Taller de syscalls

Alejandro Deymonnaz[1] - actualizado por Ignacio Vissani

[1]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, 2 cuat. de 2010

Menú del día

Hoy veremos[1]:

- Syscalls (en linux)
- Herramienta `strace`
- Syscall `ptrace`
- Taller

[1] Los ejemplos de este taller NO son compatibles con 64bits

Syscalls

Las syscalls son la principal interfaz de comunicación de un proceso con el sistema operativo.
Ejemplo en UNIX: `write`

```
ssize_t write(int fd, const void *buf, size_t count);
```

Una llamada a

```
printf("Hola mundo!\n");
```

terminará en una syscall

```
write(1, "Hola mundo!\n", 12);
```

(1 es el descriptor de `stdout`)

Syscalls

Nosotros ya trabajamos con syscalls.
Algunas de las que usamos son [2] [3]:

- `kill`
- `dup2`
- `wait`
- `fork`
- `...`

[2] Para ver qué hace y cómo funciona cada syscall se puede hacer `$ man 2 {nombre de syscall}`

[3] Para poder acceder a las secciones 2 y 3 de `man` hay que instalar el paquete `manpages-dev`

Syscalls en linux

- En Linux x86 las syscalls se implementan con una *"interrupción por software"* (instrucción `int`).[4]
- Todas las syscalls usan la misma interrupción (`int 0x80`).
- El número de syscall elegida se pasa en `eax` y los primeros parámetros se pasan por registros, en orden: `ebx`, `ecx`, `edx`, `esi`, `edi`.
- Cada syscall tiene un número único. Ejemplo en x86: `_exit` es 1, `fork` es 2, `read` es 3, `write` es 4, etc. [5]
- Actualmente hay más de 300 syscalls en x86.

[4] En x86_64 existe una instrucción específica `syscall` a la cual "se le pasa" el número de syscall en el registro `RAX`

[5] Ojo: la syscall 1 en x86_64 es `write`. Por eso conviene usar las constantes deifinidas en `unistd.h`

Syscalls en linux - Ejemplo

tinyhello.asm

```
section .text

hello db "Hola SO!",10
hello_len equ $-hello

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 .text

hello db "Hola SO!",10
hello_len equ $-hello

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 utiliza la syscall `ptrace` para generar una traza legible de un comando dado.

Ejemplo strace

```
$ strace -q ./tinyhello > /dev/null
execve("./tinyhello", ["./tinyhello"], [/* 30 vars */]) = 0
write(1, "Hola SO!\n", 9)                        = 9
_exit(0)                                         = ?
```

- `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.

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
```

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 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.

strace ./hello - memoria

```
brk(0)                                  = 0x831f000
brk(0x831fcb0)                          = 0x831fcb0
brk(0x8340cb0)                          = 0x8340cb0
brk(0x8341000)                          = 0x8341000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    -1, 0) = 0xb7fe6000
```

- `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.

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 opost isig icanon echo ...}) = 0
write(1, "Hola SO!\n", 9)                = 9
```

- `fstat` (o `fstat64` en kernels nuevos) devuelve información sobre un archivo (tipo, permisos, tamaño, fechas, etc). (ver comando stat).
- `ioctl` (control device) permite controlar aspectos de la terminal o el dispositivo de E/S. Por ejemplo, `ls` 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.

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) thread[6] que está corriendo.
- `exit_group` termina el proceso (y todos sus threads).

[6] Threads 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)                                = 0
...
```

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

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 (`ld`) es `_start`.
- El punto de entrada de un programa en C es `main`.
- `gcc` usa `ld` 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              xor    ebp,ebp
 8048132:  5e                 pop    esi
 8048133:  89 e1              mov    ecx,esp
 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`!

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));
```

Demo `strace`

- `date`
- `LC_ALL=C date`
- `cat`
- `time date`
- `./tinyhello | wc`
- `nc` (netcat interactivo)

Detrás de escena: ptrace

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

man 2 ptrace

```
NAME
       ptrace - process trace

SYNOPSIS
       #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.

Usando ptrace

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
- ...y más[7]

[7] Vea man 2 ptrace

Usando ptrace (cont.)

Situación:

- Proceso padre
- Proceso hijo que queremos monitorear

Inicialización, dos alternativas:

1. 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.

Usando ptrace (cont.)

Esquema de comunicación:

1. Se inicializa el mecanismo de `ptrace` (`PTRACE_TRACEME` o `PTRACE_ATTACH`).
2. padre: Llama a `wait`: espera el próximo evento del hijo.
3. 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.
5. padre: Vuelve de la syscall `wait`.
6. padre: Puede inspecionar y modificar el estado del hijo: registros, memoria, etc.
7. padre: Reanuda el proceso hijo con `PTRACE_CONT`, `PTRACE_SYSCALL` o `PTRACE_SINGLESTEP` y vuelve a 2.
8. padre: o bien: Termina el proceso con `PTRACE_KILL` o lo libera con `PTRACE_DETACH`.

Usando ptrace: launch

Recordemos nuestro programa launch

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; }
    if (WIFEXITED(status)) break; /* Proceso terminado */
  }
}
```

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; }
    if (WIFEXITED(status)) break; /* Proceso terminado */
    ptrace(PTRACE_SYSCALL, child, NULL, NULL); /* contina */
  }
  ptrace(PTRACE_DETACH, child, NULL, NULL);/*Liberamos al hijo*/
}
```

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);
```

Taller

Ejercicio 1:
Como parte de un proyecto de ingeniería reversa, se cuenta con un archivo ejecutable misterioso (`mister`, en la página de la materia) y se desea saber qué interacción tiene con el sistema operativo. Se debe entregar una breve explicación del mismo.

Ejercicio 2:
Se pide un programa[8] en C que ejecute el comando pasado por parámetro, no permitiéndole a este ejecutar ninguna de las *syscalls prohibidas*: `fork`, `clone`. En caso que el comando especificado intente ejecutar alguna de las *syscalls prohibidas* se debe abortar su ejecución y mostrar el mensaje: "`Syscall prohibida`".

Ejemplo

```
$ ./nofork date
Wed Sep 11 12:45:13 ART 1985
$ ./nofork time date
Syscall prohibida
```

[8] Sugerencia: Use el programa dos slides atrás