Example of using gdb and strace to find the cause of a segmentation fault

Adrien Kunysz, Sun, 06 Mar 2011 18:17:12

This article describes how I diagnosed a segmentation fault in apt/aptitude. It shows some basic usage of gdb and strace. We look at a bit of C code (C++ really) and x86 assembly and we show that switching between tools may make analysis faster than when you just stick to one.

The problem

A few weeks ago I was somehow logged into an Ubuntu system as root and while I was doing what I was supposed to do, I noticed aptitude was not completing as expected. It seems everything worked fine but it always ended up with a segmentation fault:

```
# aptitude install strace
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
No packages will be installed, upgraded, or removed.
0 packages upgraded, 0 newly installed, 0 to remove and 17 not upgraded.
Need to get 0B of archives. After unpacking 0B will be used.
Writing extended state information... Done
Segmentation fault
```

Notice how it basically did nothing (because there was nothing to do) then crashed. A segmentation fault generally means the process attempted to access memory it shouldn't have access to.

gdb

Naturally I reached for ulimit and tried again, installing the debug symbols at the same time:

```
# ulimit -c unlimited
# aptitude install aptitude-dbg
[...]
The following NEW packages will be installed:
  aptitude-dbg libcwidget3-dbg{a}
0 packages upgraded, 2 newly installed, 0 to remove and 17 not upgraded.
Need to get 7,668kB of archives. After unpacking 25.7MB will be used.
Do you want to continue? [Y/n/?] y
[...]
Selecting previously deselected package aptitude-dbg.
(Reading database ... 41701 files and directories currently installed.)
Unpacking aptitude-dbg (from .../aptitude-dbg_0.4.11.11-1ubuntu10_amd64.deb) ...
Selecting previously deselected package libcwidget3-dbg.
Unpacking libcwidget3-dbg (from .../libcwidget3-dbg_0.5.13-1ubuntu1_amd64.deb) ...
Setting up aptitude-dbg (0.4.11.11-1ubuntu10) ...
Setting up libcwidget3-dbg (0.5.13-1ubuntu1) ...
Segmentation fault (core dumped)
```

Well, at least it can install stuff. The `ulimit` bash built-in allows you to change some system limits. By default on that system, the maximum core size was set to 0. By running `ulimit -c unlimited`, we told the system to dump a core whenever a process encounters a segmentation fault. The core is essentially an image of the memory of the process at the time of the problem.

However, to examine a core we generally need debug symbols for the faulty program. This is what is contained in the package `aptitude-dbg` we just installed.

Let's have a look at that core:

```
# gdb `which aptitude` core
Reading symbols from /usr/bin/aptitude...Reading symbols from /usr/lib/debug/usr/bin/aptitude...done.
done.
[New Thread 1690]
[New Thread 1691]
[...]
Core was generated by `aptitude install aptitude-dbg'.
Program terminated with signal 11, Segmentation fault.
#0  0x00007fab30590a1f in __fprintf_chk () from /lib/libc.so.6
(gdb) bt
#0  0x00007fab30590a1f in __fprintf_chk () from /lib/libc.so.6
#1  0x00007fab3216946d in pkgDPkgPM::CloseLog() () from /usr/lib/libapt-pkg-libc6.10-6.so.4.8
#2  0x00007fab3216fef0 in pkgDPkgPM::Go(int) () from /usr/lib/libapt-pkg-libc6.10-6.so.4.8
#3  0x00007fab32120a85 in pkgPackageManager::DoInstallPostFork(int) ()
   from /usr/lib/libapt-pkg-libc6.10-6.so.4.8
#4  0x0000000000542e7a in download_install_manager::execute_install_run (this=0x7fff36beb760,
    res=<value optimized out>, progress=<value optimized out>) at download_install_manager.cc:149
#5  0x0000000000543347 in download_install_manager::finish (this=0x0, res=pkgAcquire::Failed,
    progress=...) at download_install_manager.cc:190
#6  0x0000000000507c67 in cmdline_do_download (m=0x7fff36beb760, verbose=<value optimized out>)
    at cmdline_util.cc:404
#7  0x00000000004dd686 in cmdline_do_action (argc=<value optimized out>, argv=0xffffffff,
    status_fname=0x7fff36bebb7d "\177", simulate=<value optimized out>,
    assume_yes=<value optimized out>, download_only=<value optimized out>, fix_broken=false,
    showvers=false, showdeps=<value optimized out>, showsize=<value optimized out>,
    showwhy=<value optimized out>, visual_preview=false, always_prompt=<value optimized out>,
    safe_resolver=false, no_new_installs=false, no_new_upgrades=false, user_tags=...,
    arch_only=<value optimized out>, queue_only=false, verbose=0) at cmdline_do_action.cc:313
#8  0x000000000041bd39 in main (argc=3, argv=0x7fff36bec278) at main.cc:641
```

