Debugging under Linux – guided gdb tour

Jan Otte

2007

Where oh where is my main() frame?

```
(gdb) break free
```

Contents

1 Several introduction words (could be skipped) 2

2 License 2

3 Preparations 2
3.0.1 Now really the instructions . . . 2

4 Failure number 0 3
4.0.2 First run – learning to read gdb output . . . 3
4.0.3 Backtrace . . . 5
4.0.4 Debugging information – how to get them in . . . 5
4.0.5 Incorrect and incomplete backtrace . . . 5
4.0.6 Basic commands . . . 5
4.0.7 Remember commands . . . 6

5 Failure number 1 7
5.0.8 Live debug . . . 7
5.0.9 Breakpoints . . . 7
5.0.10 Variable value mangling . . . 9
5.0.11 What you should remember . . . 10

6 Failure number 2 10
6.0.12 Calling functions from debugger . . . 11
6.0.13 What you should remember . . . 11

7 Failure number 3 12
7.0.14 Evaluating macros . . . 12
7.0.15 Things to remember . . . 13

8 Failure number 4 13
8.0.16 Overwritten stack . . . 13
8.0.17 Little bit of magic – assigning commands to breakpoints . . . 14
8.0.18 Important things to remember . . . 17

9 Failure number 5 17

10 Where from now? 18
10.0.19 External links . . . 18

11 Attachement – source code 19

1 Several introduction words (could be skipped)

From time to time somebody asks me about help with debugging or information about how can something be debugged. Because bugs varies, there are a lot of debugging techniques, each of them fits better or worse to the particular type of bugs. However, there is one debugging technique, or, better – a tool, which could be used on a large variety of problems and although you can in many cases find some special, better fitted tool to debug that particular type of bug, mastering of this kind-of universal tool will pay off many times in future.

As the title already told you, i am writing about gdb. Gdb, GNU Debugger, is a tool proven by time, it can be used for both post-mortem analysis and from very basic to very advanced life debugging.

There is a lot of good documentation on gdb, so, why write another one? A lot of people will point you to the gdb documentation. It is very good, but... it is *very* long!

When somebody asks me of help, he does not expect me to point him/her to the hundreds pages of documentation. He usually expects me to either help with his problem, or to show him how he can solve it by himself.

And because i am lazy and have a limited time, and i do not like to repeat myself too many times, i decided to write this, aiming to create kind of compact training material for gdb.

This material is in no way complete in the sense of showing you all the gdb features. It aims just to be a good starter which should kick you up on your gdb-usage journey. And, it should be a thing i will point at saying 'you want to explain how to use gdb? Did you go through <this>? No? Let's go there and ask me again if you still do not know what to do after you are through it.'

That's it.

2 License

3 Preparations

First, the tour is aimed for those who already has got some experience with programming. Although it could be of a good value for a skilled newbie, experienced programmer would probably value this much more. Just to got a clue: you should know what is a *core* (core-file), *preprocessor*, *symbol* and *pointer* before going on with the tour.

You will be confronted with a short C program, with 5 prepared failures. You will go through the failures and learn how to use gdb from the very basic usage to a reasonably good usage level.

I have chosen a field of pointer errors. I do not want to elaborate on why i have chosen this. If you are curious, ask me.

3.0.1 Now really the instructions

- this tour has been tested on a Linux i386 box. While it should work just anywhere gcc/gdb is present, it is recommended to go through it on the same platform if you got the possibility to do so.
- note that the output from your locally runned gdb can be a bit different to the one which is copy-pasted here. I will try to point out several possible and significant differences i found when going through the tour on several different boxes

- download the example file (`http://www.bzz.cz/data/example.c`), assure you has got several MiBs of free space (for the corefile)
- go to the directory where you downloaded the source code and stay there for the whole session
- compile the example using something like

  ```
  gcc -g -o example example.c
  ```

- allow creation of core files by disabling the core file size limit:

  ```
  ulimit -c unlimited
  ```

  Don't forget that this will be valid only in the shell you run the command in, so you need to repeat this next time you want to continue the tour with a fresh shell

You are ready for the tour now.

