Sécurité des systèmes (OS430)

Examen Session 1 2023-24

Durée totale : 1 heure 30 / Tiers Temps = 2 heures

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Aucun document autorisé.

— Pour la partie 1, plusieurs réponses peuvent être valides à chaque question, on souhaite avoir **toutes** les réponses valides. Chaque question admet au moins une réponse valide et au moins une réponse incorrecte. Le barème fait perdre des points à chaque erreur, et vous pouvez avoir des points négatifs s'il y a trop d'erreurs. Ne pas répondre à une question de QCM entraîne la note 0 à cette question.

— Partie 1, questions brèves : répondre **obligatoirement** dans les parties prévues.

— Les autres parties sont à **rédiger** sur deux copies d'examen classique **DISTINCTES**. Rédiger, c'est à dire justifier vos réponses.

— Il est probable que l'examen soit **trop long**. Nous adapterons le barème. D'ailleurs, le barème n'est donné qu'à titre indicatif (et celui du QCM sera divisé par 2).

Partie 1

1 Questions de cours

Une feuille R/V contenant des questions à choix multiples ainsi que des questions à réponses brèves est **fournie à part**. Elle est à rendre dans votre copie "Partie 2 (LG)".

Partie 2 - LG

2 Vulnérabilité logicielle

En *C* une extension de signe est effectuée lorsqu'une valeur d'un type entier signé est convertie vers un type plus large, et le bit de signe est alors propagé dans les bits inutilisés du bit du type plus large. L'intention de cette extension de signe est celle de la préservation de la valeur lors de la conversion d'un entier signé en un entier signé plus large.

Cette extension de signe peut arriver dans le cas d'une conversion de type (*cast*) implicite ou explicite, par exemple dans une affectation, un appel de fonction ou encore des expressions arithmétiques.

Ainsi, cette extension de signe a quelques avantages, mais peut poser problème lorsqu'elle est appliquée d'un type signé à un type non signé, comme illustré ci-dessous :

```
char len;  // unsigned char
scanf("%d", &len) ; // read the value of len from stdin
strncpy(dst, src, len); // copy len characters from src to dst
```

Si la valeur donnée par l'utilisatrice est négative, alors cette valeur négative est passée en argument à `strncpy()`, dont le troisième argument est de type `size_t` (ie, un entier non signé capable de stocker des valeurs dans l'intervalle $[0, SIZE\_MAX]$). Ce troisième argument devient donc une valeur positive (très) grande.

Question #1 (1 point)

Expliquer pourquoi `len` peut devenir une grande valeur.

Question #2 (2 points)

Donner un exemple de programme où une extension de signe peut mener à un *heap buffer overflow*.

Question #3 (1 point)

Peut-il être utile d'utiliser des *canaries* (`-fstack-protector` option) pour prévenir votre *buffer overflow* ?

Question #4 (2 points)

Plus généralement, quelles protections pouvons nous imaginer pour détecter ce type de problèmes avec l'extension de signe (dans le source, au moment de la compilation, à l'exécution) ?

3 Fuite d'information logicielle

d'après un exercice de L. Mounier

On considère le code suivant, dans lequel la fonction `checKey` est supposée vérifier si la chaîne d'entrée `inputKey` est égale à la valeur secrète `secretKey`. Cette fonction utilise la fontion auxilliaire `equalString` qui réalise une comparaison de chaînes. On suppose par ailleurs que les tableaux/buffers sont alloués correctement, et ils sont de taille `KLENGTH` (donc pas de *buffer overflow*.

```
1  #include<stdio.h>
2  #define KLENGTH 16
3  char secretKey[KLENGTH] ; // valeur secrète
4
5  int equalString(char* s1 , char* s2){
6    int i ;
7    for (i=0; i<KLENGTH; i++) {
8      if (s1 [i] != s2 [i])
9        return 0 ;
10   }
11   return 1 ;
12 }
13
14 void checkKey (char*  inputKey){
15   if (! equalString(inputKey, secretKey))
16     printf("wrong␣key!");
17 }
```

Question #1 (1 point)

Une attaquante réalise une attaque brute force sur le code en essayant tour à tour chacune des possibilités pour la clé. Quel est le coût de cette attaque dans le pire cas (en nombre d'essais) ?

