Introducción a bpftrace

Juan Carlos Sáez Alcaide

Índice

1 Introducción

La depuración del kernel Linux y de sus módulos cargables es en general una tarea mucho más compleja que la depuración de programas de usuario. En particular, en el kernel la depuración basada en breakpoints no es tan sencilla de implementar como en espacio de usuario, y su uso además ocasiona problemas de temporización significativos. Por ejemplo, poner un breakpoint en el kernel no detiene la ejecución de una función concreta de una CPU, sino de todo el sistema, lo cual provoca que no se procesen interrupciones durante esta parada de la ejecución. Esto, a su vez, puede desencadenar el mal funcionamiento de partes críticas del núcleo.

Debido a éste y otros problemas relacionados, muchos desarrolladores del kernel optan por el uso de herramientas de trazado o *tracing* para llevar a cabo la depuración en Linux. La idea general de este tipo de herramientas es permitirnos capturar el valor de parámetros de funciones y variables locales de éstas durante la ejecución del código, sin llegar a detener la ejecución del sistema operativo, y todo ello sin contaminar el código del kernel o del módulo cargable en cuestión con sentencias de depuración (p.ej., mostrar mensajes con `printk()`).

En la actualidad existen distintas herramientas de tracing para el kernel Linux, como es el caso de `bpftrace`, `systemtap` o `LTTng`. Este artículo realiza una introducción a `bpftrace`, que constituye una herramienta de tracing de nueva creación. A pesar de ser una tecnología tan novedosa, `bpftrace` ya se emplea en diversas empresas a nivel mundial (como Netflix, Facebook, Red Hat, etc.) para identificar problemas de rendimiento y de seguridad en entornos de producción. Una de las claves de su éxito es la rapidez con la que el código de trazado, que es el que permite recopilar datos durante la ejecución del código del núcleo, se inserta en el kernel.

2 Tecnologías relacionadas

`bpftrace` se basa en BPF (Berkeley Packets Filter) para capturar estadísticas y otra información del kernel Linux. BPF es una máquina virtual integrada en el kernel que procesa bytecode específico; BPF es también el nombre del lenguaje de ese bytecode (secuencias de instrucciones). La particularidad de los programas BPF es que constituyen un entorno seguro de ejecución dentro del núcleo, permitiendo extender su funcionalidad en caliente de diversas formas sin afectar a la integridad del sistema.

`bpftrace` permite al usuario escribir programas de trazado a medida usando un lenguaje de scripting de alto nivel. Cada script lanzado con `bpftrace` se convierte a código BPF y el programa (o programas) BPF correspondiente(s) se inyectan en el kernel para recabar información en tiempo de ejecución. Una vez el script finaliza, el código BPF inyectado se elimina del kernel.

En realidad el comando `bpftrace` actúa como un frontend para utilizar tecnologías de trazado de más bajo nivel que implementa el propio kernel Linux, como *tracepoints* o *kprobes*. Ambas tecnologías, de forma combinada, nos permiten capturar estadísticas –recabar valores de parámetros o variables– en puntos marcados de forma estática en el kernel (caso de *tracepoints*) o cuando se produce la llamada o el retorno de una determinada función (caso de kprobes). En este artículo, no profundizaremos en el funcionamiento interno de kprobes y tracepoints, pero la idea general es que nos permiten invocar código inyectado dinámicamente en el kernel cuando se produce un evento de interés. Por ejemplo, mediante el uso de *kprobes* desde `bpftrace` es posible recabar el valor de los parámetros de una función del kernel cuando ésta se invoca. Para más información sobre el funcionamiento a bajo nivel de las tecnologías de *tracing* del kernel, se recomienda consultar el capítulo 3 del Trabajo Fin de Grado de Lázaro Clemen, antiguo alumno de LIN.

3 Fundamentos del lenguaje de scripting de `bpftrace`

El comando `bpftrace`, que ha de ejecutarse como *root* o vía `sudo`, permite la ejecución de sentencias de trazado especificadas en la propia línea de comandos (*one liners*) o bien de scripts almacenados en ficheros con la extensión `.bt`.

Aquí se muestra un ejemplo de *one liner* de `bpftrace` (uso de opción `-e`) que imprime en pantalla el mensaje *"Function was invoked"* cada vez que se invoca la función `vfs_read()` del kernel Linux:

```
$ sudo bpftrace -e 'kprobe:vfs_read { printf("Function was invoked\n"); }
Attaching 1 probe...
Function was invoked
Function was invoked
Function was invoked
Function was invoked
Function was invoked
....
```

Otro ejemplo, sería la ejecución del siguiente script `vfs_read_count.bt`.

```
kprobe:vfs_read {
        @times_vfs_read=count();
}
```

Este script de `bpftrace` contabiliza el número de veces que se invoca la función `vfs_read()` del kernel durante la ejecución del script. Para ello, se usa una variable global llamada `times_vfs_read` que se actualiza de forma segura mediante la función built-in `count()` de `bpftrace`.

