TP - LIF12 système d'exploitation

Linux

2017/04/25

Introduction

Pour ce tp vous devez utiliser une machine virtuelle préparée spécialement. Vous devez vous connecter sur `http://cloud-info.univ-lyon1.fr`, le serveur cloud du département, et utiliser l'utilisateur `LIF12.chaprot`. Votre encadrant de TP vous fournira le mot de passe.

Créez une machine avec :

— le gabarit `m1.xsmall` ;

— depuis l'image `LIF12-15.04` ;

— sans forcement mettre de clef ssh.

Vous pouvez vous connecter dessus via le login `chaprot` et le même mot de passe que pour vous connecter à la plateforme.

I Débordement de buffer

On parle de débordement de buffer (`buffer overflow`) lorsque les données fournies à un logiciel sont trop nombreuses pour être contenues dans le buffer qui est sensé les contenir. Ce type d'erreur ne cause pas forcément un arrêt du programme, mais la mémoire est alors modifiée par ce que l'utilisateur a fourni sans que le programmeur puisse connaître cette modification. Cela cause un problème de sécurité et souvent, c'est un moyen pour un attaquant de détourner un programme pour acquérir des droits.

Pour ce genre d'attaque, il suffit de connaître un problème de dépassement de capacité sur une commande système. Si cette dernière a des droits particuliers (par exemple le SUID bit), ce qu'on fait faire à cette commande peut être fait avec des droits étendus. Par exemple vous allez ici détourner une commande proche de la commande `su` pour passer administrateur sur l'ordinateur sans son mot de passe.

Sur la machine virtuelle, nous avons installé la commande `/usr/bin/ChangeUser.ex` dont le code est dans le répertoire personnel sur la machine virtuelle, ainsi que sur le site de l'UE. Comme `su`, cette commande permet de changer l'utilisateur courant en fournissant son mot de passe. Elle est mal programmée et permet via un débordement de buffer de changer l'utilisateur sans connaître le mot de passe.

Le code c fourni doit être compilé avec la commande

```
gcc -g -Wall ChangeUser.c --param ssp-buffer-size=200 -fno-stack-protector
                                                      -lcrypt -o ChangeUser.ex
```

Il ne peut pas être réellement fonctionnel sans que son utilisateur ait les droits d'administration. Par contre, on peut étudier son fonctionnement avec un débugger comme kdbg ou gdb.

Q.I.1) - Vérifiez le fonctionnement de la commande ChangeUser.ex :

— tapez ChangeUser.ex autre qui doit permettre d'ouvrir un shell pour l'utilisateur `autre` dont le mot de passe est « autre »[1]. Vous pouvez vérifier que le changement d'utilisateur a réellement lieu grâce à la commande whoami.

1. attention, suite à une mauvaise programmation lors de l'ouverture du shell, certaines variables d'environnement ne changent pas, notamment le répertoire personnel.

— tapez `./ChangeUser.ex autre` en utilisant la commande que vous avez vous même compilée et vérifiez que cela ne fonctionne pas.
— Si le code est identique, pourquoi cela fonctionne-t'il dans un cas et pas dans l'autre ?

Solution: La différence est le SETUID bit et l'utilisateur. Dans la commande installée, le programme peut être lancé par tout le monde mais ses droits sont ceux de l'administrateur. La commande peut donc changer l'utilisateur. Le même code compilé sans les mêmes droits ne le peut pas.

Q.I.2) - En étudiant la commande grâce au débugger, devenez root. Dans la fonction `verif()`, vous pouvez remarquer :
— que l'utilisteur entre un mot de passe et que la lecture (par un simple scanf()) n'est pas sécurisée ;
— que si le mot entré est trop long, ce que l'utilisateur tape va s'écrire sur les variables locales de la fonction `verif()` ;
— que si l'utilisateur parvient à modifier la valeur de la variable `res`, le résultat de cette fonction sera vrai *même si le mot de passe est incorrect.*

Solution: Avec le débugger, on peut voir l'adresse du tableau pass[] et celle de la variable res. Si on écrit un mot de passe suffisamment grand, les données lues vont remplir le tableau pass[] puis toutes les cases mémoire jusqu'à celle de la variable res. Comme res est mis à 0 au début de la fonction et qu'elle n'est pas modifiée, il suffit de lui donner n'importe quelle valeur non nulle pour que le programme la considère comme vraie. Donc il suffit de donner un mot de passe de la bonne taille, contenant n'importe quel caractère et sans espace. Attention, il faut que la taille soit assez précise. Trop long, il va changer l'adresse de la variable de retour de la fonction et le programme va faire une erreur. Pas assez long, la variable res ne va pas être changée.

Sur ubuntu 15.04 avec des machines 64 bits, il faut 80 carractère dans le mot de passe. Un bon moyen pour trouver le bon nombre est de le faire par dichotomie. En effet, l'erreur si le mot de passe est trop court n'est pas la même que celle si le mot de passe est trop long.

