



Stephen Mcswenny, 123RF

Depuración con strace

EXCAVANDO

Después del “Hola mundo”, llega el momento de conocer con un mayor nivel de detalle la forma en que se realizan las llamadas al sistema. En esta segunda y última parte haremos una introducción a los métodos de depuración que se usan en el mundo real

POR JULIET KEMP

En el artículo del número del mes pasado [1] explorábamos el utilísimo rastreador de llamadas al sistema `strace` [2]. `Strace` nos ayuda a determinar la causa de los posibles problemas en aplicaciones caseras o programas de código abierto con errores que impiden su ejecución en nuestros sistemas. En este artículo veremos más de cerca la forma de interpretar la información obtenida mediante `strace` acerca de algunas llamadas al sistema, y describiremos ciertas formas de hacer de `strace` un programa aún más útil en situaciones prácticas.

¡Hola de Nuevo!

En el artículo anterior introdujimos el simple programa de prueba que se muestra en el Listado 1. Para ejecutar `strace` con este programa y enviar la salida a un archivo, escribimos:

```
strace -o >
archivo_de_salida.txt >
holamundo.c
```

La salida del `strace` ejecutado en el Listado 1 se muestra en el Listado 2. Una mirada más detenida a un par de las líneas de la

salida nos hará entender el comportamiento de Linux en cuanto a su relación con los archivos y la memoria.

`/etc/ld.so.nohwcap`

En el Listado 2 podemos ver cómo el programa trata de acceder, sin éxito, a `/etc/ld.so.nohwcap`. Por ejemplo, la línea 11 saca el mensaje:

```
access >
("/etc/ld.so.nohwcap", F_OK) >
= 1 ENOENT >
(No such file or directory)
```

En este caso, es un problema bastante difícil de rastrear, ya que el archivo `/etc/ld.so.nohwcap` no está demasiado documentado. Con este paso se revelan los tipos de librerías que busca el enlazador.

Si `/etc/ld.so.nohwcap` está presente, el enlazador carga la versión no optimizada de la librería (o lo que es lo mismo, la versión no optimizada para nuestro hardware en particular). En caso de no estar presente, buscará la versión optimizada.

Para obtener más información sobre `ld.so`, podemos consultar la página de manual, pero básicamente es responsable

de cargar las librerías compartidas, preparar el programa para su ejecución, y ejecutarlo. Cualquier programa Linux requiere del trabajo previo de `ld.so` para su ejecución, a menos que se haya hecho un enlazado estático.

Archivos y Memoria

Tal y como se puede apreciar al ver por primera vez la salida de este programa, una de las líneas que aparece con bastante frecuencia es `mmap`. La llamada `mmap` es responsable de mapear archivos o dispositivos a memoria. Veamos cómo se estructura una llamada `mmap` en el Listado 3, por ejemplo.

Antes de la llamada `mmap`, la salida muestra que el archivo `/etc/ld.so.cache` ha sido abierto correctamente con un descriptor de archivo de 3.

Luego hay una llamada a `fstat`. El primer argumento, 3, es el descriptor de archivo de la línea anterior.

Quizá el lector se haya percatado de que al abrir los archivos y asociarlos a un número, dicho número es mayor que 2. Esto ocurre porque los descriptores 0 al 2 ya están asignados: 0 a `stdin`, 1 a `stdout` y 2 a `stderr`.

La llamada `fstat` devuelve información sobre un archivo. El primer argumento es el descriptor del archivo que, como aquí, habrá sido configurado en una llamada `open` anterior. La siguiente sección, dentro de los paréntesis, es una estructura `stat`. En realidad, `strace` no la muestra toda, pero esta estructura `stat` contiene información acerca del archivo: ID del usuario e ID del grupo, ID de dispositivo, última modificación, etc (Si desea obtener toda la información a este respecto, puede utilizar la opción `-v` de `strace`).

