SDE : TP Mémoire

1 Préliminaires

Téléchargez l'archive sur Moodle, puis tapez `tar -xvf TPMem.tar.gz`. Vous allez travailler dans le répertoire `TPMem` créé. Ce répertoire comprend un certain nombre de programmes ainsi qu'un Makefile. Répondez soigneusement aux questions sur une feuille.

2 Aspects matériels

L'unité centrale (CPU/processeur) dispose de registres (zones de mémoire contenant un mot binaire *machine*, généralement de 32 ou 64 bits) et communique via un bus (media de communication par diffusion : tous les communicants reçoivent les données qui circulent) avec la mémoire vive et les divers périphériques (écran, clavier, souris, disque dur, etc.)

Dans une machine cohabitent diverses technologies de mémoire ayant des coûts et des vitesses d'accès différentes.

* mémoire vive, non persistente (par ordre décroissant de coût et de vitesse) :
  1. registre, utilisation temporaire
  2. SRAM, mémoire tampon/cache/antémémoire, proche de l'unité centrale
  3. DRAM, mémoire centrale
* mémoire persistente :
  - ROM, en lecture seule, mémoire de démarrage
  - swap, mémoire d'échange/auxiliaire sur disque dur. C'est une technique logicielle qui donne à l'utilisateur l'apparence d'une capacité de mémoire vive plus grande en utilisant une partie du disque dur.

On peut constater un *principe de localité* dans l'utilisation de la mémoire : des accès successifs à des adresses proches sont plus fréquents que des accès successifs à des adresses éloignées. On a donc intérêt à charger une copie de toute une suite de données dans une mémoire tampon plus rapide, pour accélérer les accès futurs. Le recours à la mémoire tampon est maintenant systématique et hiérarchisé en niveaux.

— Visualisez le fichier `/proc/cpuinfo`[1]. Combien y a-t-il de processeurs sur votre machine ? D'après vous, où sont stockées ces informations ?

— Quel est le modèle des processeurs ? Cherchez sur internet quels sont les niveaux de cache pour ce modèle.

— D'après `/proc/cpuinfo`, quelles sont les tailles des adresses physiques et virtuelles ?

1. Sur les systèmes du type Unix, procfs (process file system) est un pseudo-système de fichiers utilisé pour accéder aux informations du noyau sur les processus. Il est habituellement monté au démarrage sur le répertoire /proc. Ce répertoire ne consomme aucun espace disque mais seulement une quantité limitée de mémoire vive (un fichier non vide a une taille affichée de 0 avec ls).

— Sur le principe du *mapping mémoire*, les périphériques d'entrée/sortie sont vus comme des adresses mémoires. Pour envoyer une donnée vers un matériel donné, le processeur va écrire à une adresse particulière. Le fichier `/proc/iomem` fournit cette correspondance. A votre avis, s'agit-il d'adresses physiques ou d'adresses virtuelles ?
— Evaluez le nombre d'adresses disponibles au titre de `system RAM`. Vous pouvez utiliser `gdb` pour calculer en hexa. Par exemple, dans l'invite de commandes gdb, `print/x 0xf - 0x3` renverra le résultat de la soustraction en hexa (0xc), tandis que `print/d` renverra le résultat en décimal (12). Vous voyez que les nombres préfixés par 0x sont à interpréter en hexa.
— Sachant qu'une adresse identifie un octet, comparez avec la ligne `MemTotal` du fichier `/proc/meminfo`.
— Quelle est la taille de la mémoire d'échange (`SwapTotal`) ? A quoi sert cet espace mémoire ?

3 Codage en mémoire

La mémoire est organisée comme un tableau de mot binaire *mémoire* (de 8 bits, soit un octet). Chacun est identifié par son indice appelé adresse mémoire. Les adresses sont codées sur un mot binaire *machine* (de 32 ou 64 bits).

