Année Universitaire 2022-2023 session 1 de printemps

Parcours : TCY Master CSI **UE :** 4TCY801U
Épreuve : Examen Sécurité logicielle
Date : 3 mai **Heure :** 9h **Durée :** 3h
Modalités : Épreuve sans document, calculatrice autorisée
Épreuve de : Mr Samuel Thibault **4 pages**

université de BORDEAUX

Collège Sciences et technologie

Note : soyez **précis** dans vos réponses. Pas forcément long, mais précis.

Il n'est pas fourni de table ASCII, mais le sujet contient suffisament d'informations pour retrouver ce dont vous avez besoin.

Exercice 1. Promenons-nous sur la pile

J'exécute dans `gdb` un programme dont une partie du code source est ainsi :

```
void f(void) {
        char s[16];
        gets(s);
        printf(s);
        printf(": %d\n", atoi(s));
}
```

J'ai placé un breakpoint sur `atoi` et j'ai tapé `123A` puis entrée, et `gdb` s'est arrêté au début de la fonction `atoi`. `pframe` me montre alors ceci :

```
0xffffc130           0x00000001
0xffffc12c           0x08049b17
0xffffc128           0x00000001
0xffffc124           0x00000001
0xffffc120           0x080eca20
0xffffc11c           0x08049b17
0xffffc118           0x00000001
0xffffc114    arg2   0xffffc130
0xffffc110    arg1   0x080a34a7
0xffffc10c    ret@   0x0804978c
0xffffc108      bp   0xffffc118
0xffffc104           0x00000000
0xffffc100           0x00000000
0xffffc0fc           0x54534150
0xffffc0f8           0x53534150
0xffffc0f4           0x08075300
0xffffc0f0           0x41333231
0xffffc0ec           0x00000001
0xffffc0e8           0x00002354
0xffffc0e4           0x00000000
0xffffc0e0           0xffffc0f0
0xffffc0dc         sp 0x08049735
```

Q1.1 Expliquez ce qu'est `ret@`

Q1.2 La fonction `f` ne prend pas de paramètre, pourtant `pframe` montre 2 paramètres, comment cela se fait-il ?

Q1.3 Vers quoi pointe le pointeur qui est à l'adresse `0xffffc0dc` ?

Q1.4 Expliquez la valeur apparaissant à l'adresse `0xffffc0e0`.

Q1.5 Où retrouve-t-on les `123A` que j'ai tapé ? Pourquoi cela se retrouve-t-il écrit ainsi ?

Q1.6 Je sais qu'avant d'appeler `f`, le programme a testé un mot de passe, que je cherche à obtenir. Où peut-on voir ce mot de passe dans la pile, que vaut-il ?

Q1.7 Pourquoi ce mot de passe est-il encore dans la pile ?

Q1.8 Comment récupérer ce mot de passe en lançant le programme en-dehors de gdb ? Soyez vraiment précis en expliquant vos calculs, en faisant au besoin quelques hypothèses sur la façon dont se déroule l'exécution.

Q1.9 Pourquoi `bp` pointe encore vers l'adresse `0xffffc108` ?

Q1.10 Quand je relance le programme, j'obtiens exactement le même affichage avec `pframe`. Pourquoi est-ce embêtant du point de vue sécurité ? Quelle solution technique mettre en œuvre pour l'éviter ? Pourquoi elle ne m'empêchera cependant pas de récupérer le mot de passe même sans `gdb` ?

Exercice 2. Lecture d'assembleur

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

```
00000000 <f>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp,%ebp
   3: 8b 45 08              mov    0x8(%ebp),%eax
   6: 8a 10                 mov    (%eax),%dl
   8: 84 d2                 test   %dl,%dl
   a: 74 0b                 je     17 <f+0x17>
   c: 80 fa 31              cmp    $0x31,%dl
   f: 75 03                 jne    14 <f+0x14>
  11: c6 00 32              movb   $0x32,(%eax)
  14: 40                    inc    %eax
  15: eb ef                 jmp    6 <f+0x6>
  17: 5d                    pop    %ebp
  18: c3                    ret
```

Q2.1 Quelle·s instruction·s récupère·nt le·s argument·s de la fonction ?

Q2.2 À quoi servent les instructions aux adresses `0` et `1` ?

Q2.3 Repérez le corps de la boucle, que se passe-t-il pour le registre `%eax` ?

Q2.4 Dans quel·s cas la boucle se termine-t-elle ?

Q2.5 Que fait la fonction, en fait ?

Q2.6 Dessinez l'état de la pile au moment de l'entrée dans la fonction, puis l'état de la pile après l'exécution des instructions aux adresses `0` et `1`.