I.1 Note à ceux qui souhaitent utiliser leur propre machine

Le code utilisé est très simple et donc très sensible aux contres-mesures mises en place par les compilateurs pour éviter ce genre de problèmes. C'est pourquoi, si vous souhaitez utiliser votre ordinateur personnel, il est probable que cela ne fonctionne pas. Notamment, certains compilateurs (gcc sous ubuntu par exemple) ajoutent des protections lorsqu'on utilise des buffers dans la pile. Il est nécessaire de les désactiver pour obtenir l'effet recherché. De plus, il faut avoir les droits d'administration pour mettre en place la commande ChangeUser.ex.

— Téléchargez le code et compilez le par la commande :

```
gcc -g -Wall ChangeUser.c --param ssp-buffer-size=200 -fno-stack-protector
                                                    -lcrypt -o ChangeUser.ex
```

— Pour qu'il fonctionne vous devez ensuite changer le propriétaire et les droits du fichier exécutable :
  — `su root` (ou sudo...)
  — `chown root:root  ChangeUser.ex`
  — `chmod a+x ChangeUser.ex`
  — `chmod u+s ChangeUser.ex`
  — `logout`
— La deuxieme condition est que l'utilisateur cible ait un mot de passe (par défaut root n'en a pas sous MacOS et Ubuntu).
— Vérifiez le bon fonctionnement du logiciel. La commande suivante devrait vous permettre de passer administrateur en donnant le mot de passe de l'administrateur :

— `ChangeUser.ex root`

Attention, une fois installée, cette commande permet de passer root sans connaître le mot de passe. Pensez à l'effacer !

II Modification de l'ordonanceur

Dans cet exercice, vous allez utiliser explicitement l'ordonnanceur du noyau Linux. C'est assez dangereux car un processus prioritaire qui ne s'arrête pas, par définition, bloque l'ordinateur. Vous devez donc utiliser les machines virtuelles.

Le code `sched.cpp` a été préparé, il lance un certains nombre de threads qui font des calculs en affichant de temps en temps des informations. Pour le moment, le code de la fonction change_ordonnancement() n'est pas fait et l'ordonnancement n'est pas modifié.

Q.II.1) - Complétez le code de cette fonction pour qu'elle change l'ordonnancement du thread qui l'appelle.

Il n'y a pas de fonctions C++ définies dans la librairie standard pour manipuler l'ordonnanceur. Vous allez devoir utiliser les fonctions C suivantes pour faire ce travail :

— pthread_self() qui retourne le numéro du thread qui l'appelle.

— int pthread_getschedparam(pthread_t target_thread, int *politique, struct sched_param *param); qui permet de récupérer les paramètres d'ordonnancement courant. Elle vous permettra d'initialiser les variables.

— int pthread_setschedparam(pthread_t target_thread, int politique, const struct sched_param *param); qui permet de modifier la politique d'ordonnancement et ses paramètres.

La valeur de la politique est définie dans des macros : SHED_FIFO, SHED_RR ou SHED_OTHER.

Faites quelques expériences :

Q.II.2) - Lancez 3 threads RR, l'un de priorité 4 et deux autres de priorité 2.

Q.II.3) - Lancez 3 threads FIFO de priorité identique.

Q.II.4) - Lancez 1 threads FIFO et 2 RR de priorité identique.

Q.II.5) - Lancez 1 thread FIFO de priorité 99 et 2 autres FIFO de priorité plus faibles.

Q.II.6) - Pouvez-vous stopper le programme pendant que des threads très prioritaires tournent ? Pourquoi [2] ?

Solution: Les 4 premières questions ont un comportement qui semble conforme avec ce qui est attendu. La tâche de plus forte priorité gagne, à priorité égale, lorsque qu'une fifo commence elle se termine, les tâches RR s'alternent.

Mais on peut stopper le programme en cours ce qui est par contre anormal (il faut déjà pouvoir le contacter or cela nécessite d'utiliser la gestion du réseau, le programme ssh, le shell ... qui sont des tâches en temps partagé. Si on réfléchit bien, aucun des résultat n'est naturel car pour voir les affichages, il a bien fallu faire du réseau et cela aurait du être bloqué. En fait, on peut s'apercevoir que quoi qu'on dise, les tâches `OTHER` sont parfois exécutées alors qu'il y a des tâches plus importantes en cours.

La raison de tout cela est que pour éviter les « freeze » du système, il y a une sécurité. Un certain pourcentage de temps CPU est réservé aux taches temps partagé. En fait, les tâches temps réelles ne peuvent utiliser que sched_rt_runtime_us/sched_rt_period_us du CPU. On peut avoir un comportement plus conforme en mettant -1 dans le fichier /proc/sys/kernel/sched_rt_runtime_us. Dans ce cas, en refaisant les expérience précédentes, le système est totalement bloqué tant que les processus `RR` ou `FIFO` ne sont pas terminés.

2. Pour vous inspirer, vous pouvez regarder les valeurs des 2 variables du noyau /proc/sys/kernel/sched_rt_runtime_us et /proc/sys/kernel/sched_rt_period_us

Q.II.7) - Si vous avez votre propre ordinateur, refaites les expériences sur ce dernier. Avez-vous les même résultats ? Pourquoi ?

Solution: Il y a une bonne chance qu'il y ait une différence avec les FIFO qui s'alternent par exemple. Cela est du au fait que les ordinateurs sont multi-processeurs. Une tâche n'occupe donc pas tout le processeur.