Année universitaire 2019-2020
Isabelle Puaut

TP SEL

OBJECTIF GLOBAL

L'objectif des travaux pratiques de SEL (Systèmes d'Exploitation, application à Linux) est de développer une application utilisant un ensemble important d'appels système et spécificités de Linux. Le TP consiste à réaliser un outil de modification de code à la volée, par exemple à des fins d'optimisation de code (par exemple pour remplacer dynamiquement le code d'une fonction par une fonction plus rapide à exécuter). Pour rendre le TP réalisable dans le temps imparti, la modification dynamique de code se concentrera sur une seule fonction, et le programme modifié dynamiquement sera composé d'un seul thread. Le programme qui sera modifié dynamiquement sera un de vos programmes exécutables (code binaire d'un TP écrit au préalable en C ou tout langage compilé, table des symboles disponible).

Le travail est divisé en un certain nombre de challenges pour vous aider à progresser vers l'objectif final. Vous avez carte blanche pour organiser votre code en fichiers/fonctions.

CHALLENGE 1 - PRISE DE CONTROLE D'UN PROCESSUS

OBJECTIF

Ce premier challenge consiste à s'attacher à un processus, en utilisant l'appel système *ptrace(PTRACE_ATTACH)* et déclencher un *trap* quand le processus exécute une fonction dont le nom est passé en paramètre (écriture du code de l'instruction machine *trap* à la place de la première instruction de la fonction, de telle sorte qu'elle s'arrête quand elle est exécutée).

RESSOURCES UTILES

- Commande *man* (pages de documentation Linux)
  - Pour obtenir toute la documentation des commandes utilisateur, appels système et fonctions de la *libc*. Les pages de documentation sont organisées en sections (1 = commandes utilisateurs, 2 = appels système, 3=libc). L'option –s vous sera utile (man man pour savoir comment utiliser *man*).
- Contrôle des processus
  - Documentation de l'appel système *ptrace*, en particulier ses paramètres PTRACE_ATTACH, PTRACE_DETACH et PTRACE_CONT
  - Documentation des appels système *waitpid, popen* et *pgrep*

- Gestion des symboles d'un programme
    - Bibliothèque *libelf :* ouverture et parcours d'un binaire ou d'une bibliothèque, notamment de sa table des symboles. Peut être remplacé si vous développez une allergie au format ELF ou à par un parsing de la sortie des commandes *nm* ou *objdump*.
    - Utilitaires *nm* ou *objdump :* sortie texte de la table des symboles d'un programme exécutable (option –t pour *objdump*).
- Visualisation d'informations relatives à un processus en cours d'exécution : répertoire /proc
    - /proc/pid/maps (plan mémoire du processus)
        - Permet de connaître les adresses et droits des segments constituant un exécutable ou une bibliothèque, ainsi que le chemin complet de l'exécutable/bibliothèque.
    - /proc/pid/mem (le contenu de mémoire d'un processus)
        - Utilisé pour lire et écrire dans la mémoire d'un autre processus. */proc/mem* est accessible comme un fichier standard, via les fonctions d'accès aux fichiers (*fopen/fseek/fread/fwrite/fclose* ou équivalent sous la forme d'appel système *open/seek/read/write/close*)
        - **Note** : pour écrire, le processus dont l'espace d'adressage doit être modifié doit être attaché via *ptrace*. L'écriture dans les fichiers spéciaux n'est effective qu'après fermeture du fichier.
        - **Note** 2 : on préférera écrire dans /proc/pid/mem que d'utiliser le paramètre POKE_DATA de *ptrace*, qui vous force à avoir conscience de *l'endianess* de l'architecture.
    - /proc/pid/cmdline
        - Commande qui a été lancée (permet de récupérer le nom du programme en train de s'exécuter à partir de son pid)
- Informations diverses :
    - Le code de l'instruction trap : 0xCC (sur un octet)
    - *atoi/atol/strtol* pour récupérer les arguments de la fonction *main*
    - *sprintf/sscanf/sprintf* pour la manipulation de chaines
    - *basename :* extraction de la base d'un chemin absolu ou relatif

POUR ETRE CAPABLE DE DEMARRER SUR VOTRE MACHINE PERSONNELLE

Les versions récentes de Linux permettent de contrôler qui a le droit d'appeler un *ptrace* sur un autre processus. Le contenu du fichier */proc/sys/kernel/yama/ptrace_scope* peut contenir une des valeurs suivantes :

- 0 : classical ptrace (can trace own processes)
- 1 : restricted ptrace (can trace as root only)

Si on est dans le second cas, il faut exécuter la commande suivante :

- echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope

POUR Y ALLER PROGRESSIVEMENT

- Commencer en passant l'*adresse* de la fonction (obtenue en appelant *nm* ou *objdump*) à la place de son nom.