So we are really segfaulting in a variant of `fprintf()` in the glibc as called from `pkgDPkgPM::CloseLog()` (looks like C++) in libapt-pkg. The debug symbols we installed are pretty much useless for this as they only cover aptitude and not libapt-pkg or the glibc (notice how we have more detailed information from frame #4 while we are missing lot of things above that).

It seems there is no apt-dbg package. This means we should rebuild apt with the debug symbols to get a nice complete backtrace. I am far too lazy to do that. Maybe we can figure it out just by looking at the code. A bug in apt seems more likely than in the glibc so let's look at `pkgDPkgPM::CloseLog()`:

```
$ apt-get source apt
[...]
dpkg-source: info: extracting apt in apt-0.7.25.3ubuntu7
dpkg-source: info: unpacking apt_0.7.25.3ubuntu7.tar.gz
$ grep -lR CloseLog apt-0.7.25.3ubuntu7/
apt-0.7.25.3ubuntu7/apt-pkg/deb/dpkgpm.cc
apt-0.7.25.3ubuntu7/apt-pkg/deb/dpkgpm.h
grep: apt-0.7.25.3ubuntu7/buildlib/config.sub: No such file or directory
grep: apt-0.7.25.3ubuntu7/buildlib/config.guess: No such file or directory
$ vi apt-0.7.25.3ubuntu7/apt-pkg/deb/dpkgpm.cc
 640 bool pkgDPkgPM::CloseLog()
 641 {
 642    char timestr[200];
 643    time_t t = time(NULL);
 644    struct tm *tmp = localtime(&t);
 645    strftime(timestr, sizeof(timestr), "%F  %T", tmp);
 646
 647    if(term_out)
 648    {
 649       fprintf(term_out, "Log ended: ");
 650       fprintf(term_out, "%s", timestr);
 651       fprintf(term_out, "\n");
 652       fclose(term_out);
 653    }
 654    term_out = NULL;
 655
 656    string history_name = flCombine(_config->FindDir("Dir::Log"),
 657                                    _config->Find("Dir::Log::History"));
 658    if (!history_name.empty())
 659    {
 660       FILE *history_out = fopen(history_name.c_str(),"a");
 661       fprintf(history_out, "End-Date: %s\n", timestr);
 662       fclose(history_out);
 663    }
 664
 665    return true;
 666 }
```

