LinuxFocus article number 343
http://linuxfocus.org

by Guido Socher (homepage)

About the author:

Guido likes the possibilities that an open source system like Linux offers to investigate Problems. You can really find the root cause given that you are motivated to invest time.

Why does this not work!? How to find and fix faults in Linux applications.

Abstract:

Everybody claims that it is easy to find and fix bugs in programs written under Linux. Unfortunately it is very hard to find documents explaining how to do that. In this article you will learn how to find and fix faults without first learning how an application internally works.

Introduction

From a user perspective there is hardly any difference between closed and open source systems as long as everything runs without faults and as expected. The situation changes however when things do not work and sooner or later every computer user will come to the point where things do not work.

In a closed source system you have usually only two option:

- Report the fault and pay for the fix
- Re-install and pray that it works now

Under Linux you have these options too but you can also start and investigate the cause of the problem. One of the main obstacles is usually that you are not the author of the failing program and that you have really no clue how it works internally.

Despite those obstacles there are a few things you can do without reading all the code and without learning how the program works internally.

Logs

The most obvious and simplest thing you can do is to look at file in /var/log/... What you find in those files and what the names of those logs files are is configurable. /var/log/messages is usually the file you want to look at. Bigger applications may have their own log directories (/var/log/httpd/ /var/log/exim ...).

Most distributions use syslog as system logger and its behavior is controlled via the configuration file /etc/syslog.conf The syntax of this file is documented in "man syslog.conf".

Logging works such that the designer of an program can add a syslog line to his code. This is much like a printf except that it writes to the system log. In this statement you specify a priority and a facility to classify the message:

```
       #include <syslog.h>

       void openlog(const char *ident, int option, int facility);
       void syslog(int priority, const char *format, ...);
       void closelog(void);

facility classifies the type of application sending the message.
priority determines  the  importance  of  the message. Possible
values in order of importance are:

       LOG_EMERG
       LOG_ALERT
       LOG_CRIT
       LOG_ERR
       LOG_WARNING
       LOG_NOTICE
       LOG_INFO
       LOG_DEBUG
```

With this C-interface any application written in C can write to the system log. Other languages do have similar APIs. Even shell scripts can write to the log with the command:

```
logger -p err "this text goes to /var/log/messages"
```

A standard syslog configuration (file /etc/syslog.conf) should have among others a line that looks like this:

```
# Log anything (except mail) of level info or higher.
# Don’t log private authentication messages.
*.info;mail.none;authpriv.none     /var/log/messages
```

The "*.info" will log anything with priority level LOG_INFO or higher. To see more information in /var/log/messages you can change this to "*.debug" and restart syslog (/etc/init.d/syslog restart).

The procedure to "debug" an application would therefore be as follows.

```
1) run tail -f /var/log/messages and then start the application which
fails from a different shell. Maybe you get already some hints
of what is going wrong.

2) If step 1) is not enough then edit /etc/syslog.conf and
change *.info to *.debug. Run "/etc/init.d/syslog restart" and
repeat step 1).
```

The problem with this method is that it depends entirely on what the developer has done in his code. If he/she did not add syslog statements at key points then you may not see anything at all. In other words you can find only problems where the developer did already foresee that this could go wrong.

strace

An application running under Linux can execute 3 type of function:

1. Functions somewhere in its own code
2. Library functions
3. System calls

Library functions are similar to the application's own functions except that they are provided in a different package. System calls are those functions where your program talks to the kernel. Programs need to talk to the kernel if they need to access you computer's hardware. That is: write to the screen, read a file from disk, read keyboard input, send a message over the network etc...

These system calls can be intercepted and you can therefore follow the communication between application and the kernel.

A common problem is that an application does not work as expected because it can't find a configuration file or does not have sufficient permissions to write to a directory. These problems can easily be detected with strace. The relevant system call in this case would be called "open".