CHALLENGE 2 – EXECUTION DE CODE DANS UN PROCESSUS DIFFERENT

OBJECTIF

Ce second challenge consistera à faire exécuter par le traçant une fonction *foo* située dans l'espace d'adressage du processus tracé, et de vérifier que la fonction s'est exécutée normalement. Deux étapes pourront être distinguées dans ce challenge :

- Appel d'une fonction avec le mode de paramètre par *valeur* (ex : int foo(int i)).
- Appel d'une fonction avec un passage de paramètre par *référence* (ex : int foo(int *i)), pour retourner une valeur en passant en paramètre un pointeur sur cette dernière.

PRINCIPE DE REALISATION

Comme on veut appeler une fonction qui se trouve dans un processus séparé, on ne peut bien entendu ni appeler la fonction de manière standard, ni via un pointeur de fonction. Le principe de réalisation sera donc le suivant :

1. Faire en sorte de stopper l'exécution du processus tracé (challenge 1).
2. Récupérer la valeur des registres du processus
3. Modifier le code du processus tracé pour insérer l'appel à la fonction à appeler. On ne peut pas se contenter ici d'initialiser *rip* à l'adresse de *foo* car on aurait alors un problème lors du retour de fonction. On pourra utiliser par exemple un appel indirect via le registre *rax* (code 0xff 0xd0), suivi d'un trap pour récupérer le contrôle.
4. Initialiser les registres pour les paramètres de l'appel
5. Redémarrage du processus, qui exécutera de ce fait le code de la fonction
6. Test du code de retour, restauration du code et de la valeur initiale des registres

Le plus simple (mais pas la seule solution) pour effectuer ce challenge est de sélectionner une fonction bar que l'on sait exister et être exécutée de manière répétitive dans le processus tracé. Le point 1 met un *trap* à la place de la première instruction de la fonction *bar*, et le code d'appel à *foo* sera placé en début de la fonction bar.

RESSOURCES UTILES

- Documentation de l'appel système *ptrace*, paramètres PTRACE_GETREGS et PTRACE_SETREGS
- Conventions de passage de paramètre x86 par registres (cf annexe)

CHALLENGE 3 – CREATION ET REMPLISSAGE D'UN CODE CACHE

OBJECTIF

Ce second challenge consiste en la création et le remplissage d'un « code cache ». Un code cache, dont la taille sera donnée en paramètre, est une zone de mémoire allouée dynamiquement dans l'espace d'adressage du processus contrôlé, qui contiendra un code exécutable (par exemple le code optimisé d'une fonction que l'on veut appeler à la place de la fonction originale). Pour que votre code reste simple, le code cache mis en place ne contiendra qu'une fonction.

PRINCIPE DE REALISATION

- Appel à *posix_memalign* (allocation avec alignement, ici sur une frontière de page) *par le processus contrôlé*, en utilisant le code écrit au challenge 2.
- Appel à *mprotect* pour permettre l'écriture et l'exécution du « code cache », en suivant le même principe que pour l'appel à *posix_memalign*
- Recopie du code dans la zone ainsi réservée

RESSOURCES UTILES

- *posix_memalign* : pour l'allocation mémoire avec alignement
- *getpagesize* : récupération de la taille d'une page
- *mprotect* pour rendre la zone modifiable et exécutable
- *Gcc* et *objdump* (-d) pour compiler la fonction contenant le code alternatif et récupérer son code en hexa. Option –T de *objdump* pour avoir les offsets des symboles de la *libc*
- Remarque : une première approche pour ne pas gérer la *libc* qui est liée dynamiquement est dans un premier temps de la lier en statique avec le programme tracé

TEST

Comment tester que ça fonctionne correctement ? A ce stade, On peut juste s'assurer que les appels système utilisés se déroulent correctement en récupérant leur code de retour. On ne peut pas en revanche à ce point tester que le code inséré dans le code cache s'exécute correctement, étant donné qu'il n'est pas encore appelé.

CHALLENGE 4 – OPTIMISATION A LA VOLEE

Ce quatrième challenge consiste à remplacer le code de la fonction par une fonction alternative. Deux options de réalisation sont disponibles :

- Trampoline (plus facile à réaliser mais plus lent à l'exécution) : le *début* du code de la fonction à remplacer est modifié pour y insérer un saut (*jump*) à la fonction insérée dans le « code cache ».
- Replacement des appels : les *appels* à la fonction d'origine sont remplacés par des appels à la nouvelle fonction. Cette option sera réalisée par insertion d'un *trap* dans la fonction d'origine, pour repérer l'appel et le changer par l'appel à la nouvelle fonction.

On prévoira un moyen de revenir à la fonction de départ, par exemple si on se rend compte que l'optimisation effectuée n'est pas adaptée (code plus lent, fonction pas appelée assez souvent).

