TP de Systèmes d'exploitation
François Cayre / francois.cayre@grenoble-inp.fr

Gestionnaire de mémoire virtuelle

Notions abordées : pages de mémoire, droits associés, fautes de segmentation.

Fonctions utilisées : `mmap(2)`, `munmap(2)`, `mprotect(2)`.

1 Retour sur l'allocation dynamique

Bien qu'un appel explicite (`malloc(3)`, `calloc(3)`, `realloc(3)`, `posix_memalign(3)`) à l'allocateur de mémoire soit réputé coûteux, différentes stratégies sont mises en œuvre par le système d'exploitation pour retourner rapidement un bloc de mémoire (situé sur le tas) au processus appelant.

Par exemple, l'allocateur va pré-allouer des blocs de mémoire de tailles réputées être souvent utilisées[1]. Puisque le système va privilégier la performance (retourner *rapidement* un bloc de mémoire), il peut choisir de retourner un bloc (bien) plus grand que demandé par le processus appelant. Techniquement, l'allocateur mémoire repose sur le gestionnaire de mémoire virtuelle du système d'exploitation, sur lequel il vient greffer ses propres méta-données pour gérer des blocs qui ne sont pas nécessairement des pages (on rappelle que le gestionnaire de mémoire virtuelle ne sait gérer que des pages). Généralement, ce sont ces méta-données que l'on vient corrompre en écrivant par erreur *sous* (*underflow*) le bloc renvoyé par `malloc(3)`, en obtenant un message du type `double free or corruption`. Il est également possible de corrompre ces méta-données en écrivant au-dessus (*overflow*) du bloc mémoire alloué.

Une faute de segmentation résulte d'un accès illégal à la mémoire, hors d'un bloc *réellement* alloué sur le tas par exemple. Un signal `SIGSEGV` est alors envoyé au processus ayant commis l'accès illégal. L'action par défaut associée à la réception de `SIGSEGV` est la terminaison anormale du processus fautif (et l'affichage du fameux message `Segmentation fault`).

Ainsi, il est possible, parce que `malloc(3)` a effectivement renvoyé un bloc plus grand que demandé, que des accès illégaux passent inaperçus pendant la phase de mise au point d'un programme par les développeurs (si on ne sort pas du bloc *réellement* alloué). Et donc qu'un programme qui semblait fonctionner pendant la phase de mise au point, se mette à planter une fois déployé en production (parce que le bloc fera alors *réellement* la taille demandée). Illustration classique de la Loi de Murphy…

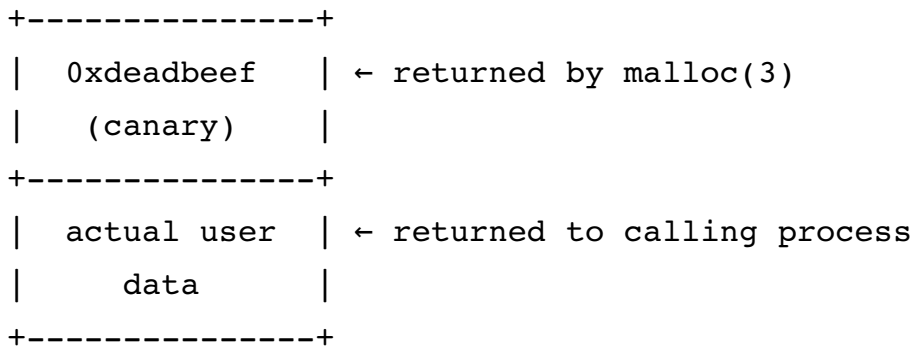
Au fil des années, plusieurs outils ont été créés pour assister le développeur lors de la phase de mise au point d'un programme, pour répondre spécifiquement à

1 Dans la vraie vie du dehors, les structures dynamiques que vous connaissez (listes, piles, files, arbres, etc.) utilisent souvent un pointeur générique pour indiquer où trouver le contenu de la cellule (de la liste) ou du nœud (de l'arbre). Ainsi, il ne paraît pas absurde, par exemple, de pré-allouer des blocs de taille `2*sizeof(void*)` et `3*sizeof(void*)`.

ce problème[2]. À l'heure actuelle, l'outil de choix s'appelle `valgrind` (valgrind.org), et nous ne saurions trop appuyer sur son utilité de premier ordre. Avant lui, les développeurs utilisaient soit la librairie ElectricFence, soit leurs propres techniques, dont nous donnons les grandes lignes dans ce TP. Une fois certains de la correction de leur code, les développeurs reviennent alors au `malloc(3)` original pour la version de production.

