Taller de syscalls

Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina

18 de Agosto de 2015 - 2c2015

(SisOp - DC) Taller de syscalls 18/08/2015 1 / 27

Menú

Ahora veremos¹:

- Herramienta strace
- Syscall ptrace
- Taller

Syscalls en linux - Ejemplo

tinyhello.asm

```
section .data
hello: db 'Hola SO!',10
hello_len: equ $-hello
section .text
global _start
start:
 mov eax, 4; syscall write
 mov ebx, 1; stdout
 mov ecx, hello; mensaje
 mov edx, hello_len
  int 0x80
 mov eax, 1; syscall exit
 mov ebx, 0;
  int 0x80
```

Syscalls en linux (x86_64) - Ejemplo

$tinyhello_64.asm$

```
section .data
hello: db 'Hola SO!',10
hello_len: equ $-hello
section .text
global _start
start:
 mov rax, 1; syscall write
 mov rdi, 1; stdout
 mov rsi, hello; mensaje
 mov rdx, hello_len
  syscall
 mov rax, 60; syscall exit
  mov rdi, 0;
  syscall
```

strace

strace es una herramienta que nos permite generar una traza legible de las llamadas al sistema usadas por un programa dado.

Ejemplo strace

- execve convierte el proceso en una instancia nueva de ./tinyhello y devuelve 0 indicando que no hubo error.
- write escribe en pantalla el mensaje y devuelve la cantidad de caracteres escritos (= 9).
- exit termina la ejecución y no devuelve ningún valor.

(SisOp - DC) Taller de syscalls 18/08/2015 5 / 27

Complicando las cosas: Hello en C

Mismo ejemplo pero en C.

```
hello.c
#include <stdio.h>
int main(int argc, char* argv[]) {
  printf("Hola SO!\n");
  return 0;
}
```

Compilado estáticamente:

```
compilación de hello.c
```

```
gcc -static -o hello hello.c
```

(ロ) (部) (注) (注) (注) (注) のQ((**)

(SisOp - DC) Taller de syscalls 18/08/2015 6 / 27

strace en C

strace de un programa en C, ejecutado desde la tty2

```
$ strace -q ./hello
execve("./hello", ["./hello"], [/* 17 vars */]) = 0
uname({sys="Linux", node="nombrehost", ...}) = 0
brk(0)
                                         = 0x831f000
brk(0x831fcb0)
                                         = 0x831fcb0
set_thread_area({entry_number:-1 -> 6, base_addr:0x831f830...}) = 0
brk(0x8340cb0)
                                         = 0x8340cb0
brk(0x8341000)
                                         = 0x8341000
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(4, 2), ...}) = 0
ioctl(1, SNDCTL_TMR_TIMEBASE or TCGETS,
    \{B38400 \text{ opost isig icanon echo } ...\}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
   -1, 0) = 0xb7fe6000
write(1, "Hola SO!\n", 9)
                                         = 9
exit_group(0)
                                         = ?
```

¿Qué es todo esto?

Más syscalls - memoria

Llamadas referentes al manejo de memoria.

- brk y sbrk modifican el tamaño de la memoria de datos del proceso.
 malloc/free usan estas syscalls para agrandar o achicar la memoria usada del proceso. (malloc no es una syscall y ofrece otra funcionalidad que brk)
- mmap y mmap2 asignan un archivo o dispositivo a una región de memoria. En el caso de MAP_ANONYMOUS no se mapea ningún archivo, sólo se crea una porción de memoria disponible para el programa.
 Para regiones de memoria grandes, malloc usa esta syscall.

(SisOp - DC) Taller de syscalls 18/08/2015 8 / 27

Más syscalls - salida en pantalla

Llamadas referentes a la salida en pantalla (descriptor 1).

strace ./hello - salida en pantalla

```
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(4, 2), ...}) = 0
ioctl(1, SNDCTL_TMR_TIMEBASE or TCGETS,
    \{B38400 \text{ opost isig icanon echo } ...\}) = 0
write(1, "Hola SO!\n", 9)
```

- fstat (o fstat64 en kernels nuevos) devuelve información sobre un archivo (tipo, permisos, tamaño, fechas, etc). (ver comando stat).
- ioct1 (control device) permite controlar aspectos de la terminal o el dispositivo de E/S. Por ejemplo, 1s imprimirá en colores y en columnas del ancho de la terminal. ls | cat o ls > archivo imprime de a un archivo por línea.
- Para ello, TCGETS obtiene información sobre el file descriptor.
- write escribe el mensaje en la salida.

