Voyage à la frontière du noyau

Thomas Petazzoni et David Decotigny

8 novembre 2003

Résumé

Cet article permet de voir de manière concrète comment un programme utilisateur simple interagit avec les différents mécanismes étudiés dans l'article *Concepts fondamentaux et structure du noyau Linux*.

Introduction

La manipulation que l'on vous propose consiste à intercepter les appels système d'un *shell* qui permettent de lancer une commande tapée au clavier. Ceci amènera à étudier le comportement général indiqué dans la figure 1.

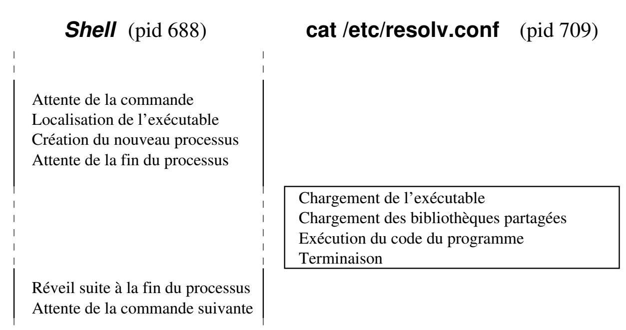


FIG. 1 – Principe de l'expérimentation

Nous avons effectué ces manipulations en utilisant l'outil `strace`, standard sous Linux et équivalent à `truss` sous Solaris, qui permet de tracer les appels système. Il existe un outil analogue, moins répandu, pour tracer les appels de fonctions vers les bibliothèques partagés : `ltrace`.

1 Mise en œuvre

Pour cet article, nous avons effectué la manipulation suivante :

- Dans un terminal (console, `xterm`, ...), taper "`echo $$`" qui affiche l'identifiant du processus (`pid`) correspondant au *shell*. Ce numéro est noté `pid_shell` ci-dessous.
- Dans un autre terminal, taper "`strace -o trace-cat.txt -f -p pid_shell`" qui permet de stocker dans le fichier `trace-cat.txt` l'ensemble des appels système effectués par le *shell* (paramètre `-p pid_shell`) et ses fils (paramètre `-f`).
- Dans le premier terminal, taper "`cat /etc/resolv.conf`"
- Dans le second terminal, tuer `strace` en tapant `Contrôle C`

Dans la suite de l'article, nous analysons le contenu du fichier `trace-cat.txt` produit comme ci-dessus. Le premier nombre sur chaque ligne dénote l'identifiant du processus qui effectue l'appel système, les colonnes suivantes indiquent l'appel système effectué, ses paramètres, et le code de retour :

```
688   open("/lib/libc.so.6", O_RDONLY)  = 3
```

Dans nos traces, le `pid` numéro `688` dénote le *shell*, et `709` dénote `cat /etc/resolv.conf`. Certains appels système modifient le/les paramètres qui leur sont passés (par référence), dans ce cas `strace` indique la valeur du paramètre après modification. C'est le cas par exemple du deuxième paramètre de l'appel suivant :

```
688 stat64("/bin/cat", {st_mode=S_IFREG|0755, st_size=13912, ...}) = 0
```

2 Analyse et interprétation

2.1 Saisie de la commande

Lorsque `strace` a été lancé, le *shell* était *bloqué* sur le descripteur de fichier numéro 0, c'est-à-dire l'*entrée standard* (*i.e.* le clavier), ce qui correspondait à la trace :

```
688   read(0,
```

Le *thread* du *shell* est donc en attente de la disponibilité de la ressource *entrée standard*. Il n'est donc pas éligible par l'ordonnanceur.

À l'appui sur une touche, le clavier déclenche une interruption matérielle, dont la routine de traitement au sein du noyau signale que la ressource *entrée standard* est désormais disponible. Le *thread* du *shell* passe dans l'état *prêt* et devient donc éligible, ce qui se traduit par la poursuite de son exécution. À la fin de cette phase, la trace devient :

```
688   read(0, "c", 1)                 = 1
```

Le *shell* se charge de reporter chaque lettre tapée sur le terminal vers la *sortie d'erreur* (descripteur de fichier numéro 2) en utilisant `write()`. Pendant le reste de cette étape, le *shell* s'occupe aussi de gérer les signaux (appels à `rt_sigprocmask()`, voir `man sigprocmask`).

