Advanced Debugging in the Linux Environment

Stephen Rago
NEC Laboratories America

Tracing is an important debugging technique, especially with nontrivial applications, because it often isn't clear how our programs interact with the operating system environment. For example, when you're developing a C program, it is common to add `printf` statements to your program to trace its execution. This approach is the most basic of debugging techniques, but it has several drawbacks: you need to modify and recompile your program, you might need to do so repeatedly as you gain more information about what your program is doing, and you can't put `printf` statements in functions belonging to libraries for which you don't have source code. The Linux operating system provides more powerful tools that make debugging easier and can overcome these limitations. This article provides a survey of these tools.

We'll start with a simple C program and see how some of these tools can help find bugs in the program. The program shown in Figure 1 includes three bugs, all marked by comments.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BSZ 128           /* input buffer size */
#define MAXLINE 100000    /* lines stored */

char *lines[MAXLINE];
char buf[32];   /* BUG FIX #2: change 32 to BSZ */
int nlines;

char *
append(const char *str1, const char *str2)
{
     int n = strlen(str1) + strlen(str2) + 1;
     char *p = malloc(n);

     if (p != NULL) {
          strcpy(p, str1);
          strcat(p, str2);
     }
     return(p);
}

void
printrandom()
{
     long idx = random() % nlines;
     fputs(lines[idx], stdout);
/*   free(lines[idx]);     BUG FIX #3: add this line */
     lines[idx] = lines[--nlines];
}

int
main()
{
     int n, doappend;
     char *str, *ostr;

     close(0); /* BUG FIX #1: remove this line */
     srandom(time(NULL));
     nlines = 0;
     doappend = 0;
     str = NULL;
     while (fgets(buf, BSZ, stdin) != NULL) {
          n = strlen(buf);
          if (n == BSZ-1 && buf[BSZ-2] != '\n')
               doappend = 1;
          else
               doappend = 0;
          if (str != NULL) {
```

```
                ostr = str;
                str = append(ostr, buf);
                while (str == NULL) {
                        printrandom();
                        str = append(ostr, buf);
                }
                free(ostr);
        } else {
                str = strdup(buf);
                while (str == NULL) {
                        printrandom();
                        str = strdup(buf);
                }
        }
        if (!doappend) {
                if (nlines == MAXLINE)
                        printrandom();
                lines[nlines++] = str;
                str = NULL;
        }
    }
    while (nlines != 0)
        printrandom();
    exit(0);
}
```

Figure 1. Simple program to reorder lines

The test program, called `reorder`, is a simple command that reads lines from standard input and outputs them in a random order. The first function, `append`, concatenates two strings passed to it. It differs from `strcat` in that the buffer corresponding to the first argument is not modified; instead, `append` allocates a new memory buffer to hold the resulting string and returns this to the caller.

The second function, `printrandom`, prints a random line that was copied from the standard input. After writing the string on the standard output, it moves the last string in the array to the location previously occupied by the one just printed, and decrements the count of saved strings. As you can see, there is a memory leak here, because we should free the memory for the string printed before overwriting its pointer, but we'll get to this shortly.

The `main` function is basically one big loop that reads a line from standard input and saves the string in an array. We use a temporary buffer to read the input and then copy the string using `strdup`. If the input string is longer than our temporary buffer, we need to continue reading more and appending new input to the end of the string until we find an input string terminated by a newline.

If we run out of memory or our array is full, we call `printrandom` to print a string from a random location in the array and free up space in the array. In the former case, we might need to call `printrandom` multiple times until enough space is freed. When we no longer have any input to read, we print out all of the saved lines in a random order.

`strace`

The first bug is somewhat contrived, but it illustrates the usefulness of the `strace` tool. When we run the program, as in

```
./reorder <file
```

it produces no output. We can gain some insight into what is happening by using the `strace` command. The beauty of `strace` is that it doesn't require you to compile your program with debugging enabled, nor does it require access to the source code. The catch is that it only traces system calls and signals; if your program doesn't make system calls and doesn't receive any signals, then `strace` won't help.

Figure 2 shows the abbreviated output of

```
strace ./reorder <file
```

(We removed the system calls involved with loading shared libraries when starting the binary.) It is easy to see that reading from standard input fails because the file descriptor is closed (the `EBADF` error

indicates we used an invalid file descriptor). To fix the bug, we need to check our test program for calls to `close` or for calls to functions that might have a side-effect of closing file descriptor 0.