— Lisez le programme `boutisme.c`. Que fait-il ?
— Compilez (`make boutisme`) et exécutez ce programme. L'octet de poids faible se trouve-t-il à l'adresse la plus petite ou la plus grande ? Qu'en concluez-vous sur le boutisme (endianness) sur votre machine ?
— Lisez le programme `tableau.c`. Que fait-il ?
— Compilez et exécutez ce programme. Si `ADR` est l'adresse affichée par le programme, demandez l'affichage du contenu des cases mémoires situées aux addresses : `ADR+1`, `ADR+2` et `ADR+4`. Pourquoi les valeurs affichées ne sont pas toutes les mêmes ?
— Compte-tenu du codage que vous avez constatez précédemment, expliquez les valeurs que vous obtenez pour `ADR+1`, `ADR+2` et `ADR+4`.
— A la fin du programme `tableau.c`, incrémentez la variable `addr` de 1, puis affichez `addr` et `*addr`. Que constatez-vous ? D'après vous, pourquoi a-t-on besoin de spécifier le type pointé lors de la déclaration d'un pointeur en C (`int* addr;`) ?

4 Exécution d'un programme en mémoire

La mémoire vive sert principalement à contenir un programme à exécuter. Un programme chargé en mémoire est structuré en 4 parties :

1. les données statiques (variables globales connues à la compilation),
2. les instructions,
3. le tas qui contient les données allouées à l'exécution et qui n'ont pas de portée locale (croit vers le bas)

4. la pile qui contient les variables à partée locale, paramères de fonctions, adresses et valeurs de retour des fonctions (croit vers le haut).

Les deux premières parties ont une taille fixe, tandis que les deux dernières ont une taille variable.

L'unité centrale exécute un programme en lisant et traitant une à une les instructions et les donnés en mémoire centrale et en y écrivant les résultats. Ces opérations se font par petits pas successifs à chaque top de l'horloge interne. L'unité centrale charge l'adresse de la prochaine instruction à exécuter dans un registre spécial, charge l'instruction correspondante, l'exécute et parallèlement incrémente l'adresse de la prochaine instruction ; mais certaines instructions (branchements, appels ou retours de fonctions) peuvent changer cette adresse.

— Lisez attentivement le programme `addition.c`, puis compilez-le (`make addition`).
— Nous allons utiliser `gdb` pour tracer finement l'exécution du programme. Tapez `gdb addition`.
— Dans l'invite de commandes gdb, tapez `disassemble main`, `disassemble addition5`. Comparez ce code assembleur avec le code C (aidez-vous des conventions de notation de l'assembleur en annexe).
— Ajoutez les points d'arrêt suivants pour examiner les registres et la pile au cours de l'exécution : `break main`, puis `break addition5`. Une fois que vous aurez lancé l'exécution du programme par la commande `run`, votre programme s'arrêtera successivement à chacun des points d'arrêt ci-dessus.
— Au premier point d'arrêt (main), tapez `disassemble main` pour repérer à quelle instruction vous en êtes et `i r` (pour `info registers`) afin de vérifier l'état des registres. L'adresse de l'instruction marquée par une flèche correspond-elle à celle de la prochaine instruction à exécuter stockée dans le registre `rip` ? Pour la suite, notez l'adresse de `rbp` et l'adresse située à `main<+25>`.
— On peut aussi taper `x /20xg $rsp` pour voir l'état de la pile (on examine en hexa les 20 mots suivants celui du haut de la pile dont l'adresse est donnée par le registre `rsp`).
— Tapez `c` (pour `continue`) afin d'atteindre le deuxième point d'arrêt (addition5). Tapez successivement `disassemble addition5`, `i r`, puis `x /20xg $rsp`. Où se trouvent l'ancienne valeur de `rbp` et l'addresse de retour, c'est-à-dire l'adresse de la prochaine instruction à exécuter au retour de la fonction (`main<+25>`) ?
— En examinant les adresses plus petites que celle du registre `rsp` (par `x /20xg` suivie d'une adresse bien choisie), découvrez où est stockée la valeur de l'argument (1). Aidez-vous de la ligne `addition5<+4>`.
— Pour voir où est stockée la valeur de la variable `res` (contenant 5 puis 6), nous allons ajouter un point d'arrêt par la commande `break *ADR2` où `ADR2` est l'adresse située à `addition5<+20>`.
— Encore par la commande `x /20xg`, découvrez où est stockée la valeur de la variable `res` en vous aidant de la ligne `addition5<+7>`.

