242 exec et recouvrement de processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Recouvrement de processus

Principe : demander à changer de programme exécuté

- Vérifier l'existence et droits d'exécution
- Écraser le segment de code avec le nouvel exécutable
- Écraser les données statiques
- Réinitialiser tas et pile
- Positionner correctement les registres
- Mettre à jour les données internes du SE

Autodestruction sans risque

Autodestruction car

- Pendant un recouvrement, code et données sont inutilisables
- À la fin il n'en reste rien

Mais c'est sans risque car

- Tout est fait par le SE avec les données du SE
- Le code et les données du processus ne participent pas au recouvrement (heureusement)

Appel système exec

En fait une famille de fonctions

Un appel système (section 2)

```
int execve(const char *filename, char *argv[], char *envp[])
```

Fonctions pratiques (section 3)

`execl, execlp, execle, execv, execvp`

- v : passage par vecteur (`char *argv[]`)
- l : passage par liste (`char *argv, ...`)
- p : utilisation de PATH pour trouver l'exécutable
- e : précision des variables d'environnement

Choses perdues après un execve

Perdu : quelques trucs

- Segments mémoires (code, données statiques, tas, pile, etc.)
- Threads
- Gestionnaires de signaux...

Conservé : tout le reste

- Identité : pid, parent, etc.
- Caractéristiques : Utilisateur, droits, priorité, etc.
- Entrées sorties : répertoire courant, fichiers ouverts, etc.
- Statistiques : consommation ressources

Exemples de exec

Utilisation de execl

```
execl("/bin/ls", "ls", "/etc", NULL);
perror("Échec du exec");
exit(1);
```

Utilisation de execlp

```
execlp("ls", "ls", "-l", "/usr", NULL);
perror("Échec du exec");
exit(1);
```

Exemple exec.c

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  printf("On exécute %s avec %d arguments!\n",
      argv[1], argc-2);
  execvp(argv[1], argv+1);
  perror(argv[1]);
}
```

Chargement des exécutables

Format des exécutables binaires

- ELF pour Unix
- PE pour Windows
- D'autres formats historiques ou +/- répandus existent

$\rightarrow$ Un noyau pourrait connaitre plusieurs formats

ELF: Executable and Linking Format

- `elf(5)` pour le format
- `objdump(1)` ou `readelf(1)` pour afficher l'information
- `nm(1)` pour juste lister les symboles

Contenu des exécutables binaires

De l'information pour le système

- Quels blocs d'octets charger ?
- À quelle adresse dans la mémoire ?
- Avec quels droits rwx ?
- Quelle est la taille du BSS ?
- Etc.

Et plein d'autres choses

- Pour l'éditeur de liens
- Pour l'éditeur de liens dynamiques
- Pour le débogueur
- Etc.

Point d'entrée

Indique l'adresse de la première instruction machine du programme

Symbole `_start`

- Sous Unix, c'est souvent le point d'entrée
- L'éditeur de liens décide en fait

Quoi faire entre `_start` et `main` ?

- Charger les bibliothèques (dont la libc!)
- Préparer des segments mémoires
- Instancier des objets globaux

$\rightarrow$ Le processus est responsable

Démarrer sans rien

```
// pas de libc ni rien
// gcc nostart.c -nostdlib -static -e debut -o nostart

int ecrit(int fs, char* msg, long len)
  { asm("mov $1, %rax; syscall"); }
int quitte(int code)
  { asm("mov $60, %rax; syscall"); }

void debut(void) {
  ecrit(1, "Hello, World!\n", 14);
  quitte(0);
}
```

- Pas portable, mais ça fonctionne

Interpréteurs de scripts (*shebang*)

- Si un fichier est exécutable et commence par « `#!` »
  Exemple: « `#!/chemin/foo argument` »
- Le système exécute le programme `/chemin/foo`
- Avec le chemin du fichier en argument

→ Ça peut être n'importe quel programme

→ C'est automatique

```
$ cat shebang
#!./showargs monargument
$ ./shebang a b c
arg 0: ./showargs
arg 1: monargument
arg 2: ./shebang
arg 3: a
arg 4: b
arg 5: c
exe: /usr/local/bin/showargs
```

Utilisation habituelle du shebang

Exécuter un programme dans un langage de script

- Exemple « `#!/bin/bash` »

Chemin absolu

Problème: il faut un chemin absolu

- Solution utiliser `/usr/bin/env`
- `env(1)` exécute un programme trouvé dans le `PATH`
- Exemple « `#!/usr/bin/env python` »

Question

- Pourquoi `#` ?

Utilisation inhabituelle

Qu'affiche ce programme ?

```
#!/usr/bin/tac
   (/`-'\)
    /    \(
      )U(   _)
   =(_*_)= (
  '\`o.O'    _
   _   ,/|
```

Formats binaires divers

`binfmt_misc` (*miscellaneous binary format*)

- Permet d'associer des interpréteurs à des binaires quelconques
- En fonction de l'extension ou d'un nombre magique

Exemples

- Exécuter des `.jar` directement avec `java`
- Exécuter des `.exe` directement avec `wine`

```
$ file hello.jar
hello.jar: Java archive data (JAR)
$ ./hello.jar
Hello world!
```

Exécutables dynamiquement liés

- En gros, une bibliothèque dynamique exécutable
- Mode de compilation par défaut des distributions modernes
- On parle de PIE (*position independent executable*)

Problème

Pour être utilisable, une bibliothèque doit être liée

Solution

- Champ `PT_INTERP` (ELF) indique le chemin d'éditeur de liens dynamiques (habituellement `ld.so`)
- Le noyau le charge et l'exécute
- Qui va lier et exécuter le programme ?
- C'est de la vraie magie noire