```
execve("./reorder", ["./reorder"], [/* 37 vars */]) = 0
 ... (shared library access lined elided)
close(0)                                = 0
fstat(0, 0x7fff9e553e80)                = -1 EBADF (Bad file descriptor)
read(0, 0x7fe8e866e000, 8192)           = -1 EBADF (Bad file descriptor)
exit_group(0)                           = ?
```

Figure 2. Output from `strace`

`strace` has options that, among other things, allow it to trace child processes, trace processes that are already running, and save output to a file. The `strace` command is the Linux version of the System V UNIX `truss` command [7] that is based on `/proc`, the process file system [9] that originated in the Eighth Edition Research UNIX System.

`gdb`

If the input file contains a line longer than 32 characters, we can trigger the second bug, which results in the program receiving a `SIGSEGV` signal. We can observe this behavior with `strace`, but we can't get much more information other than that the signal occurred, because the signal wasn't generated as part of a system call. With `gdb`, the GNU debugger [1, 8], however, we can find out what was executed when the signal occurred and inspect the memory image of the process.

We can either run the program, allow the system to generate a core file (an image of the process state at the time of the error), and do a postmortem analysis of the file using `gdb`, or we can run the program under the control of `gdb`, which gives us the ability to do live debugging by setting breakpoints, and the like. (We can also attach `gdb` to a running process by using the `-p` option, but this won't help in this example.) Figure 3 shows the situation where we run the program under the control of `gdb`. In this case, our input file contains a line with more than 380 characters.

As shown in Figure 3, we were executing the line in the program where the input string is stored in the `lines` array. Because we compiled the test program with the `-g` option, it contains debugging information and the symbol table, so we can print the value of `nlines`. The debugger tells us that the value of `nlines` is ridiculously large (our program only supports 100,000 strings).

```
(gdb) run <in.long
Starting program: /home/sar/test/reorder <in.long

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400a9d in main () at reorder.c:69
69                  lines[nlines++] = str;
(gdb) print nlines
$1 = 943142453
(gdb) x/4c &nlines
0x6010c0 <nlines>:   53 '5'  54 '6'  55 '7'  56 '8'
(gdb) x/40c buf
0x6010a0 <buf>: 10 '\n'  0 '\000'  53 '5'  54 '6'  55 '7'  56 '8'  57 '9'  48 '0'
0x6010a8 <buf+8>:    49 '1'  50 '2'  51 '3'  52 '4'  53 '5'  54 '6'  55 '7'  56 '8'
0x6010b0 <buf+16>:   57 '9'  48 '0'  49 '1'  50 '2'  51 '3'  52 '4'  53 '5'  54 '6'
0x6010b8 <buf+24>:   55 '7'  56 '8'  57 '9'  48 '0'  49 '1'  50 '2'  51 '3'  52 '4'
0x6010c0 <nlines>:   53 '5'  54 '6'  55 '7'  56 '8'  57 '9'  48 '0'  49 '1'  50 '2'
```

Figure 3. Output from `gdb`

We can use the "x" command to examine memory locations. If we display the value of the `nlines` variable in character format, we see that it contains ASCII numbers. It is no coincidence that our long

input file line is composed mostly of ASCII numbers. If we then use the "x" command to dump the contents of memory starting at our buffer where we accumulate input, we can see that the input wrote past the end of the buffer, and overwrote the `nlines` variable. We got the signal when we tried to use the corrupted value of `nlines` as an index into our array, and subsequently referenced an invalid memory location.

The bug is that we read an amount larger than our buffer size. We can fix this by changing the dimension of the `buf` array to be `BSZ` instead of 32, thereby matching the maximum number of bytes we tell `fgets` it should return.

`valgrind`

With small input files, our program might work properly. But if we supply a huge input file, the program might run out of memory prematurely. Many programs would fail in this case, but ours would merely begin to generate output before we've read all of the input lines. Although the program still works in this case, the randomness of the output will be diminished, because we'll have a smaller set of lines to select from when choosing a line to print.

Even though our program leaks memory, we might not notice unless the circumstances are just right, because the program would appear to work in most cases. We can use the `valgrind` tool [2, 10, 11] to verify that our use of dynamically-allocated memory is correct. The `valgrind` tool replaces the `malloc` and `free` family of functions with ones that do extensive error checking. When we run

```
valgrind ./reorder <file
```

it produces output similar to that shown in Figure 4. The `valgrind` tool tells us that we leaked memory, but it can't tell us where the leak occurred. It can only tell us when we allocated the memory that was never freed. In this case, the memory was allocated when we called `strdup` to create the string that was stored in the `lines` array. To find out where the leak is, we need to inspect the program to see if there are any places where entries in the `lines` array are lost. We modify the `lines` array in only two places: in `main` when we add a new string to the end of the array, and in `printrandom` when we move a string from the end of the array into the slot just printed. The bug is in the latter case, because we should free the memory holding the string when we evict it from the array.