You use strace like this:

```
strace your_application
```

Here is an example:

```
# strace /usr/sbin/uucico
execve("/usr/sbin/uucico", ["/usr/sbin/uucico", "-S", "uucpssh", "-X", "11"],
       [/* 36 vars */]) = 0
uname({sys="Linux", node="brain", ...}) = 0
brk(0)                                  = 0x8085e34
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40014000
open("/etc/ld.so.preload", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70865, ...}) = 0
mmap2(NULL, 70865, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40015000
close(3)                                = 0
open("/lib/libnsl.so.1", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300;\0"..., 1024)
     = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=89509, ...}) = 0
mmap2(NULL, 84768, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40027000
mprotect(0x40039000, 11040, PROT_NONE)  = 0
mmap2(0x40039000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x11)
     = 0x40039000
mmap2(0x4003a000, 6944, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
     0x4003a000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0‘X\1\000"..., 1024)
       = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1465426, ...}) = 0
```

```
mmap2(NULL, 1230884, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x4003c000
mprotect(0x40163000, 22564, PROT_NONE)  = 0
mmap2(0x40163000, 12288, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED, 3, 0x126) = 0x40163000
mmap2(0x40166000, 10276, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40166000
close(3)                                = 0
munmap(0x40015000, 70865)               = 0
brk(0)                                  = 0x8085e34
brk(0x8086e34)                          = 0x8086e34
brk(0)                                  = 0x8086e34
brk(0x8087000)                          = 0x8087000
open("/usr/conf/uucp/config", O_RDONLY) = -1 ENOENT (No such file or directory)
rt_sigaction(SIGINT, NULL, {SIG_DFL}, 8) = 0
rt_sigaction(SIGINT, {0x806a700, [],
             SA_RESTORER|SA_INTERRUPT, 0x40064d58}, NULL, 8) = 0
rt_sigaction(SIGHUP, NULL, {SIG_DFL}, 8) = 0
rt_sigaction(SIGHUP, {0x806a700, [],
             SA_RESTORER|SA_INTERRUPT, 0x40064d58}, NULL, 8) = 0
rt_sigaction(SIGQUIT, NULL, {SIG_DFL}, 8) = 0
rt_sigaction(SIGQUIT, {0x806a700, [],
             SA_RESTORER|SA_INTERRUPT, 0x40064d58}, NULL, 8) = 0
rt_sigaction(SIGTERM, NULL, {SIG_DFL}, 8) = 0
rt_sigaction(SIGTERM, {0x806a700, [],
             SA_RESTORER|SA_INTERRUPT, 0x40064d58}, NULL, 8) = 0
rt_sigaction(SIGPIPE, NULL, {SIG_DFL}, 8) = 0
rt_sigaction(SIGPIPE, {0x806a700, [],
             SA_RESTORER|SA_INTERRUPT, 0x40064d58}, NULL, 8) = 0
getpid()                                = 1605
getrlimit(RLIMIT_NOFILE, {rlim_cur=1024, rlim_max=1024}) = 0
close(3)                                = -1 EBADF (Bad file descriptor)
close(4)                                = -1 EBADF (Bad file descriptor)
close(5)                                = -1 EBADF (Bad file descriptor)
close(6)                                = -1 EBADF (Bad file descriptor)
close(7)                                = -1 EBADF (Bad file descriptor)
close(8)                                = -1 EBADF (Bad file descriptor)
close(9)                                = -1 EBADF (Bad file descriptor)
fcntl64(0, F_GETFD)                     = 0
fcntl64(1, F_GETFD)                     = 0
fcntl64(2, F_GETFD)                     = 0
uname({sys="Linux", node="brain", ...}) = 0
umask(0)                                = 022
socket(PF_UNIX, SOCK_STREAM, 0)         = 3
connect(3, {sa_family=AF_UNIX,
    path="/var/run/.nscd_socket"}, 110) = -1 ENOENT (No such file or directory)
close(3)                                = 0
open("/etc/nsswitch.conf", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=499, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
read(3, "# /etc/nsswitch.conf:\n# $Header:"..., 4096) = 499
read(3, "", 4096)                       = 0
close(3)                                = 0
munmap(0x40015000, 4096)                = 0
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70865, ...}) = 0
mmap2(NULL, 70865, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40015000
close(3)                                = 0
open("/lib/libnss_compat.so.2", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\25"..., 1024)
     = 1024
```

```
fstat64(3, {st_mode=S_IFREG|0755, st_size=50250, ...}) = 0
mmap2(NULL, 46120, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40169000
mprotect(0x40174000, 1064, PROT_NONE)   = 0
mmap2(0x40174000, 4096, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED, 3, 0xa) = 0x40174000
close(3)                                = 0
munmap(0x40015000, 70865)               = 0
uname({sys="Linux", node="brain", ...}) = 0
brk(0)                                  = 0x8087000
brk(0x8088000)                          = 0x8088000
open("/etc/passwd", O_RDONLY)           = 3
fcntl64(3, F_GETFD)                     = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)         = 0
fstat64(3, {st_mode=S_IFREG|0644, st_size=1864, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
_llseek(3, 0, [0], SEEK_CUR)            = 0
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1864
close(3)                                = 0
munmap(0x40015000, 4096)                = 0
getuid32()                              = 10
geteuid32()                             = 10
chdir("/var/spool/uucp")                = 0
open("/usr/conf/uucp/sys", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/var/log/uucp/Debug", O_WRONLY|O_APPEND|O_CREAT|O_NOCTTY, 0600) = 3
fcntl64(3, F_GETFD)                     = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)         = 0
fcntl64(3, F_GETFL)                     = 0x401 (flags O_WRONLY|O_APPEND)
fstat64(3, {st_mode=S_IFREG|0600, st_size=296, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
_llseek(3, 0, [0], SEEK_CUR)            = 0
open("/var/log/uucp/Log", O_WRONLY|O_APPEND|O_CREAT|O_NOCTTY, 0644) = 4
fcntl64(4, F_GETFD)                     = 0
fcntl64(4, F_SETFD, FD_CLOEXEC)         = 0
fcntl64(4, F_GETFL)                     = 0x401 (flags O_WRONLY|O_APPEND)
```