Pretty short. Good. However there are four `fprintf()` in there. How can we figure out the one that is causing problem? From the backtrace we know the return address for that `fprintf()` is 0x00007fab3216946d, let's see what we get at assembly level:

```
(gdb) disass 0x00007fab3216946d
Dump of assembler code for function _ZN9pkgDPkgPM8CloseLogEv:
   0x00007fab32169300 <+0>:     push   %r13
   0x00007fab32169302 <+2>:     push   %r12
   0x00007fab32169304 <+4>:     push   %rbp
   0x00007fab32169305 <+5>:     mov    %rdi,%rbp
   0x00007fab32169308 <+8>:     xor    %edi,%edi
   0x00007fab3216930a <+10>:    push   %rbx
   0x00007fab3216930b <+11>:    sub    $0x118,%rsp
   0x00007fab32169312 <+18>:    mov    %fs:0x28,%rax
   0x00007fab3216931b <+27>:    mov    %rax,0x108(%rsp)
   0x00007fab32169323 <+35>:    xor    %eax,%eax
   0x00007fab32169325 <+37>:    callq  0x7fab320f8088 <time@plt>
   0x00007fab3216932a <+42>:    lea    0x30(%rsp),%rdi
   0x00007fab3216932f <+47>:    lea    0x40(%rsp),%rbx
   0x00007fab32169334 <+52>:    mov    %rax,0x30(%rsp)
   0x00007fab32169339 <+57>:    callq  0x7fab320f7f28 <localtime@plt>
   0x00007fab3216933e <+62>:    lea    0x1a90e(%rip),%rdx        # 0x7fab32183c53
   0x00007fab32169345 <+69>:    mov    %rbx,%rdi
   0x00007fab32169348 <+72>:    mov    %rax,%rcx
   0x00007fab3216934b <+75>:    mov    $0xc8,%esi
   0x00007fab32169350 <+80>:    callq  0x7fab320f7f88 <strftime@plt>
   0x00007fab32169355 <+85>:    mov    0x430(%rbp),%rdi
   0x00007fab3216935c <+92>:    test   %rdi,%rdi
   0x00007fab3216935f <+95>:    je     0x7fab321693b7 <_ZN9pkgDPkgPM8CloseLogEv+183>
   0x00007fab32169361 <+97>:    lea    0x1a8f2(%rip),%rdx        # 0x7fab32183c5a
   0x00007fab32169368 <+104>:   mov    $0x1,%esi
   0x00007fab3216936d <+109>:   xor    %eax,%eax
   0x00007fab3216936f <+111>:   callq  0x7fab320f7408 <__fprintf_chk@plt>
   0x00007fab32169374 <+116>:   mov    0x430(%rbp),%rdi
   0x00007fab3216937b <+123>:   lea    0x15065(%rip),%rdx        # 0x7fab3217e3e7
   0x00007fab32169382 <+130>:   mov    %rbx,%rcx
   0x00007fab32169385 <+133>:   mov    $0x1,%esi
   0x00007fab3216938a <+138>:   xor    %eax,%eax
   0x00007fab3216938c <+140>:   callq  0x7fab320f7408 <__fprintf_chk@plt>
   0x00007fab32169391 <+145>:   mov    0x430(%rbp),%rdi
   0x00007fab32169398 <+152>:   lea    0x1561e(%rip),%rdx        # 0x7fab3217e9bd
   0x00007fab3216939f <+159>:   mov    $0x1,%esi
   0x00007fab321693a4 <+164>:   xor    %eax,%eax
   0x00007fab321693a6 <+166>:   callq  0x7fab320f7408 <__fprintf_chk@plt>
   0x00007fab321693ab <+171>:   mov    0x430(%rbp),%rdi
   0x00007fab321693b2 <+178>:   callq  0x7fab320f7d18 <fclose@plt>
   0x00007fab321693b7 <+183>:   mov    0x23299a(%rip),%r13        # 0x7fab3239bd58
   0x00007fab321693be <+190>:   lea    0x181d6(%rip),%rdx        # 0x7fab3218159b
   0x00007fab321693c5 <+197>:   movq   $0x0,0x430(%rbp)
   0x00007fab321693d0 <+208>:   xor    %ecx,%ecx
   0x00007fab321693d2 <+210>:   mov    %rsp,%rdi
   0x00007fab321693d5 <+213>:   lea    0x10(%rsp),%rbp
   0x00007fab321693da <+218>:   mov    0x0(%r13),%rsi
   0x00007fab321693de <+222>:   callq  0x7fab320fe3f0 <_ZNK13Configuration4FindEPKcS1_>
   0x00007fab321693e3 <+227>:   mov    0x0(%r13),%rsi
   0x00007fab321693e7 <+231>:   lea    0x1817c(%rip),%rdx        # 0x7fab3218156a
   0x00007fab321693ee <+238>:   xor    %ecx,%ecx
   0x00007fab321693f0 <+240>:   mov    %rbp,%rdi
   0x00007fab321693f3 <+243>:   callq  0x7fab320fee50 <_ZNK13Configuration7FindDirEPKcS1_>
   0x00007fab321693f8 <+248>:   lea    0x20(%rsp),%r13
   0x00007fab321693fd <+253>:   mov    %rsp,%rdx
   0x00007fab32169400 <+256>:   mov    %rbp,%rsi
   0x00007fab32169403 <+259>:   mov    %r13,%rdi
   0x00007fab32169406 <+262>:   callq  0x7fab3210d530 <_Z9flCombineSsSs>
   0x00007fab3216940b <+267>:   mov    0x10(%rsp),%rdi
   0x00007fab32169410 <+272>:   mov    0x232b09(%rip),%rbp        # 0x7fab3239bf20
   0x00007fab32169417 <+279>:   sub    $0x18,%rdi
   0x00007fab3216941b <+283>:   cmp    %rbp,%rdi
   0x00007fab3216941e <+286>:   jne    0x7fab32169504 <_ZN9pkgDPkgPM8CloseLogEv+516>
   0x00007fab32169424 <+292>:   mov    (%rsp),%rdi
   0x00007fab32169428 <+296>:   sub    $0x18,%rdi
   0x00007fab3216942c <+300>:   cmp    %rdi,%rbp
   0x00007fab3216942f <+303>:   jne    0x7fab321694d7 <_ZN9pkgDPkgPM8CloseLogEv+471>
   0x00007fab32169435 <+309>:   mov    0x20(%rsp),%rdi
   0x00007fab3216943a <+314>:   cmpq   $0x0,-0x18(%rdi)
   0x00007fab3216943f <+319>:   lea    -0x18(%rdi),%rax
   0x00007fab32169443 <+323>:   je     0x7fab3216947e <_ZN9pkgDPkgPM8CloseLogEv+382>
   0x00007fab32169445 <+325>:   lea    0x17a45(%rip),%rsi        # 0x7fab32180e91
   0x00007fab3216944c <+332>:   callq  0x7fab320f7588 <fopen@plt>
   0x00007fab32169451 <+337>:   lea    0x1a80e(%rip),%rdx        # 0x7fab32183c66
   0x00007fab32169458 <+344>:   mov    %rax,%r12
   0x00007fab3216945b <+347>:   mov    %rax,%rdi
   0x00007fab3216945e <+350>:   mov    %rbx,%rcx
   0x00007fab32169461 <+353>:   mov    $0x1,%esi
   0x00007fab32169466 <+358>:   xor    %eax,%eax
   0x00007fab32169468 <+360>:   callq  0x7fab320f7408 <__fprintf_chk@plt>
   0x00007fab3216946d <+365>:   mov    %r12,%rdi  <= address at which the faulty call to fprintf() would have returned
   0x00007fab32169470 <+368>:   callq  0x7fab320f7d18 <fclose@plt>
[...]
```