A partir de la estructura `stat` podemos conocer la protección del archivo y su tamaño (en bytes). Es posible interpretar ambos valores: La protección usa la misma codificación que `chmod`, que en el caso que nos ocupa presenta permisos de lectura para propietario y grupo, así como de escritura y ejecución para el propietario (0644); y el tamaño, que no es más que un número – la otra parte del campo `mode` es una baliza que indica si se trata de un archivo regular, de un directorio o de otra cosa: `S_IFREG` significa que se trata de un archivo regular).

La llamada `fstat` devuelve 0 cuando se lleva a cabo con éxito, como aquí: Al llamarla, la función rellena la estructura `stat` que aquí se guarda.

En la llamada *mmap*, el primer argumento es una ubicación de memoria que sugiere al sistema la dirección donde debería comenzar el mapeo. Normalmente, como ocurre aquí, es cero (NULL, es decir, que el sistema tiene rienda suelta para escoger la ubicación que mejor le parezca). El siguiente número, fácilmente reconocible, es el tamaño en bytes del archivo (en este caso 43270 concuerda con el tamaño de la estructura *stat* de la llamada *fstat*). La sección *PROT** especifica la protección con la que debería contar esta memoria, que nunca deberá entrar en conflicto con el modo del archivo. De nuevo, en el caso que nos ocupa, se abre con permisos de sólo lectura (READ). Como no lo estamos ejecutando como root, no tenemos permisos de escritura ni de ejecución sobre el archivo.

Luego, la llamada *mmap* mapea los datos desde un archivo a memoria, y son éstas las opciones que especifican el funcionamiento del mapeo. Aquí, *MAP_PRIVATE* significa que cualquier cambio que se produzca sobre la región mapeada (o sea, cualquier cambio realizado en la memoria) no se compartirá con otros procesos y no se volverán a escribir en el archivo original, algo que obviamente es conveniente en el caso de un archivo de sistema, como *ld.so.cache*. El siguiente argumento es, de nuevo, 3, el descriptor del archivo. El valor final es el desplazamiento, que en este caso es 0.

El valor devuelto por *mmap* es la dirección de memoria en la que empieza el mapeo. Para saber cuándo hace limpieza el sistema tras su paso, buscamos una llama-

mada *munmap* sobre esa misma dirección de memoria, que es quien la limpia.

Nótese que esta llamada la obtenemos desde el rastreo hecho a un programa escrito en C, pero no desde el realizado sobre uno escrito en Perl. Debido a que C configura explícitamente su propia memoria, también la limpia explícitamente tras de sí (o puede hacerlo, dependiendo de los detalles de la implementación de *malloc*). Perl, por otro lado, simplemente deja que sea el sistema quien lo haga a su debido tiempo.

Mientras observamos con más detenimiento las llamadas al sistema, ¿qué ocurre

con la llamada *exit_group* con que terminan ambas salidas de *strace*? Todo lo que hace es finalizar los hilos. A diferencia de *exit*, no sólo limpia ese hilo, sino que también limpia todos los hilos del grupo actual.

Opciones

Hasta ahora hemos usado *strace* empleando muy pocas opciones: solamente *-o*, para enviar la salida a un archivo determinado. Pero existen otras que facilitan la comprensión y la depuración de los programas.