4 Failure number 0

4.0.2 First run – learning to read gdb output

Run the compiled example by

```
./example
```

You get something like:

```
$ ./example
Some message -- alloc done
Some message -- alloc done
Some message -- alloc done
Segmentation fault (core dumped)
$
```

Program crashed, leaving core file. Look at the core file:

```
$ ls -l core
-rw------- 1 bazil bazil 282624 2008-01-08 00:00 core
```

Prior to inspecting the core file, lets have a brief look at the sources first:

- code is very short, containing only the `main()` and the `alloc_this()` function.
- code is full of errors, of course (so we have something to debug)
- `main` accepts one parameter – you can choose which bug you want to hit. Specifying no parameter at all makes you hit bug 0.
- generally speaking, `main()` does some allocation calls and then simulates several errors by using invalid pointers, malicious copying, calling `free()` with invalid arguments etc.
- the `alloc_this()` func. allocates some space – either on *stack* or on a *heap*, possibly clearing it. It also can simulate a very trivial error (bug 0), we are just getting to it

Now, you are ready to do some *gdb debugging.* Run gdb – i will past a full output here for now

```
$ gdb example core
GNU gdb 6.6.90.20070912-debian
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
Using host libthread_db library "/lib/i686/cmov/libthread_db.so.1".
warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./example'.
Program terminated with signal 11, Segmentation fault.
#0 0xb7e70c3c in memcpy () from /lib/i686/cmov/libc.so.6
(gdb)
```

Well, a lot of massages... anything useful? Skipping version and license information you can see some warning (not important to us) and from the first line beginning with `Reading` it is useful.

`Reading` + `Loaded` informs you what binaries (libraries in this case) are opened by gdb to get the symbols resolved. Usually you debug a bit more sophisticated program than in our example so you would see more `Reading` and `Loading` messages.

```
Core was generated by `./example'.
```

tells you the name of the binary which generated the corefile.

Next line is really important:

```
Program terminated with signal 11, Segmentation fault.
```

So, now you know, that the program was terminated because it received *SIGSEGV* (segment violation) signal which means it did an *invalid memory reference* (tried to access memory without having the privilege to do so).

Useful isn't it?

Next line is one of the possible gdb differences you can hit. Maybe you are looking at these two lines:

```
#0 0x080485e1 in main (argc=1, argv=0xbf8ffea4) at example.c:47
47 memcpy((char *)p_ok, text, text_size*1000000);
```

instead of the

```
#0 0xb7e70c3c in memcpy () from /lib/i686/cmov/libc.so.6
```

For the sake of training, let's count on that you are seeing the one-line thing.

Now you know that the problem was in calling the `memcpy()`. So, either the source or the destination pointer of `memcpy` was an *invalid reference*.

So what, are you going to check all the `memcpy` commands? Yeah, you can do this for such a small program like this example one, but you cannot do this in normal situation. You can try some kind of real-time checker, e.g. `valgrind`, but you need to be able to run the program – *"reproduce the issue"*. In the case you can only do the post-mortem analysis (using the corefile), `valgrind` cannot help you. You can have a try with a static code analyze (using e.g. `lint`) – but, this is a dynamic memory problem so it probably won't be reported by a static analyze tool.

But wait – we are just running post-mortem in gdb! So, it could do no harm doing three keystrokes here, right?

4.0.3 Backtrace

```
(gdb) bt
#0 0xb7e70c3c in memcpy () from /lib/i686/cmov/libc.so.6
#1 0x080486d9 in main (argc=1, argv=0xbfaedad4) at example.c:47
```

Look – the guilty one is the `memcpy()` call *at line 48 of* `example.c`! Look there:

```
memcpy((char *)p_ok, TEXT, text_size*1000000);
```

You do not need any other tool than gdb for such a trivial problem. You can now examine the code in your favorite editor and find out where the problem exactly is. Or you can use the debugger to inspect things a little bit more.

4.0.4 Debugging information – how to get them in

1. not *stripping* the binary after the compilation you retain the *symbols* – aka basic debugging information (you will see the function names in the *backtrace*)
2. compiling in *debugging information* (`-g` option to the compiler) you are able to see source file names, line numbers and even much more (see later)
3. using `ulimit -c unlimited` you enable the *corefile* creation
4. calling `gdb <binary> <core>` you invoke the gdb session
5. typing `bt` at the gdb prompt usually rewards you with the backtrace which can help you much when analyzing problem

However, there is one really important thing to be aware of:

4.0.5 Incorrect and incomplete backtrace

- The gdb can show you the backtrace only and only if there are all involved elements available. What does it means? It means that if you are missing some dynamic library when you start gdb session, the backtrace *will be incomplete*. It also means, that when there is a (lot) different library version loaded in the session than it was at the original environment the process was running in, you can get *incorrect backtrace*. This happens in the case you get the corefile from the "*customer*" and you are *not having the same* system *libraries* (sometimes, you can have a very different `libc` – with a different threading support or so). In that case, you have three options:

  1. get the whole set of libraries from the customer (make a list with the complete paths and send it to the customer. You should be able to get the list of the mapped libraries and the paths from where they were mapped from the core file – consult the gdb manual). After that, consult gdb manual how to tell gdb that it should load the libraries from the different path (search for `solib-absolute-prefix`) or load them manually one by one (again, look-up the manual for that)
  2. let the customer run `gdb <binary> <core>`, type `bt` and send you the output. It may require him to install the gdb and unset the corefile limit.
  3. live with it

4.0.6 Basic commands

Let's look at this a bit more. You have some more useful tools built-in the debugger.

```
(gdb) bt
#0 0xb7e70c3c in memcpy () from /lib/i686/cmov/libc.so.6
#1 0x080486d9 in main (argc=1, argv=0xbfaedad4) at example.c:47
```

What are these `#0` and `#1` exactly? They are *stack frames* (if you see only one stack frame in your gdb, no problem, you will practice the frames later). What is a stack frame? It is a function call entry in the stack. What does it hold? It does hold return address, function arguments, local variables. What could this be useful for? Ah, comon:

```
(gdb) frame 1
#1 0x080486e3 in main (argc=1, argv=0xbf9ae194) at example.c:47
47 memcpy((char *)p_ok, text, text_size*1000000);
```

Now we are at the *`main()` stack frame*.

```
(gdb) info locals
p_ok = (void *) 0x804a008
p_clear = (void *) 0x804a048
p_stack = (void *) 0xbfc0d2c0
switcher = 0
```

There are the *local variables*.

```
(gdb) list
42 p_ok = alloc_this(FALSE, FALSE, FALSE, text_size);
43 p_clear = alloc_this(TRUE, FALSE, FALSE, text_size);
44 p_stack = alloc_this(TRUE, TRUE, FALSE, text_size);
45
46 if (!switcher) {
47         memcpy((char *)p_ok, TEXT, text_size*1000000);
48 }
49
50
51 memcpy((char *)p_ok, TEXT, text_size);
```

You see just the surroundings of the failed call.

Ok, now look at the arguments:

```
(gdb) print p_ok
$3 = (void *) 0x804a008
(gdb) print text
$4 = "This is some very clever and not at all short sample text\n"
(gdb) print text_size
$5 = 59
(gdb) print text_size*1000000
$6 = 59000000
```

So, now you can clearly see, that the call to memcpy requested to copy 59MB from the `text` string to the `p_ok` variable. We cannot say it the error occurred when reading the 59MB from `text` or when trying to store 59MB to `p_ok`. It does not matter, the error is in specifying the enormous size.

We found first error and are able to fix it. Fix and recompile it if you want.

4.0.7 Remember commands

- bt
- frame
- info
- list
- print

and the bonus one:

- help

which is an online help system. Remember it is there, you will need it some time.

5 Failure number 1

Run the example binary (either you fixed the 0 bug and recompiled or not) with the parameter 1:

```
$ ./example 1
Some message -- alloc done
Some message -- alloc done
Some message -- alloc done
Segmentation fault (core dumped)
```

Output looks the same as failure 0. But, open it using gdb. No, i won't copy paste full input/output here, you should know how to call the gdb by now.

After looking at the backtrace, you should see something like

```
(gdb) bt
#0 0xb7ec7c35 in memcpy () from /lib/i686/cmov/libc.so.6
#1 0x080485bc in alloc_this (flag_clear=0, flag_onstack=0, flag_fail=1,
size=59) at example.c:19
#2 0x08048746 in main (argc=2, argv=0xbfff62d4) at example.c:54
```

if your gdb shows you the libc frame or something like

```
#0 0x08048556 in alloc_this (flag_clear=0, flag_onstack=0, flag_fail=1,
size=59) at example.c:19
#1 0x080486e4 in main (argc=2, argv=0xbfcc17b4) at example.c:54
```

if it does not. Be it either case, you now have at least two frames. You can try the `frame` command to switch between frames and inspect the local variables.