This is what `pkgDPkgPM::CloseLog()` looks like in assembly. As an aside, notice how it is renamed `_ZN9pkgDPkgPM8CloseLogEv()` after compilation. This is due to [name decoration (or mangling really)].

A quick inspection of the assembly dump shows the four calls to `fprintf()` at 0x00007fab3216936f, 0x00007fab3216938c, 0x00007fab321693a6 and 0x00007fab32169468. Since the return address for the faulty `fprintf()` is 0x00007fab3216946d which is right after that fourth call, the problem is with the `fprintf()` at 0x00007fab32169468. This doesn't necessarily mean it's the last `fprintf()` in the C as code may be rearranged and inlined by the compiler. Still, seeing how the first three are grouped and the fourth one is all by itself immediately between a `fopen()` and a `fclose()`, this looks a lot like line 661 in the C:

```
660       FILE *history_out = fopen(history_name.c_str(),"a");
661       fprintf(history_out, "End-Date: %s\n", timestr);
662       fclose(history_out);
```

Indeed, if `fopen()` fails and returns NULL, we are going to pass that NULL pointer to `fprintf()` for its first argument. That would certainly explain a segmentation fault with such a backtrace.

So, what did `fopen()` try to open? This information is somewhere in `history_name` and returned by its `c_str()` method. While we *could* figure out where that object resides in memory by looking at the assembly then find the offset at which we'll find the pointer to the string containing the name of the file we are interested in, this is far more work than I am willing to do at this point (and if it goes down to that I would rather just rebuild apt with the debug symbols).

Switching to strace

Let's just strace the failure, only looking at the `open()` system call since that's the only thing we are interested in:

```
# strace -e open -f aptitude install aptitude-dbg
[...]
[pid  6079] open("/var/log/apt/history.log", O_WRONLY|O_CREAT|O_APPEND, 0666) = -1 ENOENT (No such file or directory)
[pid  6079] --- SIGSEGV (Segmentation fault) @ 0 (0) ---
[pid  6081] +++ killed by SIGSEGV (core dumped) +++
```

Strace is just a way to observe all the system calls the process is performing. The point is that it also retrieves the arguments of the system calls and that's what we are interested in.