Para usar *strace* con programas complicados necesitaremos un mecanismo con el

Listado 2: Strace Completo de holamundo.c

```
01 execve("./holamundo", ["/holamundo"], [/* vars */]) = 0
02 uname({sys="Linux", node="the.earth.li", ...}) = 0
03 brk(0) = 0x501000
04 access("/etc/ld.so.nohwcap", F_OK) = 1 ENOENT (No such file
or directory)
05 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 1,
0) = 0x2afb0be09000
06 access("/etc/ld.so.preload", R_OK) = 1 ENOENT (No such file
or directory)
07 open("/etc/ld.so.cache", O_RDONLY) = 3
08 fstat(3, {st_mode=S_IFREG|0644, st_size=43270, ...}) = 0
09 mmap(NULL, 43270, PROT_READ, MAP_PRIVATE, 3, 0) = 0x2afb0be0b000
10 close(3) = 0
11 access("/etc/ld.so.nohwcap", F_OK) = 1 ENOENT (No such file
or directory)
12 open("/lib/libc.so.6", O_RDONLY) = 3
13 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200\305"...
, 640) = 640
14 lseek(3, 624, SEEK_SET) = 624
15 read(3, "\4\0\0\0\20\0\0\0\1\0\0\0GNU\0\0\0\0\2\0\0\0\6\0\0"...
, 32) = 32
16 fstat(3, {st_mode=S_IFREG|0755, st_size=1286104, ...}) = 0
17 mmap(NULL, 2344904, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
3, 0) = 0x2afb0bf0a000
18 mprotect(0x2afb0c02b000, 1161160, PROT_NONE) = 0
19 mmap(0x2afb0c12b000, 98304, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x121000) = 0x2afb0c12b000
20 mmap(0x2afb0c143000, 14280, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, 1, 0) = 0x2afb0c143000
21 close(3) = 0
22 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 1,
0) = 0x2afb0c147000
23 mprotect(0x2afb0c12b000, 86016, PROT_READ) = 0
24 arch_prctl(ARCH_SET_FS, 0x2afb0c1476d0) = 0
25 munmap(0x2afb0be0b000, 43270) = 0
26 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 51), ...}) = 0
27 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 1,
0) = 0x2afb0be0b000
28 write(1, "Hola mundo", 11) = 11
29 munmap(0x2afb0be0b000, 4096) = 0
30 exit_group(11) = ?
```

Estático o Dinámico

El enlazado estático significa que el programa aporta sus propias librerías; el enlazado dinámico, que usa las librerías del sistema. Con el enlazado dinámico sólo hace falta una copia de las librerías estándar del sistema, en vez de una copia por programa. Casi todos los programas Linux usan ya enlazado dinámico, a menos que hayan sido optimizados específicamente para reducir la sobrecarga.

Listado 1: holamundo.c

```
01 #include<stdio.h>
02
03 main()
04 {
05     printf("Hola mundo");
06 }
```

que rastrear también sus forks. El parámetro *-f* provoca el rastreo de los procesos hijos (*forks*) conforme se van creando. *-F* hace lo mismo, pero con los *vforks* (un *vfork* es un caso especial de *fork* que no hereda algunos aspectos del proceso padre).

Otro juego útil de opciones tiene que ver con *time*. Éstas vienen bien a la hora de depurar un programa que se cuelga o que parece estar ejecutándose de una forma especialmente lenta. De este modo se puede determinar el origen del problema. Las opciones *-t*, *-tt* y *-ttt*, ponen todas el momento del día al inicio de cada línea (segundos, microsegundos y segundos + microsegundos, respectivamente). El parámetro *-T* muestra el tiempo empleado en las llamadas al sistema, es decir, la diferencia de tiempo entre el inicio y el final de cada llamada al sistema.

La opción *-c* contabiliza el tiempo, las llamadas y los errores de cada llamada al sistema, y elabora un resumen a partir de esta información al salir el programa. Dicha información es útil cuando se está depurando pero no se sabe exactamente por dónde empezar, ya que ayuda a determinar la posible causa del problema.

Además podemos usar *-e trace=llamadas,separadas,por,comas* para rastrear sólo las llamadas al sistema que especifiquemos. Por ejemplo, *-e trace=open,read,write* para rastrear sólo las llamadas al sistema *open*, *read* y *write*. De este modo es posible minimizar la salida generada, que probable-

mente será analizada en busca de lo que pueden ser llamadas problemáticas.

Finalmente, para generar una salida un poco más radical usaremos *-v*, que mostrará los valores completos de *stat* y *environment*, así como de otras llamadas comunes. Usaremos la opción *-v* y echaremos un nuevo vistazo a la llamada *fstat*, como comentábamos más arriba, para disponer de toda la información devuelta por el archivo.

Scripts Envueltos

A pesar de que no siempre es posible ejecutar un comando bajo condiciones “reales” cuando algo va mal, una posible solución alternativa podría ser la escritura de un envoltorio para el script de intérprete de comandos de forma que lo podamos ejecutar vía *strace*.

