Profiling and Debugging Lab

Aaron Birkland
Cornell Center for Advanced Computing

June 18, 2013

1 GDB debugging

This lab exercise serves as an introduction to debugging via GDB (The GNU Debugger). While one may normally wish to debug within an IDE using a comfortable GUI, GDB and its command-line interface is lightweight, powerful, installed virtually everywhere, and usable with little fuss. It is a useful "least common denominator" to know.

This lab will focus around a poorly program "scramble" containing several bugs. This program is supposed to accept a user-provided text string and print a scrambled representation of this string to `STDOUT`.

1.1 Setup

To begin, we will unpack the lab materials and compile the example program.

1. Unpack the lab materials into your home directory if you haven't done so already.

```
$ cd
$ tar xvf ~tg459572/LABS/profile_debug.tar
$ cd profile_debug
```

2. Compile the scramble program. We are intentionally starting off *without* specifying debug symbols.

```
$ gcc scramble.c -o scramble
```

3. Run the scramble program with some text to scramble as an argument. It should crash with a segmentation fault.

```
$ ./scramble "scramble me"
Segmentation fault (core dumped)
```

1.2 Analyzing core dumps

When a program crashes unexpectedly, the OS can dump a copy of its current memory state into a core file. This file can be analyzed later with GDB. Typically, a user can set a size limit for core dumps. This is useful to prevent serious disk usage mishaps for programs that use large amounts of memory. On Stampede, the default is 0, i.e. it will not dump a core grater than zero bytes large unless you direct otherwise.

1. In the bash shell (defailt on Stampede), use `ulimit -a` to see defailt values. (If you switched to C shell, use `limit` instead)

   ```
   $  ulimit -a
   core file size          (blocks, -c) 0
   data seg size           (kbytes, -d) unlimited
   scheduling priority             (-e) 0
   file size               (blocks, -f) unlimited
   pending signals                 (-i) 514620
   max locked memory       (kbytes, -l) 64
   max memory size         (kbytes, -m) unlimited
   open files                      (-n) 1024
   pipe size            (512 bytes, -p) 8
   POSIX message queues     (bytes, -q) 819200
   real-time priority              (-r) 0
   stack size              (kbytes, -s) unlimited
   cpu time               (seconds, -t) unlimited
   max user processes              (-u) 150
   virtual memory          (kbytes, -v) 8388608
   file locks                      (-x) unlimited
   ```

   As you can see, the default is 0.

2. Change the core dump size to unlimited. (on C shell use `limit coredumpsize unlimited`)

   ```
   $ ulimit -c unlimited
   ```

3. Run the scramble program again and look for the dump file. Its name should be something like `core.PID` where `PID` is the process ID number. For example, `core.11781`.

   ```
   $ ./scramble "scramble me"
     Segmentation fault (core dumped)
   ```

4. Run GDB using the executable and core file as arguments. This will tell the debugger to analyze the given memory image created by the given executable.

```
$ gdb scramble core.29016
```

You will see some text flash by saying how the program was invoked and how it crashed (Segmentation fault), ending up at a gdb prompt `(gdb)`. Note the various "no debugging symbols found" messages.

```
  Reading symbols from /home1/01871/apb18/profile_debug/scramble...
  (no debugging symbols found)...done.
[New Thread 29016]
Reading symbols from /lib64/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...
(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Reading symbols from /lib/modules/2.6.32-279.14.1.el6.x86_64/vdso/vdso.so...
Reading symbols from
/usr/lib/debug/lib/modules/2.6.32-279.14.1.el6.x86_64/vdso/vdso.so.debug...done.
Loaded symbols for /lib/modules/2.6.32-279.14.1.el6.x86_64/vdso/vdso.so
Core was generated by `./scramble scramble me'.
Program terminated with signal 11, Segmentation fault.
#0  0x00000000004005b1 in scramble ()
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6_3.6.x86_64
  (gdb)
```

5. To figure out *where* the program crashed, print out a stack backtrace. At the `(gdb)` prompt, type in `bt` to print a stack backtrace.

```
(gdb) bt
#0  0x000000000040055e in scramble ()
#1  0x00000000004005b6 in main ()
```

As you can see, the output is somewhat helpful. We can see the memory addresses of our stack frames, as well as the name of the functions they represent. So we know that our program crashed somewhere in `scramble()`, but not much else. Look at the code. Intuitively, `strlen()` could be a problem (is the string null terminated?), as could array bounds or pointers. We don't have enough information to tell.

6. Try to print out a variable. Unfortunately, this does not work. Our problems stem from the fact that we forgot to compile with debugging symbols. We will correct this in the next exercise.

```
(gdb) print i
No symbol "i" in current context.
```

7. Exit GDB by typing in `q` at the prompt.

```
(gdb) q
```

1.3 Debugging symbols

When we compile with debugging symbols enabled, the debugger becomes much more useful, as it can correlate our source code with functions and variables present in memory.

1. Compile the program with debugging symbols and no optimization. Aggressive optimization will break the correlation between between the source code and the native machine instructions, complicating the debugging process.

```
$ gcc -g -O0 scramble.c -o scramble
```

2. Run the program, allow it to dump core when it segfaults, and load the *new* core file into gdb as before.

```
$ ./scramble "scramble me"
Segmentation fault (core dumped)
$ gdb  scramble core.28616
Reading symbols from /home1/01871/apb18/profile_debug/scramble...done.
[New Thread 30715]
Reading symbols from /lib64/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Reading symbols from /lib/modules/2.6.32-279.14.1.el6.x86_64/vdso/vdso.so...
Reading symbols from /usr/lib/debug/lib/modules/2.6.32-279.14.1.el6.x86_64/vdso/vdso.so.debu
done.
Loaded symbols for /lib/modules/2.6.32-279.14.1.el6.x86_64/vdso/vdso.so
Core was generated by `./scramble scramble me'.
Program terminated with signal 11, Segmentation fault.
#0  0x00000000004005b1 in scramble (message=0x7fff6e6e6e67 "./scramble",
    buffer=0x400728 "") at scramble.c:9
9           buffer[i] = ((message[i] + i) % 93) + 33;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6_3.6.x86_64
(gdb)
```

Much better! Even without doing a back trace, we see exactly where the crash occurred, line 9 of `scramble.c`

3. Print out some variables to help us figure out what is going on at line 9. By inspecting `i`, `message`, and `buffer`, can you figure out a possible cause of the crash?

```
(gdb) print i
$1 = 0
(gdb) print buffer
$2 = 0x400728 ""
(gdb) print message
$3 = 0x7fff6e6e6e67 "./scramble"
```

4. We see that `message` has a valid string, and `i` has not been incremented yet, so it is likely that something is wrong with writing to `buffer`. As it turns out, we mistakenly initiated it with an unmodifiable string literal on line 16. Change line 16 from `char *buffer = "";` to `char buffer[16];` (not an ideal fix, as it opens the door to different kinds of bugs, but it will fix our segfault). Name the fixed executable `scramble_fixed`. Compile and run.

   ```
   $ gcc -g -O0 scramble_fixed.c -o scramble_fixed
   $ ./scramble_fixed "scramble me"
   OQ9*:*7-82-59I77
   ```

   It worked! we fixed our bug.