La ejecución del script puede interrumpirse en cualquier momento con `CTRL+C` :

```
$ sudo bpftrace ./vfs_read_count.bt
Attaching 1 probe...

^C

@times_vfs_read: 13
```

Un script de `bpftrace` ya sea codificado como *one liner* (argumento de la opción `-e`) o incluido en un fichero aparte, consta de una serie de sentencias de trazado que especifican qué evento o eventos se quiere capturar, y qué hacer cuando el evento se produzca. La sintaxis de una sentencia de trazado es la siguiente:

```
probe[,probe,...] /filter/ { action }
```

`probe` determina el evento específico que queremos capturar (puede haber varios); `filter` es una condición booleana opcional que puede emplearse para filtrar eventos más específicos (p.ej. seleccionar eventos por comando o pid); y `action` es un programa de dimensiones reducidas, formado por un conjunto de sentencias de asignación, llamadas a funciones (como `printf()`) o sentencias de control (p.ej., if-then-else) que indican qué hacer cuando se captura el evento. El lenguaje de scripting para codificar este programa es muy parecido a C, y se documenta de forma detallada aquí. Si bien `probe` se asocia a una tecnología de trazado del kernel (p.ej. kprobes), `action` se acaba traduciendo a código BPF que se inyecta en el kernel, y que se ejecutará cuando ocurra el evento seleccionado.

La siguiente tabla resume los tipos de probes actualmente soportados por `bpftrace`.

Tipo	Descripción
tracepoint	Punto de instrumentación estático del kernel (Linux *tracepoints*)
usdt	Punto de instrumentación estático definido en programa de usuario
kprobe	Instrumentación dinámica de la invocación de una función del kernel (tecnología *kprobes*)
kretprobe	Instrumentación dinámica del retorno de una función del kernel (tecnología *kprobes*)
uprobe	instrumentación dinámica de la invocación de una función definida en un programa de usuario (tecnología *uprobes*)
uretprobe	instrumentación dinámica del retorno de una función definida en un programa de usuario (tecnología *uprobes*)
software	Evento software definidos en el kernel (subsistema *perf events*)
hardware	Instrumentación basada en contadores hardware (subsistema *perf events*)
profile	Probe que se ejecuta cada cierto tiempo (configurable) en cada CPU
interval	Probe que se ejecuta cada cierto tiempo (configurable) en una sola CPU
BEGIN	Se ejecuta al iniciar un script de `bpftrace`
END	Se ejecuta al finalizar la ejecución de un script de `bpftrace`

En un script de `bpftrace` la mayor parte de las probes (todas menos las cuatro últimas de la tabla) se especifican usando el tipo de la probe y el nombre de evento que se desea capturar, ambos separados por ":". Por lo tanto, en el caso del ejemplo anterior `kprobe:vfs_read` denota que se desea registrar el evento `vfs_read` de tipo `kprobe`; es decir, el código asociado a la probe se ejecutará cuando se invoque la función `vfs_read()` del kernel. Mediante este tipo de probes es posible también trazar la invocación de funciones que están definidas en el código un módulo del kernel que se encuentra actualmente cargado; la sintaxis en ese caso es la misma: `kprobe:<nombre_función>`

Para otros tipos de probe como `tracepoint` o `software`, el nombre de evento indicado tras ':' no hace referencia a un nombre de función, sino que identifica un punto de traza estático o evento software específico definido en el kernel. Por ejemplo, el siguiente comando (que emplea una `tracepoint`) permite imprimir un mensaje por pantalla cuando cualquier proceso de usuario abre un fichero en el sistema:

```
$ sudo bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm, str(args->filename)) }'
Attaching 1 probe...
...
```

Para obtener los eventos disponibles para un determinado tipo de probe es posible ejecutar el comando `bpftrace -l <probe_selector>`, donde `<probe_selector>` es una cadena que especifica el tipo de probe y un patrón de evento. Por ejemplo, el siguiente comando permite listar todos los eventos de tipo `software` que podemos capturar en el kernel:

```
$ sudo bpftrace -l 'software:*'
software:alignment-faults:
software:bpf-output:
software:context-switches:
software:cpu-clock:
software:cpu-migrations:
```

```
software:dummy:
software:emulation-faults:
software:major-faults:
software:minor-faults:
software:page-faults:
software:task-clock:
```

Además de probes asociadas a puntos de traza de funciones del kernel, e incluso de funciones de un programa de usuario, `bpftrace` ofrece también las probes `BEGIN` y `END`, que cuando están definidas en el script se ejecutan al comienzo y al final del script respectivamente. Estas probes suelen emplearse para inicializar variables globales del script, o registrar el tiempo de inicio y fin (respectivamente) de la sesión de tracing de `bpftrace`.