— Enfin, devinez pourquoi le code assembleur d'une fonction commence toujours par les instructions `push %rbp` et `mov %rsp,%rbp`

5 Mémoire virtuelle

5.1 pagination

Une adresse mémoire est codée sur un mot binaire *machine*, par exemple de 32 bits. Il y a au total 2^{32} adresses possibles. Comme une adresse identifie un octet, cela fait un espace mémoire adressable de 4 Gio (Ki, Mi, Gi sont respectivement un kilo-, mega-, gigaoctet binaire valant respectivement 2^{10}, 2^{20} ou 2^{30} octets). On découpe cet espace en *page* dont la taille est généralement de 2^{12} octets soit 4 Kio. Dans cet espace, une adresse est interprétée comme un numéro de page (sur les 20 premiers bits) et une adresse relative dans cette page (offset, sur les 12 derniers bits). La mémoire réelle est découpée en *cadre de page* de même taille. L'unité de gestion mémoire (MMU pour Memory Management Unit, proche voire dans l'unité centrale), à l'aide d'une table de correspondance, a pour rôle de savoir quelles pages de la mémoire virtuelle sont physiquement en mémoire et où, charger les pages en mémoire centrale au fur et à mesure des besoins et les enlever.

La pagination permet de ne pas charger en mémoire la totalité d'un programme, permet d'exploiter la mémoire d'échange (si la mémoire nécessaire à l'exécution du programme dépasse la capacité de la mémoire centrale) et permet de tirer parti du principe de localité (toute une page est chargée pour un accès mémoire).

— Regardez le programme `matrice.c`. Que fait-il ? Compilez-le et mesurez le temps d'exécution pour initialiser des matrices de taille croissante en les parcourant par ligne ou par colonne. Par exemple `time ./matrice 100 0` donne le temps d'exécution pour initialiser une matrice 100 par 100 par colonne. Qu'observez-vous ? Comment expliquez-vous ce résultat ?

— Que fait le programme `bombeswap.c` ? Quel effet produit-il quand vous l'exécuter pour des arguments croissants 1, 2, etc. Comment expliquez-vous ce phénomène ?

5.2 Segmentation

La pagination ne résoud pas les problèmes de sécurité, de gestion dynamique de la mémoire, du chargement dynamique des bibliothèques. C'est pourquoi, un programme est aussi organisé en un ensemble de segments logiques (qui correspondent à une partie bien précise, ayant des permissions particulières : texte du programme, pile, fonction, tableau, etc.), indépendants et de taille variable. Ils comportent un nombre entier de pages si bien qu'une adresse peut être interprétée aussi en un numéro de segment, de page dans le segment et d'adresse dans la page.

— Lancez `tableau` dans deux fenetres *shell* différentes. Si `ADR` est l'adresse affichée par le premier processus, demandez l'affichage du contenu de la case mémoire située à l'adresse `ADR` dans le second processus. Que se passe-t-il ? Demandez l'affichage du

contenu de la case mémoire située à l'adresse `ADR` dans le premier processus cette fois-ci. Que se passe-t-il ? L'adresse `ADR` est-elle une adresse virtuelle ou une adresse physique ?

— Qu'est-ce qu'une erreur de segmentation (*segmentation fault*) ? Quand se produit-elle ?

— Exécutez à nouveau `tableau` et repérez son numéro de processus (par exemple en le lançant en tâche de fond avec `&`, puis `fg` pour le remettre au premier plan). Un répertoire a été créé dans `/proc` avec comme nom son numéro de processus. Consultez le fichier `status`, de ce répertoire. Que vaut la ligne `VmSize`. Qu'est-ce que cela signifie ?

— Que font les programmes `pile.c` et `tas.c` ? Compilez et exécutez ces programmes. Consultez les lignes `VmData` et `VmStk` du fichiers `/proc/[pid]/status` avant et après l'instruction de création du tableau. Qu'observez-vous ?

