par Guido Socher (homepage)

L´auteur:

Guido aime les possibilités d'un système à sources ouvertes comme Linux, qui offre la possibilité d'étudier en profondeur les problèmes. Vous pouvez vraiment trouver la cause d'un problème si vous vous donnez la peine d'y investir un peu de temps.

Pourquoi ça ne marche pas !? Comment repérer et corriger les erreurs dans une application Linux.

Résumé:

Si on entend ici et là qu'il est particulièrement aisé de trouver et corriger des bugs dans les programmes écrits sous Linux, il n'en est pas moins difficile de trouver de la documentation sur la façon de s'y prendre. Dans cet article, vous allez en découvrir plus sur la façon de trouver et de réparer les erreurs sans devoir apprendre quoi que ce soit à propos de la manière dont une application fonctionne.

Traduit en Français par:
Laurent Richard
<kouran(at)linuxmail.org>

Introduction

Du point de vue de l'utilisateur, il n'est pas facile de voir une différence entre un système à sources ouvertes et un système fermé tant que tout fonctionne sans problème comme prévu. Néanmoins, la situation change lorsque plus rien ne tourne et un jour ou l'autre, tout utilisateur sera face à cette situation où plus rien ne veut se passer comme prévu.

Dans un système avec des sources fermées, vous avez habituellement deux options :

- Rapporter l'erreur et payer pour la réparation
- Réinstaller et espérer que cela fonctionne ensuite

Sous Linux, vous avez ces options également mais vous pouvez aussi vous lancer dans la recherche de la cause du problème. Un des obstacles les plus importants à cela est le fait que vous n'êtes pas l'auteur du programme défaillant et que vous n'avez absolument pas la moindre idée de la façon dont il fonctionne.

Malgré ces obstacles, il y a différentes choses que vous pouvez faire sans pour cela devoir lire tout le code et sans apprendre comment le programme fonctionne.

Logs

La chose la plus évidente et la plus simple que vous puissiez faire est de regarder dans les fichiers qui se trouvent dans /var/log/... Ce que vous allez trouvez dans ces fichiers et le nom de ceux−ci est entièrement configurable. /var/log/messages est généralement le fichier sur lequel vous devriez vous pencher. Les grosses applications peuvent avoir des répertoires spécifiques pour leurs logs (/var/log/httpd/ /var/log/exim ...).
La plupart des distributions utilisent syslog comme system logger et son comportement est contrôlé par le fichier de configuration /etc/syslog.conf La syntaxe de ce fichier est commentée dans "man syslog.conf".

Le logging fonctionne comme si le créateur d'un programme pouvait ajouter une ligne syslog à son code. C'est comme un printf sauf que cela écrit dans le log du système. Dans ce rapport, vous spécifiez une priorité et un "facility" pour classifier le message :

```
       #include <syslog.h>

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

```
facility classe le type d'application envoyant le message.
priority détermine l'importance du message. Les valeurs possibles dans leur ordre d'importance so
```

```
       LOG_EMERG
       LOG_ALERT
       LOG_CRIT
       LOG_ERR
       LOG_WARNING
       LOG_NOTICE
       LOG_INFO
       LOG_DEBUG
```

Avec cette interface en C, toute application écrite en C peut écrire sur le log du système. Les autres langages ont des APIs similaires. Même les scripts shell peuvent écrire dans le log avec la commande :

```
logger -p err "ce texte va vers /var/log/messages"
```

Une configuration standard de syslog (fichier /etc/syslog.conf) devrait avoir parmi les autres une ligne ressemblant à celle−ci :

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

Le "*.info" va logger tout ce qui a un niveau de priorité LOG_INFO ou supérieur. Pour avoir plus de renseignements dans /var/log/messages, vous pouvez changer cela en "*.debug" et relancer syslog (/etc/init.d/syslog restart).

La procédure pour "debuger" une application serait donc comme suit.

1. lancer tail −f /var/log/messages et ensuite lancer l'application défaillante dans un autre shell. Peut−être aurez vous quelques pistes pour savoir ce qui ne va pas.
2. Si la première étape est insuffisante, éditez /etc/syslog.conf et changer *.info en *.debug. Lancez "/etc/init.d/syslog restart" et répéter l'étape 1).

Le problème de cette méthode est qu'elle dépend entièrement de ce que le développeur a mis dans son code. Si il/elle n'a pas ajouté de rapport syslog aux points cruciaux, vous ne verrez alors rien du tout. En d'autres mots, vous ne serez en mesure de voir que les problèmes que le développeur avait présentis.

strace

Une application sous Linux peur exécuter 3 types de fonctions :

1. les fonctions se trouvant quelque part dans son propre code
2. les fonctions de bibliothèques
3. les appels système

Les fonctions de bibliothèques sont similaires aux fonctions propres des applications sauf que celles–ci sont fournies dans un autre paquetage. Les appels système sont les fonctions où le programme communique avec le noyau. Les programmes ont besoin de communiquer avec le noyau si ils ont besoin d'accéder aux matériel de l'ordinateur. C'est le cas pour l'affichage à l'écran, la lecture d'un fichier sur un disque, la lecture de l'encodage au clavier, l'envoi d'un message sur le réseau, etc...

Ces appels système peuvent être interceptés et vous pouvez dès lors suivre la communication entre l'application et le noyau.

Un problème courant est celui où une application ne fonctionne pas comme prévu car elle ne peut pas trouver un fichier de configuration ou n'a pas les permissions suffisantes pour écrire dans un répertoire. Ces problèmes peuvent facilement être détectés avec strace. L'appel système relevant s'appelle dans ce cas–ci "open".

Vous utilisez strace comme ceci :

```
strace votre_application
```

Voici un exemple :