(SisOp - DC) Taller de syscalls 18/08/2015 9 / 27

Más syscalls - otros

strace ./hello - otros

```
execve("./hello", ["./hello"], [/* 17 vars */]) = 0
uname({sys="Linux", node="nombrehost", ...}) = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x831f830...}) = 0
exit_group(0)
```

- uname devuelve información del sistema donde se está corriendo (nombre del host, versión del kernel, etc).
- set_thread_area registra una porción de memoria como memoria local del (único) thread² que está corriendo.
- exit_group termina el proceso (y todos sus threads).

10 / 27

(SisOp - DC) Taller de syscalls 18/08/2015

²Threads se verá más adelante

Syscalls - Hello world dinámico

Compilamos el mismo fuente hello.c con bibliotecas dinámicas. Corremos strace sobre este programa y encontramos aún más syscalls:

```
strace ./hello
access("/etc/ld.so.nohwcap", F_OK)
                                        = -1 ENOENT (No such file or ...)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE,
      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0) = 0xb8017000
access("/etc/ld.so.preload", R_OK)
                                        = -1 ENOENT (No such file or ...)
open("/etc/ld.so.cache", O_RDONLY)
                                        = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=89953, ...}) = 0
mmap2(NULL, 89953, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb8001000
close(3)
```

• La secuencia open, fstat, mmap2 y close mapean el archivo /etc/ld.so.cache a una dirección de memoria (0xb8001000).

18/08/2015 (SisOp - DC) Taller de syscalls 11 / 27

Muchas syscalls para un Hello world

Recordemos nuestro código del programa:

hello.c

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  printf("Hola SO!\n");
  return 0;
}
```

- El punto de entrada que usa el linker (1d) es _start.
- El punto de entrada de un programa en C es main.
- gcc usa 1d como linker y mantiene su punto de entrada por defecto.
- ¿Qué hay en el medio? ¿Alguna idea?

Muchas syscalls para un Hello world (cont.)

hello.c desensamblado

```
Disassembly of section .text:
08048130 < start>:
 8048130:
           31 ed
                                   ebp,ebp
                           xor
8048132:
           5e
                                   esi
                           pop
8048133: 89 e1
                                   ecx, esp
                           mov
8048135: 83 e4 f0
                           and
                                   esp,0xfffffff0
8048138:
           50
                           push
                                   eax
8048139:
           54
                           push
                                  esp
 804813a:
           52
                           push
                                  edx
804813b:
           68 70 88 04 08
                           push
                                 0x8048870
 8048140:
           68 b0 88 04 08
                           push
                                 0x80488b0
8048145:
           51
                           push
                                   ecx
 8048146:
           56
                           push
                                 esi
8048147: 68 f0 81 04 08
                           push
                                 0x80481f0
804814c:
           e8 cf 00 00 00
                           call
                                   8048220 <__libc_start_main>
8048151:
           f4
                           hlt
```

¡La libc!



(SisOp - DC)

Código libc

libc: función __libc_start_main

```
STATIC int LIBC_START_MAIN (int (*main) (int, char **, char **
MAIN_AUXVEC_DECL),
   int argc,
   char *_unbounded *_unbounded ubp_av,
#ifdef LIBC_START_MAIN_AUXVEC_ARG
   ElfW(auxv_t) *_unbounded auxvec,
#endif
   __typeof (main) init,
   void (*fini) (void),
   void (*rtld_fini) (void),
   void *_unbounded stack_end)
   __attribute__ ((noreturn));
```

- (ロ) (部) (注) (注) 注 り(()

(SisOp - DC) Taller de syscalls 18/08/2015 14 / 27

Ejemplos de "la vida real"

Demo strace

- date
- LC_ALL=C date
- cat
- time date



15 / 27

(SisOp - DC) Taller de syscalls 18/08/2015

Detrás de escena: ptrace

ptrace es una syscall más. Antes que nada, man es nuestro amigo.