Quitamos de en medio el programa *comando* renombrándolo, por ejemplo, a *comando.original*. Luego, guardamos el contenido del script del Listado 4 con el nombre *comando*.

De este modo no sólo se llama al verdadero programa *comando* con cualesquiera argumentos de los usados originalmente en el *comando* original, sino que además podemos ejecutar *strace* sobre él volcándose la salida a un nuevo archivo cada vez que se llama al script. Es posible entonces comprobar qué ocurre bajo estas circunstancias reales. No hay que olvidar mover el programa real de nuevo a su sitio al terminar.

Cajas de Arena y Punteros

Una de las mejores cosas de Linux es lo bien documentadas que están las llamadas al sistema. Con las páginas del manual es posible obtener información sobre cualquier función mostrada en la salida de *strace* mediante *man nombre-delafuncion* o *man 2 nombredelafuncion*. La documentación facilita enormemente el aprendizaje, basta con analizar la salida de *strace*.

Los programas de usuario de un sistema moderno se ejecutan dentro de una pequeña caja de arena por seguridad: No se les permite interactuar directamente con la máquina (por lo que no podemos simplemente meter números en registros como hace 20 años). En vez de eso, todo pasa a través del kernel. El hecho de que *strace* no genere la salida correspondiente a una parte del programa, no quiere decir necesariamente

que el programa se haya quedado colgado; podría estar haciendo algo dentro de su propia sandbox (cálculos, por ejemplo) que no requiere comunicación con el kernel.

Las llamadas al sistema usan a menudo punteros a estructuras de datos. Un puntero es una referencia a memoria que le dice al sistema dónde ha de buscar una determinada pieza de información. Si fuésemos máquinas nos vendría genial, pero siendo humanos, supone un inconveniente, ya que rastrear en la memoria no es fácil. Cuando *strace* ve un puntero, en vez de mostrar su valor, va hasta esa zona de memoria y nos muestra los datos que allí se almacenan. Lo que vemos en la salida de *strace* son los punteros a estructuras de datos o punteros a búferes de los que hablan las páginas de manual.

Listado 3: Llamada mmap

```
01 open("/etc/ld.so.cache",
02   O_RDONLY) = 3
03 fstat(3,
04   {st_mode=S_IFREG|0644,
05     st_size=43270, ...}) = 0
06 mmap(NULL, 43270, PROT_READ,
07   MAP_PRIVATE, 3, 0) =
08   0x2afb0be0b000
```

Listado 4: Envolviendo Otro Comando con Strace

```
01 #!/bin/sh
02 #
03 strace -o/tmp/comando_out.$$
04   /usr/bin/comando.old $*
```

Otra útil opción es la utilización de *-p PID*: Con ello se vincula *strace* al proceso con el ID de proceso especificado. De esta forma, si de repente nos encontramos con un programa que está consumiendo cantidades ingentes de CPU, es posible determinar en tiempo real qué está pasando.

Conclusión

Conociendo el aspecto que deberían presentar las llamadas al sistema y cómo funcionan éstas, es mucho más fácil detectar algo que no marcha del todo bien. A menudo se ve claramente qué falla (la apertura fallida de un archivo en particular, por ejemplo). Puede ser conveniente también echar un vistazo a los permisos de archivo.

Strace es increíblemente útil para depurar, pero también se puede usar con fines educativos con programas o comandos que funcionen perfectamente, simplemente por ver qué ocurre entre bambalinas en el sistema.

Una de las principales ventajas del uso y ejecución de Linux es que las entrañas del sistema son accesibles y están muy bien documentadas. *Strace* es una de esas herramientas con las que sacar provecho de dicho acceso. ■

RECURSOS

- [1] Primera parte del artículo sobre *strace*: “Bug Bumper”, *Linux Magazine* – Edición en Castellano, número 53, Pág. 49.
- [2] *strace*: <http://sourceforge.net/projects/strace/>