```
# 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)
```

Que voyez−vous ? Attardons notre regard par exemple sur les lignes suivantes :

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

Le programme essaye de lire /etc/ld.so.preload et échoue. Ensuite, il continue et lit /etc/ld.so.cache. Ici, il réussit et obtient un file descriptor 3. Maintenant, l'erreur de lecture de /etc/ld.so.preload peut ne pas être un problème du tout car le programme peut juste essayer de lire ceci et l'utiliser si possible. En d'autre mots, ce n'est pas obligatoirement un problème si un programme échoue lors de la lecture d'un fichier. Tout dépend de la façon dont le programme a été élaboré. Jetons un coup d'oeil à tous les appels open dans les résultats de 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
```

Le programme essaye maintenant de lire /usr/conf/uucp/config. Oh! C'est étrange, j'ai le fichier de configuration dans /etc/uucp/config ! ... et donc il n'y a pas de ligne où le programme essaye d'ouvrir /etc/uucp/config. Voilà l'erreur. Le programme a donc manifestement été configuré au moment de la compilation avec une localisation erronée du fichier de configuration.

Comme vous pouvez voir, strace peut être très utile. Le problème est que cela nécessite un peu d'expérience de programmation en C pour comprendre l'ensemble des résultats de strace mais normalement, vous ne devriez pas avoir besoin d'en arriver là.

gdb et les fichiers core

Parfois, il arrive qu'un programme se termine avec ce message empreint de dépit : "Segmentation fault (core dumped)". Celà signifie que le programme a essayé (suite à une erreur de programmation) d'écrire au delà de la zone de mémoire qui lui était allouée. C'est la cas spécialement quand un programme écrit ne serait–ce que quelques octets de trop, ce quii peut survenir en de rares occasions. C'est dû à la mémoire qui est allouée en morceaux et il arrive parfois qu'accidentellement, il n'y ait plus de place suffisante pour les octets supplémentaires.

Lorsque ce "Segmentation fault" apparaît, un fichier core est créé dans le répertoire de travail actuel du programme (normalement, votre répertoire personnel). Ce fichier core est une copie du contenu de la mémoire au moment où l'erreur est survenue. Certains shells fournissent des possibilités de contrôler lorsque les fichiers core sont écrits. Sous bash, par exemple, le comportement par défaut est de ne pas écrire du tout de fichier core. Pour activer les fichiers core, vous devez utiliser cette commande :

```
# ulimit –c unlimited

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

Le fichier core peut maintenant être utilisé avec le débugeur gdb pour trouver ce qui n'a pas fonctionné. Avant de démarrer gdb, vous pouvez vérifier que vous cherchez bien dans le bon fichier core :

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

OK, lshref est le programme qui a crashé, donc chargeons le dans gdb. Pour appeler gdb en vue de l'utiliser avec le fichier core, vous ne devez pas uniquement spécifier le nom du fichier core mais aussi le nom de l'exécutable qui va avec le fichier core.

```
# 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)
```

Ainsi, nous voyons que le programme crashe lorsqu'il essaye de faire un strcpy. Le problème est que cela peut se situer à beaucoup d'endroits dans le code, partout où strcpy est utilisé.

En général, il y a maintenant 2 possibilités pour trouver exactement dans quelle partie du code cela ne fonctionne plus.

1. Recompiler le code avec l'information pour le débugage (gcc option –g)
2. Réaliser une analyse de la pile dans gdb

Le problème dans notre cas est que strcpy est une fonction de bibliothèque et que même si nous recompilions absolument tout le code (en incluant le libc) cela nous dirait toujours que cela échoue à une certaine ligne de la bibliothèque C.

Ce dont nous avons besoin est une analyse de la pile trace qui va nous dire quelle fonction a été appelée avant que strcpy ne soit exécuté. La commande pour réaliser une analyse de la pile dans gdb est "backtrace". Cela ne va cependant pas fonctionner uniquement avec le fichier core. Nous devons relancer la commande dans gdb (reproduire l'erreur) :

```
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)
```

Maintenant, nous pouvons voir que la fonction main() a appelé string_to_list() et de string_to_list strcpy() est appelé. Nous allons nous tourner vers string_to_list() et regarder le 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);
```

Cette ligne malloc semble contenir une faute de positionnement typographique. Probablement, cela aurait dû être :

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

Nous modifions cela, recompilons et ... hourra ... cela fonctionne.

Regardons un deuxième exemple où l'erreur n'est pas détectée dans une librairie mais dans le code de l'application. Dans un tel cas, l'application peut être compilée avec le drapeau "gcc –g" et gdb sera capable de montrer la ligne exacte où l'erreur a été détectée.

Voici un exemple simple.

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

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);
}
```

Nous le compilons :

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

Lancons le ...

```
# ./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 nous dit que l'erreur a été détectée à la ligne 6 et que le pointeur "p" pointe vers une mémoire qui ne peut pas être atteinte.

Nous regardons le code ci−dessus et c'est bien−sûr un bête exemple où p est un pointeur nul et vous ne pouvez pas conserver une donnée dans un pointeur nul. Facile à réparer ...

Conclusion

Nous avons vu des cas où vous pouvez vraiment trouver la cause de l'erreur sans connaître grand chose sur les méandres du fonctionnement d'un programme.

Je n'ai pas abordé le cas des erreurs fonctionnelles, par ex. un bouton dans une GUI qui est dans une mauvaise position mais qui fonctionne quand même. Dans ces cas, vous devrez en apprendre plus sur le fonctionnement interne du programme. Cela prend généralement plus de temps et donc, il n'y a pas de recette préfabriquée sur la méthode à appliquer.

Cependant, les techniques simples pour trouver les erreurs peuvent être quand même appliquées dans la plupart des cas.

Bonne chasse !