```
man 2 ptrace
```

```
NAME
```

```
ptrace - process trace
```

SYNOPSTS

```
#include <sys/ptrace.h>
```

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Permite observar y controlar un proceso hijo. En particular permite obtener una traza del proceso, desde el punto de vista del sistema operativo.

Al llamar a la syscall ptrace desde un proceso padre el sistema operativo le "avisa" cada vez que el proceso hijo hace una syscall o recibe una señal.

(SisOp - DC) Taller de syscalls 18/08/2015 16 / 27 Vamos a usar ptrace para monitorear un proceso.

prototipo de ptrace

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

request puede ser alguno de estos:

- PTRACE_TRACEME, PTRACE_ATTACH, PTRACE_DETACH
- PTRACE_KILL, PTRACE_CONT
- PTRACE_SYSCALL, PTRACE_SINGLESTEP
- PTRACE_PEEKDATA, PTRACE_POKEDATA
- PTRACE_PEEKUSER, PTRACE_POKEUSER
- ...v más³



³Vea man 2 ptrace

Usando ptrace (cont.)

Situación:

- Proceso padre
- Proceso hijo que queremos monitorear

Inicialización, dos alternativas:

- El proceso padre se engancha al proceso hijo con la llamada ptrace(PTRACE_ATTACH, pid_child). Esto permite engancharse a un proceso que ya está corriendo (si se tienen permisos suficientes).
- 2 El proceso hijo solicita ser monitoreado haciendo una llamada a ptrace(PTRACE_TRACEME).

Finalización:

• Con la llamada ptrace(PTRACE_DETACH, pid_child) se deja de monitorear.

Usando ptrace (cont.)

ptrace permite monitorear tres tipos de eventos:

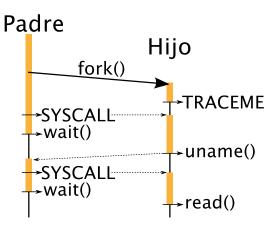
- Señales: Avisa cuando el proceso hijo recibe una señal.
- Syscalls: Avisa cada vez que el proceso hijo entra o sale de la llamada a una syscall.
- Instrucciones: Avisa cada vez que el proceso hijo ejecuta una instrucción.

Cada vez que se genera un evento el proceso hijo se detiene. Para reanudarlo hasta el siguiente evento el proceso padre puede:

- llamar a ptrace(PTRACE_CONT) para reanudar el hijo hasta la siguiente señal recibida.
- llamar a ptrace(PTRACE_SYSCALL) para reanudar el hijo hasta la siguiente señal recibida o syscall ejecutada.
- llamar a ptrace(PTRACE_SINGLESTEP) para reanudar el hijo sólo por una instrucción.

(SisOp - DC) Taller de syscalls 18/08/2015 19 / 27

Esquema de uso (simplificado)

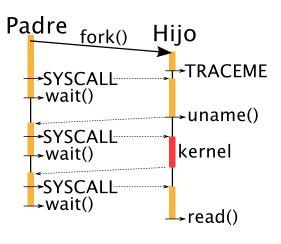


Ejemplo simplificado del mecanismo de bloqueo de ptrace.

El hijo se detiene cada vez que llama a una syscall. El padre lo reanuda con una llamada a ptrace(PTRACE_SYSCALL).

(SisOp - DC) Taller de syscalls 18/08/2015 20 / 27

Esquema de uso



En realidad, el padre recibe dos eventos, al entrar y salir de la syscall.

(SisOp - DC) Taller de syscalls 18/08/2015 21 / 27

Usando ptrace (cont.)

Esquema de comunicación:

- Se inicializa el mecanismo de ptrace (PTRACE_TRACEME o PTRACE_ATTACH).
- 2 padre: Llama a wait: espera el próximo evento del hijo.
- hijo: Ejecuta normalmente hasta que se genere un evento (recibir una señal, hacer una syscall o ejecutar una instrucción).
- 4 hijo: Se genera el evento y el proceso se detiene.
- padre: Vuelve de la syscall wait.
- padre: Puede inspecionar y modificar el estado del hijo: registros, memoria, etc.
- padre: Reanuda el proceso hijo con PTRACE_CONT, PTRACE_SYSCALL o PTRACE_SINGLESTEP y vuelve a 2.
- padre: o bien: Termina el proceso con PTRACE_KILL o lo libera con PTRACE_DETACH.

(SisOp - DC) Taller de syscalls 18/08/2015 22 / 27

Usando ptrace: launch

Veamos un programa launch para poner a ejecutar otro programa

launch.c - main()