El siguiente script ilustra un uso trivial de las probes especiales `BEGIN` y `END`:

```
BEGIN {
    printf("Hello bpftrace!\n");
}

END {
    printf("Goodbye bpftrace!\n");
}
```

Dentro de una probe es posible utilizar variables globales, locales o *built-in*. La siguiente tabla resume la notación empleada para hacer referencia a los distintos tipos variables globales y locales soportadas por `bpftrace`.

Notación	Tipo / Ámbito
`@global_name`	Diccionario (Map) / Global
`@thread_local_variable_name[tid]`	Diccionario (Map) / Global por hilo
`$scratch_name`	Local (también llamada scratch)

Para una referencia completa del uso de estas variables se ha de consultar la sección "Variables" de la guía de referencia de `bftrace`.

La siguiente tabla enumera alguna de las variables built-in (accesibles automáticamente desde cada probe mediante su identificador):

Variable	Descripción
pid	PID del proceso en nombre del cual se está ejecutando código del kernel en el ámbito de la probe
comm	Comando del programa asociado al proceso en nombre del cual se está ejecutando código del kernel en el ámbito de la probe
nsecs	Tiempo actual en nanosegundos (para registrar *timestamps*)
kstack	Traza de la pila del kernel
ustack	Traza de la pila del proceso de usuario actual
arg0...argN	Argumentos de la función (para kprobes)
args	Argumentos de un tracepoint
retval	Valor de retorno de una función (para evento de tipo kretprobe)
name	Nombre de la probe actual (para diferenciar cuando hay varias en la misma definición de probe)

4 Ejemplos

Para concluir con esta breve introducción a `bpftrace` analizaremos una serie de ejemplos prácticos haciendo uso del módulo del kernel clipboard de la Práctica 1.

Supongamos que el módulo se encuentra actualmente cargado y deseamos imprimir un mensaje por pantalla cada vez que se invoque la función `clipboard_read()`. Esto puede lograrse con `bpftrace` sin modificar el código del módulo del kernel de la siguiente manera:

```
$ sudo bpftrace -e 'kprobe:clipboard_read { printf("clipboard_read was invoked\n");}'
Attached 1 probe...
```

Para comprobar el funcionamiento de este *one liner*, procederemos a escribir un mensaje en `/proc/clipboard`, y a continuación leeremos su contenido, empleando para ello otra ventana de terminal:

```
$ echo "hello" > /proc/clipboard
$ cat /proc/clipboard
```

Mientras tanto en la otra terminal, podremos observar que se muestran dos mensajes:

```
$ sudo bpftrace -e 'kprobe:clipboard_read { printf("clipboard_read was invoked\n");}'
Attached 1 probe...
clipboard_read was invoked
clipboard_read was invoked
```

Esto revela que `read_clipboard()` se ha invocado dos veces al ejecutar `cat /proc/clipboard`. Se deja como ejercicio averiguar por qué se imprimen dos mensajes y no uno solo.

`bpftrace` también nos permite imprimir por pantalla el valor de retorno de funciones y el valor de los argumentos pasados a la función, y todo ello sin modificar el código del kernel o módulo cargable. Para ilustrar esta funcionalidad analizaremos el caso de la función `clipboard_read()`, que tiene el siguiente prototipo:

```
static ssize_t clipboard_read(struct file *filp, char __user *buf, size_t len, loff_t *off);
```

Procederemos ahora a crear un script en un fichero `trace_clipboard.bt` que imprimirá el valor de los parámetros y valor de retorno de la función. Para el acceso a esta información se usan las variables *builtin* `arg0-arg3` y `retval` de `bpftrace`. El contenido del script `trace_clipboard.bt` es el siguiente:

```
kprobe:clipboard_read {
    printf("Arguments: %p %p %d %ld\n",arg0,arg1,arg2,*arg3);
}

kretprobe:clipboard_read {
    printf("Return value: %ld\n",retval);
}
```

Nótese que el modificador de formato `%p` de `printf()` sirve para imprimir direcciones de memoria de punteros (parámetros 0 y 1 de la `clipboard_read()`).