What do we see here? Let's look e.g look at the following lines:

```
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)   = 3
```

The program tries to read /etc/ld.so.preload and fails then it carries on and reads /etc/ld.so.cache. Here it succeeds and gets file descriptor 3 allocated. Now the failure to read /etc/ld.so.preload may not be a problem at all because the program may just try to read this and use it if possible. In other words it is not necessarily a problem if the program fails to read a file. It all depends on the design of the program. Let's look at all the open calls in the printout from strace:

```
open("/usr/conf/uucp/config", O_RDONLY)= -1 ENOENT (No such file or directory)
open("/etc/nsswitch.conf", O_RDONLY)   = 3
open("/etc/ld.so.cache", O_RDONLY)     = 3
open("/lib/libnss_compat.so.2", O_RDONLY) = 3
open("/etc/passwd", O_RDONLY)          = 3
open("/usr/conf/uucp/sys", O_RDONLY)   = -1 ENOENT (No such file or directory)
open("/var/log/uucp/Debug", O_WRONLY|O_APPEND|O_CREAT|O_NOCTTY, 0600) = 3
open("/var/log/uucp/Log", O_WRONLY|O_APPEND|O_CREAT|O_NOCTTY, 0644) = 4
open("/etc/ld.so.preload", O_RDONLY)   = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)     = 3
```

The program tries now to read /usr/conf/uucp/config. Oh! This is strange I have the config file in /etc/uucp/config ! ... and there is no line where the program attempts to open /etc/uucp/config. This is the fault. Obviously the program was configured at compile time for the wrong location of the configuration file.

As you see strace can be very useful. The problem is that it requires some experience with C-programming to really understand the full output of strace but normally you don't need to go that far.

gdb and core files

Sometimes it happens that a program just dies out of the blue with the message "Segmentation fault (core dumped)". This means that the program tries (due to a programming error) to write beyond the area of memory it has allocated. Especially in cases where the program writes just a few bytes to much it can be that only you see this problem and it happens only once in a while. This is because memory is allocated in chunks and sometimes there is accidently still room left for the extra bytes.