2.2 Préparation au lancement de `cat`

Une fois que l'utilisateur a appuyé sur *Entrée*, le *shell* se prépare à lancer la commande. La première étape consiste à préparer les gestionnaires de signaux afin que le *shell* soit sensible au minimum de signaux. Notamment, elle conduit à masquer le signal `SIGCHLD` qui marquera la terminaison du processus fils (`cat`) qui va être lancé. Pour cela, le *shell* utilise les appels système `rt_sigprocmask()` et `rt_sigaction()`, voir `man sigaction` :

```
688   rt_sigprocmask(SIG_BLOCK, [CHLD], 8) = 0
```

La deuxième étape consiste à rechercher le programme à exécuter. Pour cela, le *shell* utilise la variable d'environnement `PATH` : pour chaque répertoire indiqué dans cette variable, il recherche le programme `cat` jusqu'à ce qu'il soit trouvé (code de retour 0 de l'appel système `stat()`) :

```
688 stat64("/sbin/cat", 0xbffff9f0)   = -1 ENOENT (No such file or directory)
688 stat64("/bin/cat", {st_mode=S_IFREG|0755, st_size=13912, ...}) = 0
```

Une fois le fichier localisé, le *shell* vérifie son type et ses droits d'accès (deuxième paramètre de `stat()`).

2.3 Lancement de `cat`

2.3.1 Création du processus fils

Le *shell* crée un processus fils copie conforme de lui-même à l'aide de l'appel système `fork()`. Cet appel retourne au *shell* le `pid` du processus fils nouvellement créé :

```
688   fork()                              = 709
```

À ce stade, les exécutions des *threads* des deux processus peuvent s'entrelacer :

```
709 getpid( <unfinished ...>
688   rt_sigprocmask(SIG_SETMASK, [],  <unfinished ...>
709 <... getpid resumed> )              = 709
688   <... rt_sigprocmask resumed> NULL, 8) = 0
```

2.3.2 Mise en attente du *shell*

Le *shell* continue de mettre en place ses gestionnaires de signaux jusqu'à faire l'appel système `wait4()` :

```
688   wait4(-1,  <unfinished ...>
```

Cet appel système met l'appelant en attente de la terminaison d'un des fils (premier paramètre = `-1`). À ce niveau, le processus `688` est bloqué dans l'attente de la terminaison de son fils, le processus `709`. La suite de l'histoire de ce *shell* se poursuit en section 2.7.

2.4 Initialisation de `cat`

Le processus `709` est toujours d'une copie du *shell* et met en place ses propres gestionnaires de signaux par des appels à `rt_sigaction` et `rt_sigprocmask`.

Le chargement réel de l'exécutable en mémoire commence avec l'appel système `execve()` :

```
709 execve("/bin/cat", ["cat", "/etc/resolv.conf"], [/* 20 vars */]) = 0
```

Celui-ci remplace l'espace d'adressage du processus courant par le contenu du fichier exécutable passé en premier paramètre. Le second paramètre est le tableau des arguments passés au programme. C'est ce tableau que l'on retrouve en paramètre de la fonction `main()` dans un programme C. Le troisième paramètre est le tableau des variables d'environnement.

2.4.1 Détermination du format du fichier binaire

Une fois dans le noyau (`fs/exec.c`), la fonction `do_execve()` appelle la fonction `search_binary_handler()` qui détermine le type du fichier binaire (*ELF*, *a.out*, *binfmt_misc*) et les fonctions responsables de son chargement. En ce qui nous concerne, `cat` est au format *ELF32*, c'est donc le gestionnaire `fs/binfmt_elf.c:load_elf_binary()` qui le prend en charge :

```
$ file /bin/cat
/bin/cat: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.0, dynamically linked (uses shared libs), stripped
```

Cette fonction se déroule en deux phases décrites ci-dessous.

