Warum funktioniert das nicht? Wie man Fehler in Linuxapplikationen findet und behebt

by Guido Socher (homepage)

About the author:

Guido mag die Möglichkeiten, die ein Open Source System wie Linux beim lösen von Problemen bietet. Man kann wirklich die Ursache des Problems finden, wenn man sich entsprechend viel Zeit dafür nimmt.

Abstract:

Alle behaupten, es sei leicht in Programmen, die unter Linux geschrieben wurden, Fehler zu finden und sie zu beheben. Leider ist es sehr schwierig, Dokumentation zu finden, die erklärt, wie man das macht. In diesem Artikel lernst du, wie man Fehler findet und behebt, ohne erst zu lernen, wie die Applikation intern funktioniert.

Einführung

Aus der Benutzerperspektive gibt es kaum einen Unterschied zwischen closed und open source Systemen, so lange, wie alles einwandfrei und wie erwartet funktioniert. Die Lage ändert sich jedoch, wenn die Dinge nicht funktionieren und früher oder später kommt jeder Computerbenutzer an einen Punkt, wo etwas nicht funktioniert.

In einem closed source System gibt es normalerweise nur zwei Optionen:

- Berichte den Fehler und bezahle für das Beheben des Fehlers
- Installiere die Software erneut und bete, daß es jetzt funktioniert

Unter Linux hat man diese Optionen ebenfalls, man kann aber auch untersuchen, was der Grund des Problems ist. Eines der größten Hindernisse dabei ist meistens, daß man nicht der Autor des versagenden Programmes ist und keinerlei Ahnung hat, wie das Programm intern funktioniert.

Trotz dieser Hindernisse gibt es einige Dinge, die man tun kann, ohne den ganzen Code zu lesen und ohne zu lernen, wie das Programm intern funktioniert.

Logs

Das offensichtlichste und einfachste, was man tun kann, ist sich die log−Dateien in /var/log/... anzusehen. Was man in diesen Dateien findet und was die Namen der log Dateien sind, ist konfigurierbar. /var/log/messages ist normalerweise die Datei, die du anschauen möchtest. Größere Applikationen haben eventuell ihre eigenen log Verzeichnisse (/var/log/httpd/ /var/log/exim ...).
Die meisten Distributionen benutzen syslog als system logger und sein Verhalten wird über die Konfigurationsdatei /etc/syslog.conf gesteuert. Die Syntax dieser Datei ist in "man syslog.conf" dokumentiert.

Logging funktioniert so, daß der Entwickler des Programms eine syslog Zeile in seinem Programm einfügen kann. Dies ist ähnlich wie printf, außer das es ins System−log schreibt. Dabei muß man eine Priorität und eine Facility spezifizieren, um die Message zu klassifizieren:

```
       #include <syslog.h>

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

facility klassifiziert den Typ der Applikation, die die Message sendet.

priority entscheidet über die Wichtigkeit der Message. Mögliche Werte
in der Reihenfolge der Wichtigkeit sind:

       LOG_EMERG
       LOG_ALERT
       LOG_CRIT
       LOG_ERR
       LOG_WARNING
       LOG_NOTICE
       LOG_INFO
       LOG_DEBUG
```

Mit diesem C−interface kann jede Applikation, die in C geschrieben wurde, ins System−log schreiben. Andere Sprachen haben ähnliche APIs. Sogar Shellskripte können mit dem folgenden Befehl ins Log schreiben:

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

Eine Standard syslog Konfiguration (file /etc/syslog.conf) sollte unter anderem eine Zeile haben, die wie folgt aussieht:

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

Das "*.info" logt alles mit einem Prioritätlevel LOG_INFO oder höher. Um mehr Informationen in /var/log/messages zu erhalten, kannst "*.debug" benutzen und syslog erneut starten (/etc/init.d/syslog restart).

