SEL
Rapport de TP
ISTIC - Université de Rennes 1

Luis Thomas
luis.thomas2005@gmail.com

Malo Poles
poles.malo@gmail.com

Vendredi 7 Décembre 2018

1 Introduction

Ce document témoigne du travail fourni afin de remplir les objectifs de ce TP, commençons par énumérer ces derniers.

Objectifs : L'objectif principal était de remplacer dynamiquement l'exécution d'une fonction donné dans un processus, par une autre suite d'instructions. Il fallait pour cela réussir à arrenter le processus, puis allouer de la mémoire afin de copier les nouvelles instructions, puis enfin obliger le processus a exécuter ces instructions plutôt que celle de la fonction donné en utilisant un trampoline.

Le code source est disponible sur GitLab a cette adresse : `https://gitlab.com/Luisky/sel_tp`

2 Travail Accompli

Afin de réaliser ces objectifs nous avons utilisé la fonction ptrace() de la libc, utilisant elle même l'appel système sys_ptrace (101 sur x86_64).

Cette fonction permet a un processus d'interagir avec un autre processus, en modifiant par exemple l'état de sa mémoire ou de ses registres.

Ptrace permet a un processus de se laisser tracer par un autre processus (PTRACE_TRACEME), nous n'avons pas utilisé cette fonctionnalité car il s'agissait de modifier la mémoire d'un processus sans son accord préalable. La fonctionnalité que nous avons utilisé est PTRACE_ATTACH,

nous utilisons pgrep() afin de récupérer le pid du processus tracé (que nous appellerons simplement 'tracé' dans la suite de ce document), et nous nous attachons.

Challenge 1 : Le premier challenge visait simplement a stopper le tracé sur une instruction nommé int3 (opcode 0xCC), cette dernière génère une interruption logicielle, le noyau prend en charge cette interruption en envoyé SIGTRAP au processus ayant déclenché cette interruption logicielle. Un processus recevant SIGTRAP s'arrête et son comportement standard est de créer une image de sa mémoire a l'état de son arrêt.

Notre but était donc d'aller écrire dans la mémoire du tracé cette instruction a la place de la première instruction de la fonction donné (que nous appellerons 'func').

Il nous fallait donc plusieurs choses:

- l'adresse de func (que nous pouvons obtenir avec nm)
- l'adresse du code du tracé en mémoire (en lisant /proc/pid/maps)
- que le tracé soit a l'arrêt (on ne peut écrire dans /proc/pid/mem sinon, voir man proc)

La troisième condition est remplie dès lors que l'on s'attache au tracé avec PTRACE_ATTACH, en effet ce dernier envoi SIGSTOP au tracé, le traçant effectue un waitpid() et lorsqu'il revient on vérfie qu'il s'agit bien du signal 19 (voir kill -l).

Pour obtenir l'adresse de func nous utilisons nm avec grep afin de récupérer l'offset par rapport au début du fichier ELF. Ensuite en 'parsant' le fichier maps du tracé on récupère l'adresse du code en mémoire. En additionnant les deux on obtient l'adresse de func en mémoire. Il suffit ensuite d'effectuer une routine faite de open(), lseek(), read(), lseek(), write() et close() afin d'ouvrir le fichier mem, de se placer a l'adresse de la fonction, de sauvegarder l'octet que l'on s'apprête a écraser, puis d'écrire 0xCC et enfin de fermer le fichier.

Une fois ces opérations effectuées on peut relancer le tracé avec PTRACE_CONT ceci entraîne un arrêt de ce dernier, en ayant configuré avec la commande ulimit -c unlimited l'obtention d'un core dump on peut ensuite l'analyser pour découvrir qu'il y a bien 0xCC écrit a l'adresse de func. (objdump -d core | grep [l'addresse de func en mem])

Challenge 2 : Une fois le challenge 1 réussi nous remplaçons l'écriture de 0xCC par 0xCC 0xFF 0xD0 0xCC, 0xFF 0xD0 est un opcode sur 2 bytes, il permet de faire un saut a l'adresse contenue dans le registre rax. En utilisant PTRACE_GETREGS et PTRACE_SETREGS on peut manipuler les registres du processeur.

Challenge 3 : Ce challenge consiste a allouer de la mémoire et la rendre exécutable, les fonctions proposées sont posix_memalign() et mprotect(), pour utiliser posix_memalign() il nous faut de la mémoire avec des droits en lecture et écriture (rw-p). Pour cela nous pouvons utiliser le Tas (Heap) ou la Pile (Stack), dans le premier cas il faudra sauvegarder l'ancienne valeur dans le tas, dans le second il suffira d'allouer de la place dans la pile (rsp = rsp - 8), nous avons utilisé PTRACE_PEEKDATA et PTRACE_POKEDATA pour ces opérations.

Il est a noter que la page man posix_memalign() donne une autre fonction standard en C11: aligned_alloc() et qui ne nécessite pas de mémoire, l'adresse alloué est renvoyé dans rax. Il existe aussi une fonction mmap() qui permet d'allouer de la mémoire et de lui donner des droits en un seul appel. Et puis quitte a optimiser la chose, il existe un appel système mmap (numéro 9). Cela nécessite de remplace 0xFF 0xD0 par syscall (0x0F 0x05). Néanmoins la procédure ne change pas, on place dans les bons registres les différents paramètres après lecture du manuel.

Une fois la mémoire alloué il faut aller chercher l'adresse et la taille de la fonction que nous allons copier en mémoire. Puis copier ces instructions dans la mémoire alloué.

Une fois tout cela effectué il suffit de laisser le tracé s'arrêter sur le dernier 0xCC et d'aller analyser le core. Après vérification les instructions sont bien copiés.

Challenge 4 : Il s'agit maintenant de remplacer le début de func par un trampoline, le code se compose d'un mov de l'adresse du code cache vers rax, puis d'un saut a l'addresse de rax suivit d'un ret.

Après avoir écrit ce trampoline il ne faut pas oublier de mettre rip a l'adresse de func, puis il suffit de relancer le tracé.

Notre tracé incrémentait une variable passé en paramètre en effectuant un sleep(1), le code cache n'effectue pas de sleep().

Notre tracé s'emballe et affiche frénétiquement des valeurs. Après 3 seconde nous décidons de restaurer le tracé dans son état initial, en remplaçant le trampoline par son code original. et en utilisant l'appel système unmap (numéro 11) afin de ne laisser aucune trace de notre passage.

Le tracé reprend son comportement original, et en observant le fichier /proc/pid/maps la mémoire alloué n'est plus. Mission complete.

Challenge 5 : What ? Something Else ?
Le but de ce challenge est d'échanger un pthread_mutex_lock() avec quelque chose de plus léger. Nous avons repris le fonctionnement du programme au Challenge 4 en ajoutant une fonctionnalité permettant de récupérer les TID. Une fois récupérés nous nous attachons a chacun des threads (ce qui a pour effet de les stopper). Puis nous écrivons 0xCC au début de la fonction d'incrémentation

3 Résultats

D'après l'énoncé le contrat est rempli. Nous nous sommes permis d'optimiser l'allocation du code cache afin de raccourcir la longueur du code. Le code source contient encore des fonctions pour récupérer les offset dans la libc ainsi que de l'adresse dans maps de la libc. Des tests ont été nécessaires pour le challenge 5 notamment pour trouver les bons opcode a utiliser, ils nous a fallu créer un programme en assembleur (Syntax AT&T) pour nous en sortir. Le challenge 5 a été réussi ce vendredi 7 décembre 2018.

4 Conclusions

C'était un TP intéressant.