```
/* Fork en dos procesos */
child = fork();
if (child == -1) { perror("ERROR fork"); return 1; }
if (child == 0) {
 /* S'olo se ejecuta en el Hijo */
 execvp(argv[1], argv+1);
  /* Si vuelve de exec() hubo un error */
 perror("ERROR child exec(...)"); exit(1);
} else {
  /* S'olo se ejecuta en el Padre */
 while(1) {
   if (wait(&status) < 0) { perror("waitpid"); break; }</pre>
   if (WIFEXITED(status)) break; /* Proceso terminado */
```

(SisOp - DC) Taller de syscalls 18/08/2015 23 / 27

Usando ptrace: launch + ptrace

launch.c + ptrace

```
child = fork():
if (child == -1) { perror("ERROR fork"); return 1; }
if (child == 0) {
 /* Sólo se ejecuta en el Hijo */
  if (ptrace(PTRACE_TRACEME, 0, NULL, NULL)) {
   perror("ERROR child ptrace(PTRACE_TRACEME, ...)"); exit(1);
 execvp(argv[1], argv+1);
  /* Si vuelve de exec() hubo un error */
 perror("ERROR child exec(...)"); exit(1);
} else {
 /* Sólo se ejecuta en el Padre */
 while(1) {
   if (wait(&status) < 0) { perror("wait"); break; }</pre>
    if (WIFEXITED(status)) break; /* Proceso terminado */
   ptrace(PTRACE_SYSCALL, child, NULL, NULL); /* continúa */
 ptrace(PTRACE_DETACH, child, NULL, NULL);/*Liberamos al hijo*/
```

(SisOp - DC) Taller de syscalls 18/08/2015 24 / 27

Usando ptrace - Estado del proceso hijo

ptrace permite acceder (leer o escribir) la memoria del proceso hijo:

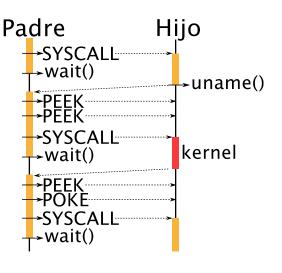
- PTRACE_PEEKDATA y PTRACE_POKEDATA Se puede leer (PEEK) o escribir (POKE) cualquier dirección de memoria en el proceso hijo.
- PTRACE_PEEKUSER, PTRACE_POKEUSER Se puede leer o escribir la memoria de usuario que el sistema guarda al iniciar la syscall: registros y estado del proceso.

Ejemplos

```
Obtenemos el número de syscall llamada:
int sysno = ptrace(PTRACE_PEEKUSER, child, 4*ORIG_EAX, NULL);
Leemos la dirección addr (direción del proceso hijo):
unsigned int valor = ptrace(PTRACE_PEEKDATA, child, addr, NULL);
Escribimos otro valor en la direccion addr:
ptrace(PTRACE_POKEDATA, child, addr, valor+1);
```

(SisOp - DC) Taller de syscalls 18/08/2015 25 / 27

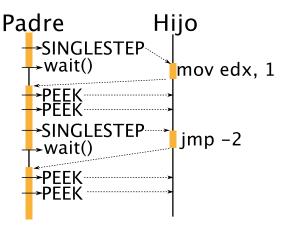
Esquema de uso - obteniendo datos



Mientras el proceso hijo está detenido, se pueden obtener y modificar datos con PTRACE_PEEKDATA, PTRACE_POKEDATA, PTRACE_PEEKUSER y PTRACE_POKEUSER.

(SisOp - DC) Taller de syscalls 18/08/2015 26 / 27

Esquema de uso - Debugger



Un debugger puede usar PTRACE_SINGLESTEP para ejecutar paso a paso cada instrucción.

(SisOp - DC) Taller de syscalls 18/08/2015 27 / 27