When this "Segmentation fault" happens a core file is left behind in the current working directory of the program (normally your home directory). This core file is just a dump of the memory at the time when the fault happened. Some shells provide facilities for controlling whether core files are written. Under bash, for example, the default behavior is not to write core files at all. In order to enable core files, you should use the command:

```
# ulimit -c unlimited

# ./lshref -i index.html,index.htm test.html
Segmentation fault (core dumped)
Exit 139
```

The core file can now be used with the gdb debugger to find out what was going wrong. Before you start gdb you can check that you are really looking at the right core file:

```
# file core.16897
core.16897: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style,
from 'lshref'
```

OK, lshref is the program that was crashing so let's load it into gdb. To invoke gdb for use with a core file, you must specify not only the core filename but also the name of the executable that goes along with that core file.

```
# gdb ./lshref core.23061
GNU gdb Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
Core was generated by './lshref -i index.html,index.htm test.html'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x40095e9d in strcpy () from /lib/libc.so.6
```

```
(gdb)
```

Now we know that the program is crashing while it tries to do a strcpy. The problem is that there might be many places in the code where strcpy is used.

In general there will now be 2 possibilities to find out where exactly in the code it goes wrong.

1. Recompile the code with debug information (gcc option -g)
2. Do stack trace in gdb

The problem in our case is that strcpy is a library function and even if we would re-compile absolutely all code (including libc) it would still tell us that it fails at a given line in the C library.

What we need is a stack trace which will tell us which function was called before strcpy was executed. The command to do such a stack trace in gdb is called "backtrace". It does however not work with only the core file. You have to re-run the command in gdb (reproduce the fault):

```
gdb ./lshref core.23061
GNU gdb Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
Core was generated by `./lshref -i index.html,index.htm test.html'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x40095e9d in strcpy () from /lib/libc.so.6
(gdb) backtrace
#0  0x40095e9d in strcpy () from /lib/libc.so.6
Cannot access memory at address 0xbfffeb38
(gdb) run ./lshref -i index.html,index.htm test.html
Starting program: /home/guido/lshref ./lshref -i index.html,index.htm test.html

Program received signal SIGSEGV, Segmentation fault.
0x40095e9d in strcpy () from /lib/libc.so.6
(gdb) backtrace
#0  0x40095e9d in strcpy () from /lib/libc.so.6
#1  0x08048d09 in string_to_list ()
#2  0x080494c8 in main ()
#3  0x400374ed in __libc_start_main () from /lib/libc.so.6
(gdb)
```

Now we can see that function main() called string_to_list() and from string_to_list strcpy() is called. We go to string_to_list() and look at the code:

```
char **string_to_list(char *string){
        char *dat;
        char *chptr;
        char **array;
        int i=0;
```

```
        dat=(char *)malloc(strlen(string))+5000;
        array=(char **)malloc(sizeof(char *)*51);
        strcpy(dat,string);
```

This malloc line looks like a typo. Probably it should have been:

```
dat=(char *)malloc(strlen(string)+5000);
```

We change it, re-compile and ... hurra ... it works.

Let’s look at a second example where the fault is not detected inside a library but in application code. In such a case the application can be compiled with the "gcc -g" flag and gdb will be able to show the exact line where the fault is detected.

Here is a simple example.

```
#include
#include

int add(int *p,int a,int b)
{
  *p=a+b;
  return(*p);
}

int main(void)
{
  int i;
  int *p = 0;   /* a null pointer */
  printf("result is %d\n", add(p,2,3));
  return(0);
}
```

We compile it:

```
gcc -Wall -g -o exmp exmp.c
```

Run it...

```
# ./exmp
Segmentation fault (core dumped)
Exit 139

gdb exmp core.5302
GNU gdb Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
Core was generated by ‘./exmp’.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
```

```
#0  0x08048334 in add (p=Cannot access memory at address 0xbfffe020
) at exmp.c:6
6         *p=a+b;
```

gdb tells us now that the fault was detected at line 6 and that pointer "p" pointed to memory which can not be accessed.

We look at the above code and it is of course a simple made-up example where p is a null pointer and you can not store any data in a null pointer. Easy to fix...

Conclusion

We have seen cases where you can really find the cause of a fault without knowing too much about the inner workings of a program.

I have on purpose excluded functional faults, e.g a button in a GUI is in the wrong position but it works. In those cases you will have to learn about the inner workings of the program. This will generally take much more time and there is no recipe on how to do that.

However the simple fault finding techniques shown here can still be be applied in many situations.

Happy troubleshooting!

2005-01-14, generated by lfparser_pdf version 2.51