Q2.7 Le registre `%eax` est numéroté `0` en langage machine x86. Pourquoi cela pose problème ici pour exploiter facilement cette fonction en tant que shell-code ?

Q2.8 Comment pourrait-on la modifier légèrement pour corriger cela ?

Exercice 3. Un problème de taille...

Qualys a rapporté un bug dans Linux [1] concernant un problème de conversion entre les types `size_t` (qui sur architecture 64bit est de taille 64 bits) et `int` (qui est de taille 32 bits) :

```
1  int seq_dentry(struct seq_file *m, struct dentry *dentry, const char *
       esc)
2  {
3          char *buf;
4          size_t size = seq_get_buf(m, &buf);
5
6          if (size) {
7                  char *p = dentry_path(dentry, buf, size);
8  /* [...] */
9  }
10
11 char *dentry_path(struct dentry *dentry, char *buf, int buflen)
12 {
13         char *p = NULL;
14
15         if (d_unlinked(dentry)) {
16                 p = buf + buflen;
17                 if (prepend(&p, &buflen, "//deleted", 10) != 0)
18 /* [...] */
19 }
20
21 static int prepend(char **buffer, int *buflen, const char *str, int
       namelen)
22 {
23         *buflen -= namelen;
24         if (*buflen < 0)
25                 return -ENAMETOOLONG;
26         *buffer -= namelen;
27         memcpy(*buffer, str, namelen);
28 /* [...] */
29 }
```

Dans les conditions de l'attaque, dans la fonction `seq_dentry`, l'appel à `seq_get_buf` stocke dans `buf` l'adresse d'un buffer de taille 2^{31} (alloué dans la mémoire du noyau), et retourne dans `size` cette même taille.

Q3.1 Dans `dentry_path`, combien vaut `buflen` ?

Q3.2 Dans `prepend`, combien vaut `*buflen` après la ligne 23 ?

Q3.3 Pourquoi la vérification échoue à éviter la faille ?

Q3.4 À quel endroit `//deleted` se retrouve écrit, du coup ? Faites un dessin.

Q3.5 Proposez trois manières de corriger le problème.

1. `https://lwn.net/ml/oss-security/20210720123335.GA19170@localhost.localdomain/`

Exercice 4. Le temps, c'est révélant

Voici une implémentation de vérification d'un mot de passe :

```
int checkpass(const unsigned char *given)
{
        const unsigned char good[] = "...";
        int i;

        for (i = 0; good[i]; i++)
                if (given[i] != good[i])
                        return 0;
        return 1;
}
```

Q4.1 Expliquez comment on peut, en observant simplement le temps que nécessite cette vérification de mot de passe, découvrir assez facilement le mot de passe. Donnez une estimation du nombre d'essais nécessaires avec cette implémentation-ci, et comparez-la au nombre nécessaire sans information particulière.

Q4.2 Proposez une autre implémentation, simple, qui corrige le problème grossièrement, i.e. tous les tours de boucle sont effectués.

Q4.3 Améliorez votre implémentation en utilisant des additions/soustractions, pour que le nombre d'opérations (du point de vue C) soit toujours exactement le même (à part un seul test final qui pourra dépendre de la validité du mot de passe).

Q4.4 Expliquez quelles optimisations le compilateur pourrait cependant vouloir faire, pour l'un et pour l'autre de vos implémentations, qui réintroduisent le problème.

Utiliser le qualificateur `volatile` permet d'éviter que le compilateur fasse ce genre d'optimisation, en le forçant à écrire réellement en mémoire les résultats intermédiaires. Lorsque l'on regarde l'assembleur généré, l'exécution prend ainsi bien toujours le même fil et exécute les mêmes instructions.

Cependant... dans le processeur chaque instruction peut avoir un temps d'exécution qui dépend du contenu des données manipulées.

Q4.5 Pour le cas d'une simple addition/soustraction, expliquez pour quelle raison l'instruction pourrait prendre plus de temps.

Q4.6 Expliquez comment votre deuxième implémentation pourrait ainsi être attaquée.

Q4.7 Expliquez pourquoi la durée des opérations bit à bit (and/or/xor) dépendent beaucoup moins des données.

Q4.8 Proposez une troisième implémentation, utilisant ce genre d'opérations (`& | ^`).

Q4.9 Expliquez pourquoi en mesurant la consommation électrique, on pourrait peut-être tout de même détecter le comportement des and/or/xor selon les données manipulées.

Épilogue : ARM et Intel proposent un mode d'exécution pour garantir que certaines instructions durent toujours le même temps : Data-Independent Timing (DIT) ou Data-Operand-Independent Timing (DOIT).

`ret`