Die Prozedur,um eine Applikation zu "debuggen", würde also folgendermaßen aussehen:.

```
1) Laß tail -f /var/log/messages laufen und starte dann die Applikation,
die fehlerhaft ist, von einer anderen Shell aus. Vielleicht bekommst du
bereits einige Hinweise, was schief läuft.

2) Wenn Schritt 1) nicht ausreicht, dann editiere /etc/syslog.conf und
ändere *.info zu *.debug. Laß "/etc/init.d/syslog restart" laufen und
```

```
wiederhole Schritt 1).
```

Das Problem mit dieser Methode ist, daß sie vollständig davon abhängt, was der Entwickler in seinem Code gemacht hat. Wenn er keine syslog Aufrufe an Schlüsselpunkten hinzugefügt hat, dann siehst du eventuell überhaupt nichts. In anderen Worten, man kann nur Probleme finden, bei denen der Entwickler schon vorhergesehen hat, daß etwas schief gehen könnte.

strace

Eine Applikation, die unter Linux läuft, kann 3 Typen von Funktionen ausführen:

1. Funktionen, irgendwo im eigenen Code
2. Funktionen in Bibliotheken (Library−funktionen)
3. Systemaufrufe

Library−funktionen verhalten sich im Prinzip wie Funktionen der Applikation, außer daß sie in einem anderen Paket bereitgestellt werden. Systemaufrufe sind solche Funktionen, wo dein Programm mit dem Kernel spricht. Programme müssen mit dem Kernel reden, wenn sie Zugriff auf die Hardware deines Computers brauchen. Das sind: Schreiben auf den Bildschrim, Lesen einer Datei von der Festplatte, Lesen der Tastatureingabe, Senden einer Nachricht über das Netzwerk etc...

Diese Systemaufrufe können abgehört werden und du kannst so der Kommunikation zwischen der Applikation und dem Kernel folgen.

Ein häufiges Problem ist, daß eine Applikation nicht wie erwartet läuft, weil sie eine Konfigurationsdatei nicht finden kann oder nicht genügend Rechte hat, um in ein Verzeichnis zu schreiben. Diese Probleme können leicht mit strace entdeckt werden. Der relevante Systemaufruf in diesem Fall würde "open" lauten.

Man benutzt strace wie folgt:

```
strace your_application
```

Hier ist ein Beispiel:

```
# 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\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\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)
```

Was sehen wir hier? Laßt uns z.B. die folgenden Zeilen anschauen:

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

Das Programm versucht, /etc/ld.so.preload zu lesen und versagt, dann macht es weiter und liest /etc/ld.so.cache. Hier ist es erfolgreich und bekommt File−descriptor 3 zugewiesen. Nun muß das Scheitern beim Lesen von /etc/ld.so.preload nicht das Problem sein, da das Programm vielleicht nur versucht, die Datei zu lesen und zu benutzen, wenn möglich. In anderen Worten, es ist nicht unbedingt ein Problem, wenn ein Programm eine Datei nicht lesen kann. Es hängt alles vom Aufbau des Programms ab. Laßt uns alle open Aufrufe in der Ausgabe von strace anschauen:

```
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
```

Das Programm versucht nun, /usr/conf/uucp/config zu lesen. Oh! Das ist komisch, ich habe das config file in /etc/uucp/config ! ... und es gibt keine Zeile, wo das Programm versucht, /etc/uucp/config zu öffnen. Das ist der Fehler. Offentsichtlich wurde das Programm zur Kompilierzeit für den falschen Speicherplatz des Konfigurationsfiles konfiguriert.

Wie du siehst, kann strace sehr nützlich sein. Das Problem ist, daß es einige Erfahrung mit C−Programmierung erfordert, um wirklich die volle Ausgabe von strace zu verstehen, aber normalerweise mußt man nicht so weit gehen.

gdb und core files

Manchmal kommt es vor, daß ein Programm einfach unvermittelt mit der Message "Segmentation fault (core dumped)" abstürzt. Dies bedeutet, daß das Programm versucht, (aufgrund eines Programmierfehlers) über den Speicherbereich, den es allokiert hat, hinaus zu schreiben.. Besonders in Fällen, wenn das Programm nur ein paar Bytes zu viel schreibt, kann es sein, daß der Fehler nur bei dir und nur hin und wieder mal auftritt. Dies ist so, weil Speicher in Blöcken allokiert wird und manchmal ist zufällig noch etwas Platz vorhanden für die Extra−Bytes.