```
==5129== Memcheck, a memory error detector
==5129== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==5129== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==5129== Command: ./reorder
==5129==
==5129==
==5129== HEAP SUMMARY:
==5129==     in use at exit: 81 bytes in 10 blocks
==5129==   total heap usage: 10 allocs, 0 frees, 81 bytes allocated
==5129==
==5129== 32 bytes in 4 blocks are definitely lost in loss record 1 of 2
==5129==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5129==    by 0x4EBAD81: strdup (strdup.c:43)
==5129==    by 0x400A4D: main (reorder.c:60)
==5129==
==5129== LEAK SUMMARY:
==5129==    definitely lost: 32 bytes in 4 blocks
==5129==    indirectly lost: 0 bytes in 0 blocks
==5129==      possibly lost: 0 bytes in 0 blocks
==5129==    still reachable: 49 bytes in 6 blocks
==5129==         suppressed: 0 bytes in 0 blocks
==5129== Reachable blocks (those to which a pointer was found) are not shown.
==5129== To see them, rerun with: --leak-check=full --show-reachable=yes
==5129==
==5129== For counts of detected and suppressed errors, rerun with: -v
==5129== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Figure 4. Output from `valgrind`

`valgrind` can detect other misuses of dynamically-allocated memory, such as using memory after it has been freed, freeing the same buffer multiple times, and using uninitialized memory. In fact, `valgrind` is a framework for several program-analysis tools, including a heap profiler and a thread error detector, among others.

systemtap

If you're developing nontrivial software, one additional tool you might want to employ is systemtap [3, 6]. This tool is useful for investigating how your programs interact with the operating system by allowing you to trace kernel functions and search through kernel data structures while the system is running. It works by converting scripts you write into code that is compiled and inserted into the kernel on the fly.

The scripting language is reminiscent of `awk`: probe points (like patterns) are associated with actions, arrays are associative, and the syntax resembles the C programming language. A special "guru" mode allows you to write actual C code to interpret kernel data structures (some operations are easier to implement this way).

Here's an example of how systemtap can be useful: suppose you have developed a server that runs several other programs and you are worried that you might be leaking (forgetting to close) file descriptors when you call `exec`. This can be a security hole, because it could give unauthorized processes access to your files. Figure 5 shows a systemtap script for Linux 3.2.0 that prints out the open file descriptors for commands as they are started. As you can see, you need to study the Linux kernel source code to determine which functions to trace and which data structures to search. The systemtap script is adapted from the `flush_old_files` kernel function, which is called as part of `exec` processing to close all open file descriptors that have the close-on-exec flag set.

```
%{
#include <linux/fdtable.h>
%}

probe begin
{
     printf("Starting probe.  Hit ^C to stop\n");
}

function get_maxfds:long(fs:long) %{
     struct files_struct *p;
     struct fdtable *f;
     long maxfd;
     p = (struct files_struct *)THIS->fs;
     f = kread(&(p->fdt));
     rcu_read_lock();
     maxfd = kread(&(f->max_fds));
     rcu_read_unlock();
     THIS->__retvalue = maxfd;
     CATCH_DEREF_FAULT();
%}

function get_long:long(fs:long, idx:long) %{
     struct files_struct *p;
     struct fdtable *f;
     fd_set *s;
     long val;
     p = (struct files_struct *)THIS->fs;
     f = kread(&(p->fdt));
     rcu_read_lock();
     s = kread(&(f->open_fds));
     val = kread(&(s->fds_bits[THIS->idx]));
     rcu_read_unlock();
     THIS->__retvalue = val;
     CATCH_DEREF_FAULT();
%}
```

```
function get_sizeoflong() %{
     THIS->__retvalue = sizeof(long);
     CATCH_DEREF_FAULT();
%}

probe kernel.function("flush_old_files").return
{
     printf("process %s: open fds", execname());
     max_fds = get_maxfds($files);
     i = 0;
     word = 0;
     sol = get_sizeoflong();
     while (i < max_fds) {
          bits = get_long($files, word);
          word++;
          for (j = 0; j < 8*sol; j++) {
               if (bits & 1)
                    printf(" %d", i);
               bits >>= 1;
               i++;
               if (i >= max_fds)
                    break;
          }
     }
     printf("\n");
}
```

Figure 5. Sample systemtap script

To figure out which file descriptors are still open, we set a probe point at the return of the `flush_old_files` function. At this point, the kernel is done closing all of the file descriptors that have the close-on-exec flag set, so any remaining file descriptors will be available to the new program we are in the process of running.