Tras lanzar este script y repetir las mismas acciones en otra terminal (escritura con `echo` y lectura con `cat` de `/proc/clipboard`) la salida será la siguiente:

```
$ sudo bpftrace clipboard_trace.bt
Attaching 2 probes...
Arguments: 0xffff941b89e36c00 0x7fda22e25000 131072 0
Return value: 6
Arguments: 0xffff941b89e36c00 0x7fda22e25000 131072 131072
Return value: 0
```

Se deja también como ejercicio analizar esta salida para comprobar si los valores son consistentes con la ejecución de

una operación de lectura del "clipboard" cuando su contenido es `hello\n` (valor escrito previamente con `echo`).

En la actualidad no es posible imprimir con `bpftrace` el valor de variables globales definidas en el código del kernel o dentro de un módulo cargable. No obstante, se puede recurrir a la inclusión de puntos de traza estáticos en el kernel para notificar a `bpftrace` en puntos críticos de la ejecución donde sea conveniente conocer el valor de algunas de esas variables. Para ello existen dos opciones: (1) crear una `tracepoint` en el kernel, o bien (2) añadir una invocación a una función vacía que reciba como parámetro el valor de las variables que se deseen consultar en un determinado punto.

Debido a que la inserción de `tracepoints` en un módulo del kernel es una tarea árdua, se procede a continuación a ilustrar cómo crear un punto de traza mediante la segunda aproximación: la invocación de una función que no hace nada. Para ello, emplearemos de nuevo el módulo `clipboard`, que modificaremos para "capturar" el valor de la variable global `clipboard` justo después de modificar su valor tras una escritura satisfactoria.

En primer lugar añadiremos en *clipboard.c* una nueva función de traza llamada `trace_clipboard()` justo antes de la definición de `clipboard_write()`:

```
void noinline trace_clipboard(char* cur_clipboard){
  asm(" ");
}
```

La función se define mediante el modificador `noinline` y con un cuerpo definido en ensamblador pero sin instrucciones, empleando la macro `asm()`. Esto permite que el compilador no elimine la invocación a la función vacía incluso empleando niveles de optimización agresivos (p.ej., `gcc -O3`) .

Finalmente el código de la función `clipboard_write()` quedará de la siguiente forma (solo se introduce la invocación a `trace_clipboard()` tras cerrar la cadena de caracteres):

```
static ssize_t clipboard_write(struct file *filp, const char __user *buf, size_t len, loff_t *off) {
  int available_space = BUFFER_LENGTH-1;

  if ((*off) > 0) /* The application can write in this entry just once !! */
    return 0;

  if (len > available_space) {
    printk(KERN_INFO "clipboard: not enough space!!\n");
    return -ENOSPC;
  }

  /* Transfer data from user to kernel space */
  if (copy_from_user( &clipboard[0], buf, len ))
    return -EFAULT;

  clipboard[len] = '\0'; /* Add the `\0' */
  trace_clipboard(clipboard);
  *off+=len;            /* Update the file pointer */

  return len;
}
```

Para realizar la impresión de la variable `clipboard` cuando la ejecución alcanza el punto indicado por la función de traza, se puede usar el siguiente *one liner* de `bpftrace`

```
bpftrace -e 'kprobe:trace_clipboard{ printf("Clipboard contents are as follows: %s\n",str(arg0)); }'
```

El script incrustado en el comando emplea la función *builtin* `str()` para convertir el primer y único argumento –que por defecto se asume de tipo entero– a cadena de caracteres, y poder así imprimir su valor en combinación con el modificador `"%s"` de `printf()` en `bpftrace`.

Para probar este script se han de emplear dos terminales, ejecutando en uno de ellos `bpftrace`, y en el otro realizando

escrituras a `/proc/clipboard`. Una posible ejecución de los comandos (tras compilar y volver a cargar el módulo del kernel) sería la siguiente:

TERMINAL 1

```
$ sudo bpftrace -e 'kprobe:trace_clipboard{ printf("Clipboard contents are as follows: %s\n",str(arg0));
Attaching 1 probe...
Clipboard contents are as follows: hello

Clipboard contents are as follows: goodbye
```

TERMINAL 2

```
$ echo "hello" > /proc/clipboard
$ echo "goodbye" > /proc/clipboard
```

5 Referencias

- *BPF Performance Tools. Linux System and Application Observability*. Brendan Gregg. 2020 Addison-Wesley. (Libro disponible en la biblioteca de la Facultad de Informática de la UCM)
- *An introduction to bpftrace for Linux* por Brendan Gregg
- *bfpftrace Reference Guide*
- Manual de BPFTrace