Wenn dieser "Segmentation fault" auftritt, bleibt ein core−file im aktuellen Arbeitsverzeichnis des Programms zurück (normalerweise ist das dein home Verzeichnis). Dieses core−file ist nur der Inhalt des Speichers zur Zeit, als der Fehler auftrat. Einige Shells verfügen über Möglichkeiten zum Steuern, ob core−files geschrieben werden. Unter bash, zum Beispiel, ist das standardmäßig eingestellte Verhalten, überhaupt keine core−files zu schreiben. Um core−files zu ermöglichen, solltest du den folgenden Befehl benutzen:

```
# ulimit −c unlimited

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

Das core−file kann jetzt mit dem gdb debugger benutzt werden, um herauszufinden, was schiefgeht. Bevor du gdb startest, kannst du überprüfen, daß du wirklich das richtige core−file anschaust:

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

OK, lshref ist ein Programm, das abgestürzt ist, laßt es uns deshalb in gdb laden. Um gdb zum Benutzen mit einem core−file aufzurufen, mußt du nicht nur das core−file spezifizieren, sondern auch den Namen der ausführbaren Datei, die zum core−file dazugehört.

```
# 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)
```

Jetzt wissen wir, daß das Programm abstürzt, während es versucht, ein strcpy zu machen. Das Problem ist, daß es eventuell viele Stellen im Code gibt, wo strcpy benutzt wird.

Allgemein gibt es jetzt 2 Möglichkeiten, um herauszufinden, wo genau der Code fehlerhaft ist.

1. Kompiliere den Code erneut mit debug Information (gcc option −g)
2. Stack trace in gdb

Das Problem in unserem Fall ist, daß strcpy eine Library Funktion ist und selbst wenn wir absolut allen Code (einschließlich libc) erneut kompilieren würden, würde es uns immer noch erzählen, daß es in einer bestimmten Zeile in der C Library scheitert.

Was wir brauchen, ist ein stack trace, das uns sagt, welche Funktion aufgerufen wurde, bevor strcpy ausgeführt wurde. Der Befehl, um so ein stack trace in gdb auszuführen, heißt "backtrace". Es läuft jedoch nicht allein mit dem core file. Du mußt das Programm in gdb noch mal laufen lassen (den Fehler reproduzieren):

```
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)
```

Jetzt können wir sehen, daß Funktion main() string_to_list() aufgerufen hat und von string_to_list wird strcpy() aufgerufen. Wir gehen zu string_to_list() und schauen uns den Code an:

```
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);
```

Diese malloc Zeile sieht wie ein Tippfehler aus. Wahrscheinlich sollte sie folgendermaßen lauten:

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

Wir ändern sie, rekompilieren und ... hurra ... es funktioniert.

Laßt uns ein zweites Beispiel anschauen, wo der Fehler nicht in einer Library entdeckt wurde, sondern im Applikationscode. In solch einem Fall kann die Applikation mit dem "gcc −g" flag kompiliert werden und gdb ist dann in der Lage, die genaue Zeile anzuzeigen, wo der Fehler entdeckt wurde.

Hier ist ein einfaches Beispiel:

```
#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);
}
```

Wir kompilieren es:

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

Lassen es laufen...

```
# ./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 sagt uns nun, daß der Fehler in Zeile 6 gefunden wurde und daß Pointer "p" auf Memory zeigt, auf die nicht zugegriffen werden kann.

Wir schauen uns den obigen Code an und es ist natürlich nur ein einfaches erfundenes Beispiel, wo p ein null pointer ist und man kann keine Daten in einem null pointer abspeichern. Leicht zu reparieren...

Schlußbemerkung

Wir haben uns Fälle angeschaut, wo man die Ursache des Fehlers wirklich finden kann, ohne zu viel über die innere Funktionsweise des Programms zu wissen.

Ich habe bewußt funktionale Fehler, z.B. "ein Knopf in einer grafischen Applikation ist an der falschen Position, aber es funktioniert", ausgeschlossen. In so einem Fall bleibt einem nichts anderes übrig, als sich mit der inneren Funktionsweise des Programms auseinanderzusetzen. Dies kostest in der Regel viel mehr Zeit und es gibt kein Rezept dafür, wie man das tun kann.

Jedoch können die einfachen Fehlerfindungstechniken, die hier gezeigt wurden, immer noch in vielen Situationen angewandt werden.

Viel Spaß beim Fehler beheben! Happy troubleshooting!