2 Un canari sous une clôture électrifiée

Les mineurs, pour détecter les émanations de grisou, travaillaient en compagnie d'un canari en cage dans la veine qu'ils étaient en train d'exploiter [1]. L'oiseau, plus sensible que les mineurs, s'arrêtait de chanter lorsqu'il sentait le gaz arriver, ayant ainsi permis de sauver de nombreuses vies. C'est le nom que porte encore aujourd'hui une technique de protection de la mémoire en informatique (*canary*). Cette technique permet de protéger une extrémité d'un bloc mémoire en détectant qu'une *écriture* a été faite au-delà de cette extrémité. En informatique, placer un canari consiste à écrire un motif binaire connu juste avant/après le bloc mémoire effectivement donné au processus appelant. Si le processus, par erreur donc, venait à écrire au-delà du bloc qu'il a reçu, il modifierait ainsi très probablement le canari, et l'on pourrait s'en rendre compte, par exemple au moment de rendre le bloc au système.



Il existe plusieurs variations sur le thème du canari (XOR, aléatoire, *etc.*) et nous utiliserons la plus simple. Un canari, s'il permet de protéger des blocs de taille arbitraire, ne permet pas de détecter l'instruction exacte qui a causé l'accès illégal en écriture (un canari ne pouvant donc absolument pas détecter aucun accès illégal en lecture).

Une autre technique pour protéger l'extrémité d'un bloc mémoire consiste à utiliser le système de droits associés à une page de mémoire par le gestionnaire de mémoire virtuelle. On peut, à l'aide de l'appel système `mprotect(2)`, faire en sorte d'interdire l'accès (en lecture/écriture) à une ou plusieurs *pages* mémoire. On parle ici, toujours par analogie, de clôture électrifiée. Cette technique ne s'appliquant qu'à des pages entières, on ne peut pas l'utiliser pour protéger les deux extrémités d'un bloc mémoire de taille arbitraire. La librairie ElectricFence utilisait cette technique, et nécessitait donc de tester le programme deux fois :

2 Pour les accès illégaux à la pile (et non plus au tas), il a fallu que les compilateurs implantent une fonctionnalité dite de protection contre l'écrasement de pile (*stack smashing protection*).

une première fois en protégeant les accès hors de l'extrémité inférieure, puis une seconde fois en protégeant les accès hors de l'extrémité supérieure des blocs renvoyés au processus appelant. Malgré cet inconvénient, une clôture électrifiée permet de détecter les accès illégaux en écriture *et* en lecture, tout en faisant planter le processus appelant *exactement* sur l'instruction fautive.

3 Travail à réaliser

Ce qui suit suppose que vous vous trouviez dans le répertoire du code fourni avec ce TP. Vous êtes invités à suivre ces explications avec le fichier `main.c`.

3.1 Quelques accès illégaux avec `malloc(3)`

Pour observer (potentiellement !) une corruption des structure internes de l'allocateur mémoire :

```
$ make clean && make ALLOC=MALLOC TEST=UNDERFLOW && ./palloc-test
string @ 0x92e010
*** Error in `./palloc-test': double free or corruption (out): 0x000000000092e010 ***
======= Backtrace: =========
…
$
```

Pour observer (très probablement) un accès illégal qui passe pourtant inaperçu, vérifiez que l'exemple suivant ne cause malheureusement aucun problème particulier :

```
$ make clean && make ALLOC=MALLOC TEST=OVERFLOW && ./palloc-test
string @ 0xb51010
$
```

3.2 Stratégie à implanter

Nous allons implanter nos propres versions de `malloc(3)`, `calloc(3)` et `free(3)`, appelées respectivement `pmalloc`, `pcalloc` et `pfree`[3], permettant une gestion relativement protégée des blocs alloués sur le tas.

Vu les contraintes des deux techniques décrites précédemment, nous choisissons de protéger l'extrémité inférieure des blocs par un canari, et l'extrémité supérieure des blocs par une clôture électrifiée. Voici déjà un aperçu du résultat final de ce TP. Détecter une écriture lors d'un débordement par le bas (canari) :

```
$ make clean && make ALLOC=PALLOC TEST=UNDERFLOW && ./palloc-test
string @ 0x7f7c669a6ff6
* PALLOC [WARNING] Detected write access below pointer 0x7f7c669a6ff6!
$
```