Because it needs to read kernel memory, systemtap needs to be run with superuser privileges. Figure 6 shows sample output when we run systemtap with the script from Figure 5.

```
$ sudo stap -g openfiles.stp
Starting probe.  Hit ^C to stop
process ls: open fds 0 1 2
process ps: open fds 0 1 2
process who: open fds 0 1 2
process bash: open fds 0 1 2
process groups: open fds 0 1 2
process lesspipe: open fds 0 1 2
process basename: open fds 0 1 2
process dirname: open fds 0 1 2
process dircolors: open fds 0 1 2
process uname: open fds 0 1 2
process sed: open fds 0 1 2
process ls: open fds 0 1 2
process uname: open fds 0 1 2
process sed: open fds 0 1 2
```

Figure 6. Output generated by running the systemtap script from Figure 5

Clearly, systemtap scripts are dependent on the version of the Linux operating system being used, because the scripts contain knowledge of data structures and functions belonging to the operating system. These can change from one kernel release to another. To make the scripts more portable, systemtap is distributed with a library of *tapsets*: script libraries containing common functionality. To use the

tapsets, all you need to do is reference one of the symbols defined in a tapset; the `stap` command will automatically search the tapset directory when it parses your script and include the tapset in the compiled version of your script that it loads into the kernel.

Tracing I/O

If you are interested in understanding the I/O characteristics of your programs, you can use `strace` to trace all of the reads and writes made by your program. However, this won't show I/O operations for memory-mapped files.

When you use the `mmap` system call to map a file into the address space of your program, reading and writing occurs indirectly as a result of loads from and stores to memory locations within the mapped address range. File system reads occur as a result of referencing addresses that aren't currently backed by pages in the system's page cache. Various Linux kernel daemons will write out dirty pages on behalf of your program as part of managing the system's page cache.

To understand the I/O patterns with memory-mapped files, we can use systemtap to see when the file system receives read and write requests. However, if we need more detail, we can use the `blktrace` command to trace operations on the disk driver's queues. With the `blkparse` command, we can print the trace in a human-readable form and see every request that is placed on the disk driver's queue, as well as when each request completes, to get a low-level idea of how disk I/O behaves on the system. One way we can use this information is to determine whether the I/O patterns for a particular disk tend to be random or sequential.

If we don't need the detail associated with individual I/O requests, we can instead use the `iostat` command to print I/O statistics for the duration of our program's execution. The `iostat` command will print the amount of data read or written for each disk device on the system.

If we want to trace network I/O instead of disk I/O, we can use either `tcpdump` [4] or Wireshark [5] to capture and display network packets. Perhaps the best known use of the `tcpdump` program is in the classic text *TCP/IP Illustrated* by Rich Stevens [12].

Conclusions

There are many tools to help us debug our programs by tracing how the programs interact with the operating system environment. One word of caution, however: many of the tools change the timing of a program's execution. If a bug depends on timing, the bug can disappear while tracing the program. Nevertheless, tracing remains a useful technique to discover how programs behave.

References

[1] `http://www.gnu.org/software/gdb`.

[2] `http://valgrind.org`.

[3] `http://sourceware.org/systemtap`.

[4] `http://www.tcpdump.org`.

[5] `http://www.wireshark.org`.

[6] Frank Ch. Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen, *Architecture of systemtap: a Linux trace/probe tool* (2005).

[7] Roger Faulkner and Ron Gomes, "The Process File System and Process Model in UNIX System V," *Proceedings of the Winter 1991 USENIX Conference,* pp. 243–252 (1991).

[8] Free Software Foundation, *Debugging with GDB* (2013).

[9] Tom Killian, "Processes as Files," *Proceedings of the USENIX Association Summer Conference,* pp. 203–307, Salt Lake City (1984).

[10] Nicholas Nethercote and Julian Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07),* San Diego (2007).

[11] Julian Seward and Nicholas Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," *Proceedings of the 2005 USENIX Annual Technical Conference,* Anaheim, CA (2005).

[12] W. Richard Stevens, *TCP/IP Illustrated, Volume 1,* Addison-Wesley (1994).