Question #2 (2 points)

Si une attaquante est capable d'exécuter plusieurs fois la fonction `checKey` avec une clé d'entrée incorrecte (non égale à la clé secrète), elle peut gagner de l'information sur la clé secrète en mesurant les temps d'exécutions. Laquelle ? Plus précisément, si la clé est "xxxxxxxxxxxxxxx" et l'entrée "xxxxyyyyyyyyyyyy", que peut-elle apprendre ? En exploitant cette information, quel est le coût d'une telle attaque ?

Pour éviter de telles attaques par temps d'exécution, il est commun de préconiser la règle suivante de programmation, dite *constant time* :

> L'information secrète ne peut être utilisée dans une instruction (en lecture/entrée) que lorsque cette entrée n'a pas d'impact sur les ressources utilisées, ni leur temps d'exécution.

Question #3 (1 point)

Expliquer pourquoi l'implémentation des fonctions `equalString` and `checKey` ne sont pas *constant time*.

Question #4 (2 points)

Proposer une réécriture de la fonction `equalString` utilisant le paradigme *constant time*. Est-ce que la fonction `checKey` est maintenant *constant time* ?

Question #5 (1 point)

Une attaque par *timing* est-elle encore possible ?

Partie 3 (DH) à rendre dans une copie séparée

4 DVFS et canaux cachés

Le DVFS (Dynamic voltage and frequency scaling) est un mécanisme mis en place sur les processeurs modernes afin d'économiser de l'énergie mais également de limiter l'effet du vieillissement sur les composants (parmi les effets du vieillissement, on peut citer la diminution des performances, l'apparition d'erreurs temporaires...). En effet, les fréquences de fonctionnement des processeurs modernes sont très élevées ce qui entraine un échauffement et une consommation plus importante, facteurs d'accélération du vieillissement des circuits intégrés Pour réduire ces phénomènes, les processeurs modernes embarquent un module (le DVFS) qui permet d'adapter la fréquence de fonctionnement des processeurs en fonction de leur charge. Lorsqu'ils ont très peu de charge, la fréquence est diminuée et est réaugmentée à la demande lorsque la charge augmente. Par exemple, un processeur qui fonctionne à une fréquence de 3,2 GHz en temps normal, peut abaisser sa fréquence à 900 MHz en fonction de la charge. La charge n'est pas le seul critère utilisé pour réguler la fréquence, d'autres critères peuvent être utilisés comme la température. Intel a introduit dans ses architectures DVFS un mécanisme qui permet d'augmenter la fréquence du processeur tant que l'on reste en dessous d'un point d'échauffement prédéfini, le TDP pour (Thermal Design Point). En d'autres termes, si la puissance dissipée par le processeur est trop importante alors le système utilise le DVFS pour baisser la fréquence du système et ainsi rester en dessous du TDP. Intel nomme cette fonctionnalité, le Turboboost. Dans le cadre de l'attaque Hertzbleed, les attaquants utilisent ce mécanisme pour attaquer le système et proposent ainsi une nouvelle attaque par canaux cachés qui exploite la mesure de la fréquence de fonctionnement du processeur.

Question #1 (1 point)

Pourquoi observe-t-on que pour des données différentes un programme n'a pas forcément le même temps d'exécution lorsque le Turboboost est activé ?

Question #2 (2 points)

Pourquoi un algorithme codé en temps constant peut être tout de même vulnérable à l'attaque HertzBleed ? Donnez un mode opératoire schématique d'une attaque exploitant le Turboboost.

+1/1/60+

Sécurité des systèmes - Partie 1 (quick)

Consignes.

- Utiliser un **stylo à bille noir ou bleu foncé**.
- QCM : **noircir ou bleuir** la/les cases, sans dépasser ! (Ne pas cocher)
- Questions rapides : rédiger dans les cadres. La partie grisée sert à la notation.

REMPLIR ICI SVP : NOM (MAJUSCULE) et Prénom :

.....