— Le fichier `fib.c` contient une fonction `fib` qui calcule la suite de fibonnaci. Produisez les exécutables `fibdyn` et `fibstat` (`make fibdyn fibstat`). Quelle est la différence entre entre les deux exécutables ? Faites `ls -l`, puis utilisez la commande `ldd` pour vérifier votre hypothèse (faire `man ldd`).

— Lancez `fibdyn` et `fibstat`. Qu'observez-vous en comparant les lignes `VmExe` et `VmLib` du fichier `/proc/[pid]/status` ?

— Dans le sous-répertoire de `/proc` correspondant à un processus, le fichier `maps` relie des zones dans la mémoire virtuelle associée au processus à certains fichiers exécutables qui doivent être chargé en mémoire dans ces zones. D'après vous, dans quelles plage d'adresses virtuelles se trouve le point d'entrée de la fonction `fib` et `printf` dans les programmes `fibstat` et `fibdyn` ?

6 Attaque par *buffer overflow*

— Que fait le programme `vulnerable.c` ? Compilez-le et exécutez-le plusieurs fois en passant un argument de longueur croissante ? Pourquoi est-ce un code vulnérable ?

— Nous allons d'abord tracer l'exécution avec gdb en tapant `gdb vulnerable -silent`.

— Ajoutez les points d'arrêt suivants :

1. `break main`
2. `break *ADR` où `ADR` est l'adresse `foo<+45>` (avant `strcpy`)
3. `break *ADR2` où `ADR2` est l'adresse `foo<+53>` (après `strcpy`)

— Tapez `run $(python -c 'print "A" * 40')` pour démarrer l'exécution. La commande python sert à passer une chaîne de 41 octets (40 caractères A et le caractère de fin de chaîne).

— Tapez `c` pour rejoindre le deuxième point d'arrêt. Repérez où se trouve l'adresse de retour (`main<+50>`) dans la pile avec `x /20xg $rsp`.

— Tapez à nouveau `c` pour rejoindre le troisième point d'arrêt. Sachant que le code ASCII du caractère A (resp. caractère de fin de chaîne) vaut 0x41 (resp. 0x0),

repérez avec `x /20xg $rsp` où se trouve les données dans la pile et qu'est devenu l'adresse de retour.
— En passant un argument bien choisi, essayez d'écrire une adresse permise dans l'adresse de retour.
— Un shellcode est une chaîne de caractères qui représente un code binaire exécutable, à l'origine destiné à lancer un shell pour prendre le contrôle de la machine. L'idée de l'attaque par *buffer overflow* consiste à passer ce shellcode en argument pour qu'il soit copié en mémoire et à le compléter pour remplacer l'adresse de retour par l'adresse du premier octet du shellcode. Essayer avec la chaîne de caractère fournit dans `shellcode.txt`.
— Enfin, observez l'adresse de la première case du tableau pour différentes exécutions de `tableau`, directement dans le *shell* ou dans gdb. Que se passe-t-il ?

A Conventions et syntaxe de l'assembleur

- `%r` : registre
- `(%x)` : contenu d'un registre
- `$v` : valeur immédiate
- `v(%r)` : décalage : prend le contenu du registre `%r` augmenté de la valeur `v` (le contenu de `%r` n'est pas modifié)
- `push a` : empile le contenu de a dans la pile et décrémente `%rsp`, le pointeur de pile
- `pop a` : dépile un élément de la pile en stockant sa valeur dans a et incrémente `%rsp`, le pointeur de pile
- `mov a,b` : copie a dans b
- `sub a,b` : soustrait a à b et stocke le résultat dans b
- `add a,b` : ajoute a à b et stocke le résultat dans b
- `call a` : appelle une fonction, empile le compteur ordinal `%rip` sur la pile et se branche à l'adresse a
- `ret` : retour d'une fonction, dépile la valeur de `%rip` (la prochaine instruction à exécuter est donc celle suivant le dernier `call`)