Linux Debugging（六）: 動態庫注入、ltrace、strace、Valgrind

By anzhsoft | Published 2014年3月6日

實際上，Linux的調試方法非常多，針對不同的問題，不同的場景，不同的應用，都有不同的方法。很難去概括。本篇文章主要涉及本專欄還沒有涵蓋，但是的確有很重要的方法。本文主要包括動態庫注入調試；使用ltrace命令處理動態庫的調試；使用strace調試系統調用的問題；Valgrind的簡要介紹。

1. 動態庫注入

如何排除其他library的調用問題？動態庫注入（library injection）有可能會讓你事半功倍。

一個大型的軟件系統，會用到非常多的動態庫。那麼如果該動態庫的一個api調用出了問題，而調用該api的地方非常非常多，不同的調用都分散的記錄在不同的log裡。那麼，如何快速的找到是哪個調用者出的問題？當然我們可以通過動態庫注入的方式去調試。

下面的代碼hook了兩個常見的函數memcpy和socket：

```
void _init(void)

{

mtrace();

printf("HOOKing: hellon");
```

```
}

void _fini(void)

{

printf("HOOKing: goodbyen");

}

typedef void* (*real_memcpy)(void*, const void*, size_t);

void *memcpy( void *dest, const void*src, size_t size)

{

real_memcpy real = dlsym((void*)-1, "memcpy" );

printf("Coping from %p to %p, size %dn", src,dest,size);

return real(dest, src, size);

}

typedef int (*real_socket)(int socket_family, int socket_type, int
protocol);
```

```
int socket(int socket_family, int socket_type, int protocol)

{

printf(" SOCKET family %d, SOCKET type %d, SOECKT protocol %d",
socket_family, socket_type, protocol);

real_socket sock = dlsym((void*)-1, "socket");

return sock(socket_family, socket_type, protocol );

}
```

將上述代碼編譯成動態庫後，需要指定環境變量LD_PRELOAD為上述動態庫。它的作用是強制load指定的動態庫，即使不需要它。你可以在上面的動態庫裡添加你想要的任何函數。

2. ltrace

ltrace能夠跟踪進程的庫函數調用,它會顯現出哪個庫函數被調用。

還是使用hello，world進行簡單的了解吧：

```
#include

int main ()
```

```
{

printf("Hello world!n");

return 0;

}
```

使用ltrace + 命令可以啟動對任何程序的調試，上述hello world的ltrace為：

```
__libc_start_main(0x8048354, 1, 0xbf869aa4, 0x8048390, 0x8048380
puts("Hello world!"Hello world!
) = 13
+++ exited (status 0) +++
```

其實ltrace是一個不用去閱讀庫的實現代碼，而去學習庫的整體調用棧的很好的方式。當然了結合代碼你可以得到更加詳細的實現。

3. strace

strace會跟踪程序系統調用。所以如果是由於程序的系統調用出問題的話，使用strace可以很快的進行問題定位。上述hello world的strace輸出為：

```
execve("./hello", ["./hello"], [/* 30 vars */]) = 0
brk(0) = 0x83d4000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7f8a000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80846, ...}) = 0
```

```
mmap2(NULL, 80846, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f76000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "177ELF111        3 3 1   000?270"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1576952, ...}) = 0
mmap2(0xb6e000, 1295780, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6e000
mmap2(0xca5000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x137) = 0xca5000
mmap2(0xca8000, 9636, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xca8000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7f75000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7f756c0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xca5000, 8192, PROT_READ) = 0
mprotect(0xb6a000, 4096, PROT_READ) = 0
munmap(0xb7f76000, 80846) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7f89000
write(1, "Hello world!n", 13Hello world!
) = 13
exit_group(0) = ?
Process 2874 detached
```

可以看到strace的輸出非常豐富。

4. Valgrind

Valgrind**是一款用於內存調試、內存洩漏檢測以及性能分析的軟件開發工具。**

Valgrind包括如下一些工具：

1. Memcheck。這是valgrind應用最廣泛的工具，一個重量級的內存檢查器，能夠發現開發中絕大多數內存錯誤使用情況，比如：使用未初始化的內存，使用已經釋放了的內存，內存訪問越界等。這也是本文將重點介紹的部分。
2. Callgrind。它主要用來檢查程序中函數調用過程中出現的問題。
3. Cachegrind。它主要用來檢查程序中緩存使用出現的問題。
4. Helgrind。它主要用來檢查多線程程序中出現的競爭問題。
5. Massif。它主要用來檢查程序中堆棧使用中出現的問題。
6. Extension。可以利用core提供的功能，自己編寫特定的內存調試工具。

IBM Developer有一篇很好的文章：https://www.ibm.com/developerworks/cn/linux/l-cn-valgrind/