `jmp 1f` qui réalise la même chose.
- le compilateur aurait aussi pu directement calculer la somme dans `%eax` plutôt que dans `%edx`, et donc initialiser eax à zéro dès le départ, et là seulement tester le nombre d'élément, et brancher directement sur ret s'il est négatif

Exercice 3. Attaquons la liaison statique !

Questions de cours

Q3.1 Q3.2

```
rw- pile
rw- données bibliothèques
r-- données RO bibliothèques
r-x bibliothèques
rw- tas
rw- données
r-- données RO
r-x texte
```

Q3.3 On l'a justement vu dans l'exercice 1, on y a vu une injection d'un shellcode dans un buffer alloué en variable locale, et le débordement du buffer dans l'adresse de retour a permis d'essayer de faire exécuter le shellcode. En empêchant l'exécutabilité de la pile, cette tentative a échoué. Cela n'empêche pas le débordement de tampon et le RoP, mais cela empêche de pouvoir exécuter du code injecté sur la pile.

Q3.4

```
@ret
canary: une valeur bien particulière
vars locals
```

S'il y a un débordement de tampon, le canary se fait écraser avant l'adresse de retour. avant d'exécuter `ret`, la fonction peut vérifier si le canary contient bien toujours la même valeur particulière. Cela n'empêche pas un attaquant bien informé de remettre la bonne valeur dans le canary, mais cela rend au moins l'attaque plus difficile.

Regardons un peu...

Q3.5 `0x7ff*` -> pile

Q3.6 Mettre la pile à un endroit différent d'une exécution à l'autre rend l'attaque de l'exercice 1 plus difficile : il faut commencer par déterminer l'adresse de la pile, pour savoir quelle addresse utiliser pour écraser l'adresse de retour.

Q3.7 `0x563d64 (main)` -> texte

Il a fallu que le compilateur rende le code position-indépendant, c'est-à-dire éviter d'utiliser des adresses absolues, ou alors sinon utiliser des *relocations* pour que la bonne adresse soit corrigée à l'exécution.

Le système peut alors placer le texte à une adresse aléatoire, et remplir les *relocations*, et répondre aux demandes de plt

Q3.8 On a vu qu'avec NX on pouvait empêcher l'exécution de code de la pile, et que la randomisation de la position de la pile le rendait plus difficile aussi, mais on pourrait à la place écraser l'adresse de retour avec l'adresse d'une fonction du programme, qui exécute un shell par exemple.

Randomiser la position du texte permet de rendre cela plus difficile : il faut d'abord déterminer où est le texe en mémoire avant de pouvoir écraser l'adresse de retour avec l'adresse de la fonction visée.

Q3.9

```
rw- pile
rw- tas
rw- données bibliothèques
rw- données
r-- données RO bibliothèques
r-- données RO
r-x bibliothèques
r-x texte
```

Q3.10 Une bibliothèque doit pouvoir être chargée dans n'importe quel processus, quel que soit l'ensemble des bibliothèques utilisées. Il faut donc que la bibliothèque puisse fonctionner n'importe où en mémoire de toutes façons, quelle que soit l'utilisation qu'on en fera.

Au moment de la compilation de la bibliothèque on n'a aucune idée de son utilisation future.

Au moment de la compilation du programme, on peut regarder quelles bibliothèques sont utilisées, mais elles pourraient être mises à jour, ce qui change leur taille, et donc leur arrangement possible en mémoire.

Le changement de valeur de l'adresse de `main`, càd le changement de la position du texte, peut aussi avoir son importance : il pourrait être placé n'importe où en mémoire, et notamment à l'endroit où on aurait pu vouloir placer la bibliothèque

Q3.11 En liaison statique, on accole la bibliothèque au programme au moment de la compilation, du coup au moment de la compilation on sait exactement quelle taille font les bibliothèques, comment elles sont arrangées en mémoire, etc. et donc on connait les adresses à la compilation, on peut les encoder en dur plutôt que via la GOT. Par contre du coup apparemment le chargement se fait toujours à la même adresse, c'est pas terrible. On pourrait plutôt compiler le code de manière repositionnable, et que ce soit tout le binaire statique qui puisse bouger en mémoire.

Q3.12 C'est le même principe que l'attaque GOT vue en cours : pour pouvoir écrire l'adresse de la fonction spécialisée, la GOT est forcément écrivable. Or en utilisant un débordement de tampon dans le tas, on peut choisir quelle valeur écrire où l'on veut en mémoire. On peut du coup écrire, dans l'entrée de la GOT pour

une telle fonction (par exemple `strcmp`), l'adresse d'une fonction que l'on cherche à exécuter. Au moment où le programme appelle la fonction `strcmp`, il se retrouve à appeler à la place la fonction ciblée.

Q3.13 On a vu en cours que `relro,now` permet de rentre la GOT non écrivable, en résolvant à l'initialisation du programme toutes les entrées de la GOT, ce qui prend du temps, mais permet de rendre la GOT complètement R/O pendant l'exécution du programme. On peut faire la même chose pour les fonctions ifunc : à l'initialisation du programme, on commence par regrder sur quel processeur on se trouve, et l'on résoud toutes les entrées ifunc de la GOT selon le résultat. On rend la GOT R/O, et l'on commence seulement l'exécution du programme. L'attaque ne pourra alors pas écrire dans la GOT.

`ret`