1.1) QCM

Question 1 ♣ (1 point) Lorsqu'une donnée secrète présente en cache est évincée de la mémoire cache, cela ne provoque aucune fuite d'information potentiellement exploitable par un attaquant :

- ☐ faux
- ☐ vrai

Question 2 ♣ (1 point) Pour réaliser une attaque en fautes sur un processeur, il faut forcement avoir un accès physique à celui-ci :

- ☐ vrai
- ☐ faux

Question 3 ♣ (1 point) Pour mettre en place une attaque en fautes, on dissocie les phases d'identification et d'exploitation :

- ☐ La phase d'identification doit être menée dans les conditions opérationnelles du produit attaqué
- ☐ Ces deux phases ne nécessitent pas toujours les mêmes manipulations
- ☐ Ces deux phases sont toujours de difficulté égale
- ☐ Il est parfois possible qu'une attaque facilement réalisable pendant la phase d'identification soit pratiquement très difficile à mettre enplace pendant la phase d'exploitation

Question 4 ♣ (1 point) Laquelle de ces attaques est dite passive :

- ☐ une attaque par microprobing
- ☐ une attaque par canaux cachés
- ☐ une attaque en fautes

Question 5 ♣ (1 point) Parmi ces propositions, lesquelles ne permettront pas d'augmenter la robustesse d'une système à base de microprocesseur (comme par exemple une console de jeux) face aux attaques en fautes :

- ☐ l'utilisation d'une horloge très lente
- ☐ la duplication dde certaines parties du code
- ☐ la réalisation de tous les calculs en temps constant
- ☐ la mise en place de capteur physique empéchant de manipuler la carte électronique
- ☐ l'utilisation d'une cible spécifique telle que un secure element
- ☐ l'utilisation d'une horloge très rapide
- ☐ la mise en place d'un code détecteur d'erreur sur les données manipulées
- ☐ l'obfuscation totale du code

Question 6 ♣ (1 point) Lors d'une attaque CPA sur un AES 128 qui vise un octet de la clef, pour parvenir à trouver la totalité de la clef, il est nécessaire de faire :

- ☐ 256 hypothèses
- ☐ 32768 hypothèses
- ☐ aucune hypothèse
- ☐ 2^{128} hypotheses
- ☐ 128 hypothèses

Question 7 ♣ (1 point) Les attaques par canaux cachés utilisant la mémoire cache permettent de lire directement une valeur secrète en mémoire alors que celle-ci est normalement inaccessible :

- ☐ faux
- ☐ vrai

Question 8 ♣ (1 point) Une contre-mesure visant à ajouter des temps de traitement aléatoires lors d'un calcul cryptographique, a pour effet :

- ☐ d'augmenter le nombre de traces nécessaires à l'attaquant pour que l'attaque converge
- ☐ de rendre l'attaque CPA impossible

Question 9 ♣ (1 point) Un compilateur C suffisamment intelligent :

- ☐ peut prévenir des undefined behaviors
- ☐ peut détecter un buffer overflow à tous les coups
- ☐ rejette les use after-free

Question 10 ♣ (1 point) Une contre-mesure logicielle :

- ☐ doit être résistante aux optimisations du compilateur
- ☐ ne peut contrer que des attaques logicielles
- ☐ doit être réalisée par un compilateur

Question 11 ♣ (1 point) Un langage *memory-safe* :

- ☐ interdit les accès mémoires invalides
- ☐ interdit les accès mémoires simultanés en lecture
- ☐ interdit l'utilisation de données sur le tas

Question 12 ♣ (1 point) Une vulnérabilité de type *use-after-free* :

- ☐ peut rendre corruptible une donnée sensible
- ☐ est un *undefined behavior* en C
- ☐ se résoud en affectant NULL au pointeur
- ☐ peut permettre de faire fuiter de l'information secrète

1.2) Questions à réponses brèves

Question 13 **(1 point)** Quel est le rôle d'un modèle de faute lorsqu'on analyse la robustesse d'un code face aux attaques en fautes ? Quels sont les éléments qui interviennent lors de la définition d'un modèle ?

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

.....

Question 14 **(1 point)** Donner un exemple de vulnérabilité du langage C qu'un langage "type-safe" permettrait d'éviter.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

.....

.....