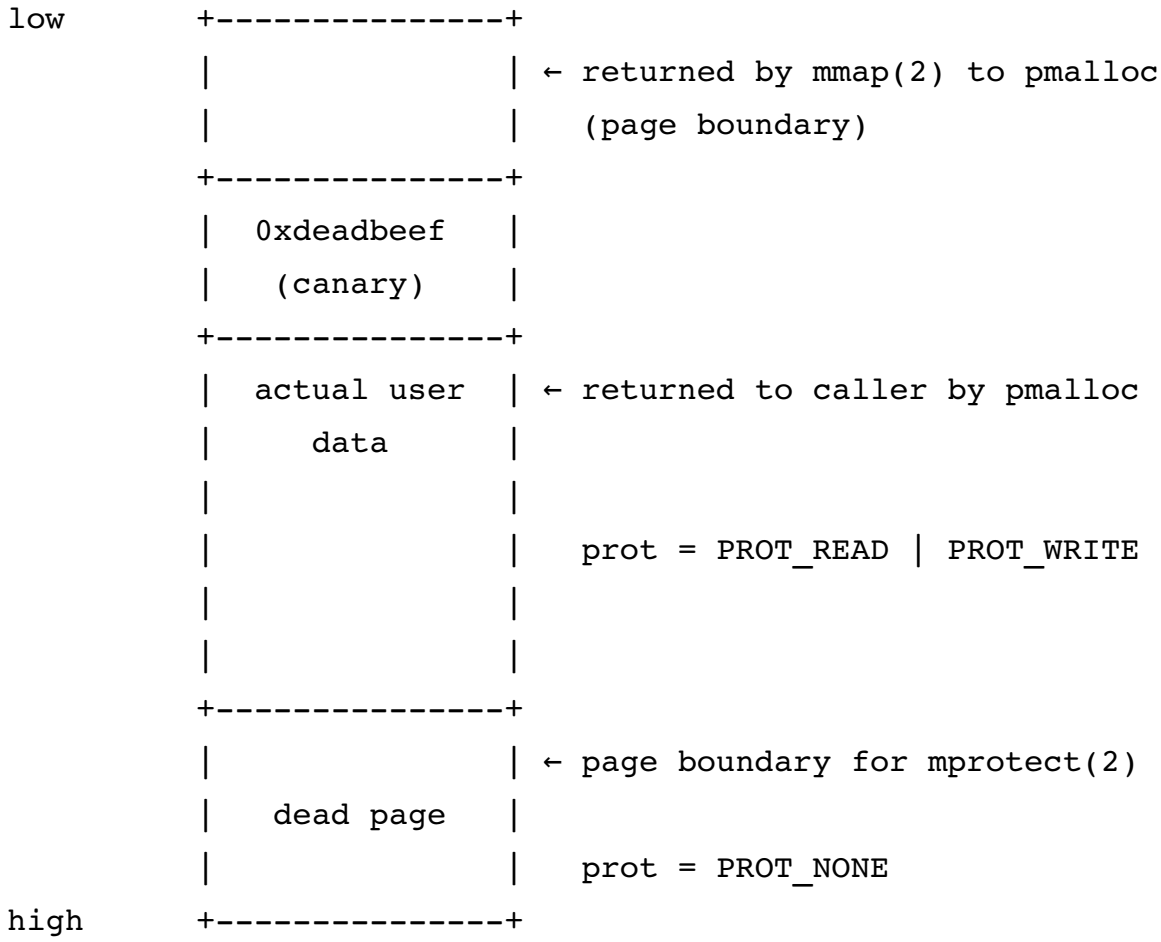
3 Dans un souci d'exhaustivité, il conviendrait aussi de traiter d'autres fonctions utilisant `malloc(3)` en interne, comme par exemple `strdup(3)`.

Puis, pour un débordement par le haut, c'est la clôture électrifiée qui agit :

```
$ make clean && make ALLOC=PALLOC TEST=OVERFLOW && ./palloc-test
string @ 0x7f45b840eff6
Segmentation fault (core dumped)
$
```

Notez que le canari ne permet que d'émettre un avertissement au moment de libérer le bloc (ce sera à `pfree` d'émettre le message), alors que la clôture électrifiée permet de causer l'émission d'un signal `SIGSEGV` *dès* que l'on écrit après le dernier octet du bloc, ce qui se révèle bien utile dans un débogueur !

Vous devrez donc vous débrouiller pour que vos fonctions `pmalloc` et `pfree` dans `palloc.c` interagissent autour de l'implantation mémoire suivante :



Si vous vous ennuyez, réfléchissez au moyen d'ajouter une option au code pour inverser les protections (canari en haut et clôture électrifiée en bas) !

4 Références

[1] Ouest-France, Édition de St-Brieuc, 28/11/2013 (lien).

Bon travail !