2.4.2 Chargement du programme en mémoire

Un fichier binaire au format *ELF* est constitué de *sections* (`man objdump`). Chaque section est un morceau du fichier contenant un élément particulier du programme, tel que le code (`.text`), les données allouées par le compilateur (`.data` et `.rodata`), les données non initialisées par le compilateur (`.bss`), des sections de débuggage (`.stab*` et `.debug_*`), des sections pour l'utilisation et le chargement dynamique de bibliothèques (`.got`, `.plt`, `.dynstr`, `.dynsym`), etc... L'éditeur de liens (`info ld`) détermine la carte mémoire du programme à charger à partir de ces sections. Ces informations sont stockées dans le *Program Header* du fichier *ELF*. Ce *Program Header* permet à la fonction `load_elf_binary()` de connaître les parties du fichier à mapper en mémoire (marquées `LOAD` avec `objdump`) :

```
LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
     filesz 0x00002def memsz 0x00002def flags r-x
LOAD off    0x00003000 vaddr 0x0804b000 paddr 0x0804b000 align 2**12
     filesz 0x000001d4 memsz 0x00000328 flags rw-
```

La première portion correspond au code, commence à l'offset 0 dans le fichier et sera mappée en mémoire à l'adresse `0x8048000` avec les droits lecture/exécution.

Même principe pour la deuxième portion qui correspond aux données. Cette fonction détermine aussi la taille de la section de données non initialisées (`.bss`). À l'issue de ces opérations, la liste des régions virtuelles (`cat /proc/709/maps`) est la suivante :

```
08048000-0804b000 r-xp 00000000 f0:00 3034114   /bin/cat
0804b000-0804c000 rw-p 00003000 f0:00 3034114   /bin/cat
0804c000-0804d000 rwxp 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Les trois premières régions correspondent aux deux entrées dans le *Program Header* et au `.bss`. La dernière correspond à la pile utilisateur initiale.

2.4.3 Chargement de `ld-linux.so.2`

La fonction `load_elf_binary()` cherche une section de nom "`.interp`". Celle-ci contient le chemin de l'*interpréteur* qui s'occupera du chargement des bibliothèques partagées (voir ci-dessous). Sous Linux, elle contient la chaîne `/lib/ld-linux.so.2`. La fonction `load_elf_interp()` est alors appelée pour mapper le fichier correspondant dans l'espace d'adressage du processus. L'adresse du mapping est indifférente à cet interpréteur : le noyau (fonction `mm/mmap.c:get_unmapped_area()`) choisit la première plage suffisamment grande disponible dans l'espace d'adressage, à partir de l'adresse `include/asm/processor.h:TASK_UNMAPPED_BASE`, qui vaut 0x40000000 sur x86. Deux nouvelles régions apparaissent alors dans l'espace d'adressage du nouveau processus (`ld-linux.so.2` est un lien vers `ld-2.3.1.so`) :

