PROG L3-MI
DS

2022-10-28

Consignes. *La durée de ce devoir surveillé est d'une heure trente. Aucun document n'est autorisé.* ***Dans tous les exercices, il est possible d'admettre le résultat d'une question pour répondre aux questions suivantes****. Toute réponse doit être justifiée.*

Exercice 1 (environ 3 points)

Q.1 : Quel problème pose l'extrait de code suivant ? Comment pouvez-vous y remédier ?

```
1  double fun_fact(unsigned n)
2  {
3      double acc;
4      for (unsigned long long i = 2; i < n; i++)
5          acc *= i;
6      return acc;
7  }
```

Q.2 : Quel problème pose l'extrait de code suivant ? Comment pouvez-vous y remédier ?

```
1  // affiche le contenu de la zone pointée par p de taille n < 1000, puis la
   ↪  libère
2  void printp(size_t n, int p[n])
3  {
4      for (size_t i = 0; i < n; i++)
5          printf("%d\n", *(p++));
6      free(p);
7  }
8
9  int main()
10 {
11     int *p = malloc(...);
12     ...
13     printp(n, p);
14     ...
15 }
```

Exercice 2 (environ 3 points) : `printf("coucou\n ")`

Un développeur C écrit un programme `mul` prenant deux arguments entiers en ligne de commande et renvoyant leur produit. Il compile ce programme via la commande `$ cc -o mul mul.c`. En le testant il obtient :

```
    $ ./mul 7 191
zsh: segmentation fault  ./mul
```

Afin de localiser le bug, le développeur ajoute une ligne `printf("coucou\n");` dans son programme. En le testant à nouveau, il obtient :

```
    $ ./mul 7 191
coucou
7 * 191 = 1337
```

Q.1 :

1. Expliquez pourquoi l'ajout de la ligne `printf("coucou\n")` ne résout pas le bug ?
2. Formulez un conseil que vous pourriez faire au développeur afin de l'aider à trouver ce bug.

Exercice 3 (environ 3 points) : Canari

Une développeuse C écrit un programme dont un extrait est fourni ci-dessous :

```
void fun257(size_t n, uint16_t p[n], uint16_t q[n])
{
    int t[257];
    int canari = 1337;
    ... // rien qui n'écrit explicitement dans canari
    if (canari != 1337)
        printf("erreur : canari == %d (devrait être 1337)\n", canari);
    ...
}
```

On sait par ailleurs que la fonction `fun257` est l'unique fonction appelée par `main`. Lors de l'exécution, le programme affiche : `erreur : canari == 257 (devrait être 1337)`.

Q.1 :

1. Expliquez *de la façon la plus précise possible* en quoi le message affiché témoigne de la présence d'un bug.
2. Donnez un exemple de scénario simple qui d'après vous pourrait mener à ce bug. Vous pouvez si besoin faire une hypothèse (que vous préciserez) sur la disposition mémoire des variables locales de `fun257`.

Exercice 4 (environ 11 points) : `scrypt`

Le but de cet exercice est d'implémenter une ***version simplifiée*** de la fonction de hachage de mot de passe `scrypt`. On définit celle-ci de la façon suivante (sans pour l'instant préciser le format de représentation des entrées et des sorties) :

Soit H une fonction de hachage prenant en entrée un mot de passe m et une clef s, et n un entier positif (potentiellement grand) inférieur à 2^{64}. On définit :

1. $h_0 = H(s, m)$, puis $h_i = H(s, h_{i-1})$ pour $i \in [\![1, n-1]\!]$.
2. $b_0 = H(s, h_{n-1})$, puis $b_i = H(s, b_{i-1} \oplus h_{b_{i-1} \mod n})$ pour $i \in [\![1, n-1]\!]$ (où '$\oplus$' désigne le OU EXCLUSIF bit à bit, et $a \mod b$ désigne le reste positif de la division de a par b).
3. $\texttt{scrypt}(H, s, m, n) := b_{n-1}$.