Look how it segfaulted immediately after failing to open `/var/log/apt/history.log`. It failed with `ENOENT (No such file or directory)`. This may seem strange as `O_CREAT` was specified. This means the file should have been created if it doesn't exist. However that will still fail if one of the directories leading to the file doesn't exist (refer to the manual pages for `open(2)` for the details). Let's see:

```
# ls -d /var /var/log /var/log/apt /var/log/apt/history.log
ls: cannot access /var/log/apt: No such file or directory
ls: cannot access /var/log/apt/history.log: No such file or directory
/var  /var/log
```

Yep, we are missing `/var/log/apt/`. Let's create it and see if it fixes the problem:

```
# mkdir /var/log/apt
# aptitude install aptitude-dbg
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
No packages will be installed, upgraded, or removed.
0 packages upgraded, 0 newly installed, 0 to remove and 17 not upgraded.
Need to get 0B of archives. After unpacking 0B will be used.
Writing extended state information... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
```

Indeed, no more segmentation fault.

At this point you may be wondering why we didn't start with strace in the first place. The problem is that a simple strace yields a lot of output and if you don't know what you are looking for, finding the information in the noise may not be easy. Segmentation faults may have many different causes. For some of them strace won't tell us anything useful at all. By the time we switched to strace we had a good idea what we were looking for (the path of the file that fails to `open()`).

The bug

Still, aptitude/apt should not segfault because it's missing a directory. It should display a nice, informative error message and possibly exit gracefully but a segmentation fault is not the right way to do this. Now, googling around we find this is addressed by Ubuntu bug 535509 which is fixed in apt 0.7.25.3ubuntu9.1. The way it was fixed looks like this:

```
--- apt-0.7.25.3ubuntu7/apt-pkg/deb/dpkgpm.cc   2010-04-14 19:30:06.000000000 +0100
+++ apt-0.7.25.3ubuntu9.3/apt-pkg/deb/dpkgpm.cc 2010-09-09 18:31:30.000000000 +0100
@@ -650,12 +650,17 @@ bool pkgDPkgPM::CloseLog()
       fprintf(term_out, "%s", timestr);
       fprintf(term_out, "\n");
       fclose(term_out);
    }
    term_out = NULL;

+   // check if the directory exists in which we want to write the file
+   string const logdir = _config->FindDir("Dir::Log");
+   if(not FileExists(logdir))
+      return _error->Error(_("Directory '%s' missing"), logdir.c_str());
+
    string history_name = flCombine(_config->FindDir("Dir::Log"),
                                    _config->Find("Dir::Log::History"));
    if (!history_name.empty())
    {
       FILE *history_out = fopen(history_name.c_str(),"a");
       fprintf(history_out, "End-Date: %s\n", timestr);
```

We simply check for the directory existence before attempting to open it. Notice we may still encounter the same problem if the directory disappears between the check and the attempt to open but I guess the maintainer finds it acceptable to crash in such an unlikely case.

Updated Mon, 07 Mar 2011 08:25:42

As noticed by wildcat, it will still segfault if the directory exists but is not writable (`chmod a-rwx` for example). A better fix has been implemented later. The current code looks like this:

```
bool pkgDPkgPM::CloseLog()
{
   char timestr[200];
   time_t t = time(NULL);
   struct tm *tmp = localtime(&t);
   strftime(timestr, sizeof(timestr), "%F  %T", tmp);

   if(term_out)
   {
      fprintf(term_out, "Log ended: ");
      fprintf(term_out, "%s", timestr);
      fprintf(term_out, "\n");
      fclose(term_out);
   }
   term_out = NULL;

   if(history_out)
   {
...
      if (dpkg_error.empty() == false)
         fprintf(history_out, "Error: %s\n", dpkg_error.c_str());
      fprintf(history_out, "End-Date: %s\n", timestr);
      fclose(history_out);
   }
   history_out = NULL;

   return true;
}
```

The opening of the file is handled "somewhere else" and we just never try to use the file descriptor if it's NULL.

The real WTF

As for the reason why `/var/log/apt` was missing on that system, it was due to an operational mistake in attempting to free space on that filesystem. Additionally there is still a problem in the way the system is managed as the 6 months old fix was not applied but that goes out of the scope of this article.

Back to all articles.