```
40000000-40011000 r-xp 00000000 f0:00 6335162   /lib/ld-2.3.1.so
40011000-40012000 rw-p 00011000 f0:00 6335162   /lib/ld-2.3.1.so
```

Le noyau indique enfin que la première instruction à exécuter par le thread du nouveau processus sera la première instruction de l'interpréteur.

2.5 Chargement des bibliothèques partagées

Une fois l'appel système `execve()` terminé, `ld-linux.so.2` prend la main dans le tout nouvel espace d'adressage pour charger les bibliothèques partagées. Celles-ci sont déterminées à partir des sections `.dynstr` et `.dynsym`. Pour chaque bibliothèque partagée, il réalise les opérations suivantes :

1. Ouvre le fichier correspondant :

   ```
   709 open("/lib/libc.so.6", O_RDONLY)  = 3
   ```

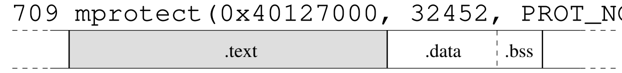
2. Lit le début du fichier pour récupérer le *header ELF* afin de déterminer le nombre de sections, leur taille, les protections (appel système `read()`).
3. Mappe la bibliothèque partagée en lecture/exécution dans une région de la taille nécessaire :

   ```
   709 old_mmap(NULL, 1113796, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x4001f000
   ```

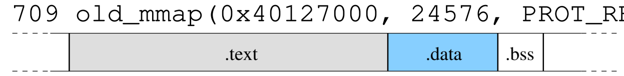
4. Modifie les protections sur la zone mémoire, car les droits lecture/exécution ne sont pas adaptés au fonctionnement de la bibliothèque partagée : les données doivent être en lecture/écriture, etc... Pour cela, il :
   (a) retire tous les droits d'accès sur la portion de la région contenant les données en utilisant `mprotect()`. Ceci a pour effet de remplacer une partie de la première région par une nouvelle région :

   ```
   709 mprotect(0x40127000, 32452, PROT_NONE) = 0
   ```

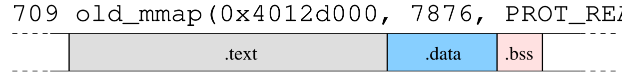
   (b) restaure des droits d'accès en lecture/écriture pour la partie de la région nouvellement créée correspondant aux données allouées par le compilateur (`.data`) :

   ```
   709 old_mmap(0x40127000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x107000) = 0x40127000
   ```

   (c) configure le reste de la région correspondant aux données non initialisées (`.bss`) en mode *anonyme* :

   ```
   709 old_mmap(0x4012d000, 7876, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4012d000
   ```

5. Ferme le fichier (appel à `close()`). Les mappings qui viennent d'être créés restent valides.

Dans le cas de `cat`, seule la bibliothèque partagée `libc` est chargée.

À l'aide de la variable d'environnement `LD_TRACE_LOADED_OBJECTS`, il est possible de demander à `ld-linux.so.2` d'effectuer toutes ces opérations en affichant les bibliothèques partagées et leur adresse de chargement. C'est ainsi que fonctionne le script `ldd`.

2.6 Fonctionnement de `cat`

Ici commence (enfin) la fonction `main()` de `cat`. Cette fonction :

1. vérifie que la *sortie standard* est bien valide :

   ```
   709 fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(4, 2), ...}) = 0
   ```

2. ouvre le fichier `/etc/resolv.conf`

   ```
   709 open("/etc/resolv.conf", O_RDONLY|O_LARGEFILE) = 3
   ```

3. récupère la taille de ce fichier

   ```
   709 fstat64(3, {st_mode=S_IFREG|0644, st_size=97, ...}) = 0
   ```

4. lit son contenu

   ```
   709 read(3, "# Insert nameservers here\n# name"..., 4096) = 97
   ```

5. affiche son contenu sur la *sortie standard*

   ```
   709 write(1, "# Insert nameservers here\n# name"..., 97) = 97
   ```

6. ferme le fichier et la *sortie standard* par appel à `close()`
7. quitte

   ```
   709 exit_group(0)                    = ?
   ```

2.7 Reprise du *shell*

Le *shell* reprend alors la main durant l'appel système `wait4()` dans lequel il était resté bloqué :

```
688   <... wait4 resumed> [WIFEXITED(s) && WEXITSTATUS(s) == 0], WUNTRACED, NULL) = 709
```

Il effectue diverses opérations internes, puis réactive les signaux, dont `SIGCHLD` qui avait été masqué (voir la section 2.2) :

```
688   rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
```

Cette réactivation provoque l'arrivée du signal `SIGCHLD` qui était en attente, puisque le fils est terminé :

```
688   --- SIGCHLD (Child exited) @ 0 (0) ---
```

Le *shell* s'assure qu'aucun autre fils est en cours d'exécution ou terminé :

```
688   wait4(-1, 0xbffff83c, WNOHANG|WUNTRACED, NULL) = -1 ECHILD (No child processes)
```

Ensuite, le *shell* calcule notre coefficient de sociabilité :

```
688   stat64("/var/mail/lAm3rZ", 0xbffff410) = -1 ENOENT (No such file or directory)
```

Il remet en place ses gestionnaires de signaux, puis réaffiche le prompt :

```
688   write(2, "\33[1;36mroot@tty2[root]#\33[0;39m \33"..., 34) = 34
```

Enfin, il attend de nouveau la frappe de touches sur l'*entrée standard* :

```
688   read(0,  <unfinished ...>
```

Goto section 2.1 pour de nouvelles aventures...

Conclusion

Nous espérons que cet article a permis de poursuivre la démystification du fonctionnement d'un système d'exploitation, entamée dans l'article *Concepts fondamentaux et structure du noyau Linux*. Il vous reste encore bien d'autres appels système à découvrir, ce qui une bonne manière de préparer votre voyage au cœur du noyau.

Thomas Petazzoni et David Decotigny
Projet KOS (`http://kos.enix.org`)
`Thomas.Petazzoni@enix.org` et `d2@enix.org`