Par la suite, on suppose que H est implémentée par *n'importe quelle fonction* de signature et spécification identique à la fonction `hash` de signature :

```
uint64_t hash(const uint8_t s[static 16], size_t m_sz, const uint8_t
↪  m[m_sz]);
```

où :
- `s` contient l'argument s, qui fait toujours 16 octets (128 bits).
- `m_sz` contient la taille de m en octet.
- `m` pointe vers une zone mémoire contenant m, dont la taille fait un nombre entier d'octets.
- La valeur de retour de `hash` correspond au résultat $H(s, m)$, qui fait toujours 64 bits.

On suppose également défini le type `hf_ptr`, correspondant au type d'un pointeur vers une fonction de même signature que `hash` (autrement dit, c'est le type de l'expression `&hash`).

On admet qu'il existe en C un opérateur binaire infixe `^` pour les types entiers qui permet de calculer le OU EXCLUSIF bit à bit. Autrement dit, `a ^ b` pour `a` et `b` de type entier s'évalue à $\texttt{a} \oplus \texttt{b}$.

Enfin, dans tout l'exercice, vous pouvez supposer que les fichiers `.h` appropriés ont déjà été inclus, et vous pouvez utiliser un appel `exit(1)` à la fonction `exit` afin de terminer l'exécution du programme si une erreur a été découverte.

Q.1 : Soit une variable `uint64_t x` :

1. Donnez une expression (fonction de `x`) de type `uint64_t *` et de valeur l'adresse de `x`.
2. Expliquez brièvement pourquoi assigner l'expression ci-dessus vers une variable `y` de type `uint8_t *` et accéder au contenu de `x` via `y[0]`,...,`y[7]` peut possiblement donner des résultats différents en fonction de l'architecture de la machine exécutant le programme.

Dans la suite de l'exercice, on considérera qu'une convention établie en amont permet d'effectuer ce type d'accès sans se soucier d'éventuels problèmes de portabilité.

Q.2 : Écrivez une fonction `scrypt_1` de signature :

```
uint64_t scrypt_1(h_ptr hf, const uint8_t s[static 16], size_t m_sz, const
↪  uint8_t m[m_sz], uint64_t n);
```

et qui calcule $\texttt{scrypt}(\texttt{hf}, \texttt{s}, \texttt{m}, \texttt{n})$. Cette fonction peut allouer dynamiquement de la mémoire, et ne doit calculer *qu'une seule fois* chacune des valeurs h_i.

Q.3 : Écrivez une fonction `scrypt_2` de même signature que `scrypt_1` et qui calcule (toujours) $\texttt{scrypt}(\texttt{hf}, \texttt{s}, \texttt{m}, \texttt{n})$. Cette fonction *ne doit pas* allouer dynamiquement de la mémoire, mais peut calculer *plusieurs fois* chacune des valeurs h_i.

Q.4 : On suppose qu'une lecture ou écriture mémoire (par exemple via une expression `t = h[i]` ou `h[i] = t`) coûte 1000 fois plus cher que le calcul de `hash` pour tous les arguments utilisés en pratique[1]. Dans ce cas, laquelle des deux fonctions `scrypt_1` et `scrypt_2` vous semble la plus efficace quand elle prend `hash` comme premier argument et que :

1. $n = 100$?
2. $n = 10\,000$?

Pour répondre à cette question, vous pourrez vous contenter d'une estimation grossière (néanmoins à justifier) du coût de vos deux fonctions en fonction de n.

Q.5 : Proposez (sans l'implémenter) une approche de *compromis temps-mémoire* permettant de calculer `scrypt` en utilisant une quantité de mémoire allouée dynamiquement spécifiée par un paramètre additionnel $d \in [\![0, n]\!]$ (dont vous êtes libres d'interpréter la signification exacte).

1. Cette supposition est intrinsèquement absurde car telle qu'elle est définie, `hash` doit lire plusieurs de ses arguments depuis la mémoire. On admettra cependant qu'une variante plus sophistiquée de cette supposition peut avoir du sens.