RESSOURCES UTILES

- Code du jump absolu : 0x48 0xb8 suivi sur 8 octets de l'adresse à laquelle on veut se brancher et ff e0 (chargement de l'adresse où l'on veut brancher dans *rax* suivi d'un branchement indirect à *rax*).
- *sigaction* : pour associer une fonction qui sera lancée à la réception du signal SIGCHLD (processus stoppé ou terminé). Cette fonction ne sera utile que dans l'option de réalisation « remplacement des appels ».

CHALLENGE 5 – BONUS MULTITHREAD POUR LES BRAVES

Ce challenge, consiste à étendre votre code de manière à modifier dynamiquement du code *multi-thread*. Un cas d'étude intéressant sera de remplacer dynamiquement le code de la fonction *increment* suivante par un code plus rapide utilisant le préfixe *lock* et l'instruction x86 inc pour que le code soit toujours correct mais s'exécute beaucoup plus vite. On évaluera le gain en performance obtenu.

```
int val ; // variable globale partagée par tous les threads
void increment(void) {
   pthread_mutex_lock(&mutex);
   val++;
   pthread_mutex_unlock(&mutex);
}
```

Le principal écueil que vous devrez surmonter sera que quand vous arrêtez un programme multithread, il faudra gérer le fait que vous ne contrôlez pas à quel point seront arrivés les threads au moment de la prise de contrôle.

ELEMENTS D'EVALUATION

- Progression dans les challenges : challenge 4 avec Trampoline montre que votre contrat est pleinement rempli !
- Robustesse de votre code aux erreurs
- Structuration et facilité à comprendre votre code
- Aptitude à expliquer votre code, évalué individuellement dans un binôme
- Tout plagiat total ou partiel de code sera passible de sanctions disciplinaires via la commission disciplinaire de Rennes 1
- Bonus pour les braves : reprendre le TP pour des programmes multi-threads. Comme exemple de test, on pourra remplacer à la volée une fonction d'incrémentation d'une variable utilisant des mutex Posix par l'équivalent utilisant le préfixe LOCK.

UN PEU DE DOCUMENTATION

CONTENU DE /PROC/PID/MAPS

Zone d'adresse, protection, chemin d'accès en fin de ligne. Le code (du processus et de la libc) sont identifiés par les droits rx.

```
00400000-00401000 r-xp 00000000 00:1b 4857                /media/sf_PartageVirtualBox/tpsel/toto
00600000-00601000 r--p 00000000 00:1b 4857                /media/sf_PartageVirtualBox/tpsel/toto
00601000-00602000 rw-p 00001000 00:1b 4857                /media/sf_PartageVirtualBox/tpsel/toto
7fc5bdc79000-7fc5bde33000 r-xp 00000000 08:01 63085       /lib/x86_64-linux-gnu/libc-2.19.so
7fc5bde33000-7fc5be033000 ---p 001ba000 08:01 63085        /lib/x86_64-linux-gnu/libc-2.19.so
7fc5be033000-7fc5be037000 r--p 001ba000 08:01 63085       /lib/x86_64-linux-gnu/libc-2.19.so
7fc5be037000-7fc5be039000 rw-p 001be000 08:01 63085       /lib/x86_64-linux-gnu/libc-2.19.so
7fc5be039000-7fc5be03e000 rw-p 00000000 00:00 0
7fc5be03e000-7fc5be061000 r-xp 00000000 08:01 63071       /lib/x86_64-linux-gnu/ld-2.19.so
7fc5be243000-7fc5be246000 rw-p 00000000 00:00 0
7fc5be25d000-7fc5be260000 rw-p 00000000 00:00 0
7fc5be260000-7fc5be261000 r--p 00022000 08:01 63071       /lib/x86_64-linux-gnu/ld-2.19.so
7fc5be261000-7fc5be262000 rw-p 00023000 08:01 63071       /lib/x86_64-linux-gnu/ld-2.19.so
7fc5be262000-7fc5be263000 rw-p 00000000 00:00 0
7ffe62248000-7ffe6226a000 rw-p 00000000 00:00 0           [stack]
7ffe62379000-7ffe6237b000 r--p 00000000 00:00 0           [vvar]
7ffe6237b000-7ffe6237d000 r-xp 00000000 00:00 0           [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0   [vsyscall]
```

REGISTRES DES X86 (EN 64 BITS) ET CONVENTION DE PASSAGE DE PARAMETRES

- Registre *rip* : compteur de programme (instruction pointer)
- Convention de passage de paramètres lors d'appels système : utilisation des registres *rdi, rsi, rdx* (dans cet ordre pour les paramètres 1, 2 et 3)
- Valeur de retour retournée dans le registre *rax*
- Remarques
    - En 32-bits, les paramètres sont passés dans la pile et pas par registres
    - Utiliser l'option –no-pie de *gcc* dans le cas où votre compilateur/noyau est généré en position-Independent