Now, it is time for something new. Exit debugger.

5.0.8 Live debug

Run debugger with this command line:

```
$ gdb example
```

So, we omitted the core file. We are going to run the program inside the debugger!

First, we want to trigger failure 1, not 0, so we need to set the program arguments somehow.

```
(gdb) set args 1
(gdb) run
```

You should end up with the program getting `SEGV` signal – this is the same state as if you were inspecting the corefile.

5.0.9 Breakpoints

Okay, anything new?

Yes. You have the possibility to stop the program just before hitting the bug. Looking at backtrace, you want to break the program somewhere in the `alloc_this` function, right?

To break the program, the debugger is using a *breakpoint*. It is just a point in the instruction flow, where the program execution is suspended and the debugger comes into charge again.

So set a breakpoint there:

```
(gdb) break alloc_this
Breakpoint 1 at 0x804844c: file example.c, line 14.
```

Gdb will insert a breakpoint just at the beginning of the `alloc_this` function. Now, run the program again:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bazil/actual/gdb/example 1
Breakpoint 1, alloc_this (flag_clear=0, flag_onstack=0, flag_fail=0, size=59)
   at example.c:14
14 void * p_bug = NULL;
```

Would you look in the sourcefile, you could see we are at the very first line of this function. Now call backtrace and you can see:

```
#0 alloc_this (flag_clear=0, flag_onstack=0, flag_fail=0, size=59)
   at example.c:14
#1 0x0804856e in main (argc=2, argv=0xbfca8284) at example.c:42
```

Wait, the line numbers are different from the case when we were examining corefile of this error, aren't they?

Yes, if you look at the first backtrace in the failure 1, you can see that it was

```
main():54 -> alloc_this():19
```

But now it is

```
main():42 -> alloc_this():14
```

Lets have a look at the sources. At the `main():54`, there is a `alloc_this()` call which failed. At `main():42`, there is first `alloc_fail()` call in the program. Because we specified to break at each call of `alloc_this()` and we restarted the program, we are now seeing the first `alloc_this()` call.

We do not want to inspect this particular `alloc_this()` call because there is no failure here.

Lets continue.

```
(gdb) cont
Continuing.
Some message -- alloc done
Breakpoint 1, alloc_this (flag_clear=1, flag_onstack=0, flag_fail=0,
   size=59) at example.c:14
14 void * p_bug = NULL;
(gdb) bt
#0 alloc_this (flag_clear=1, flag_onstack=0, flag_fail=0, size=59)
   at example.c:14
#1 0x08048596 in main (argc=2, argv=0xbfca8284) at example.c:43
```

Now we are at `main():43 -> alloc_this():15`. Let's do the `continue` command twice more and get the `alloc_this()` call in which we are interested at.

```
(gdb) bt
#0 alloc_this (flag_clear=0, flag_onstack=0, flag_fail=1, size=59)
   at example.c:14
#1 0x0804863a in main (argc=2, argv=0xbfca8284) at example.c:54
```

We are at the beginning of the failing `alloc_this()` call. We are going to proceed one source code line by another. There is a `next` command:

```
(gdb) next
16 void * mem_ptr = (flag_onstack) ? alloca(size) : malloc(size);
(gdb) next
19 if (flag_fail) memcpy((char *) p_bug, text, text_size);
```

In fact, you could have used `next 2` to do `next` two times in row.

Now, you are at the failing line. Inspect thing a little using the `print` command (won't paste it in here because it is just a repetition from failure 0)

Now, let's say that you have a real program, and you know that function `qweasd()` is the failing one. But it is failing only once a thousand calls. You do not want do type `cont` one thousand times. In gdb, you have several options depending on what you *exactly* need, let's look at several breakpoint commands now:

Ways of setting and managing breakpoints

`break` alone sets breakpoint at the very location you are at. If you are able to get using the `next` commands to a particular situation you want to break at the next time it happens, use this.

`break example.c:19` you set the breakpoint to the specified line. This could be extremely useful if you want to break in the specific branch of `if` command (note the specific failure once a thousand calls above)

`info break` shows you the actual breakpoint set. In fact, this earns an example:

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x0804844c in alloc_this at example.c:14
breakpoint already hit 4 times
2 breakpoint keep y 0x0804849c in alloc_this at example.c:19
```

Num the numeric id of the breakpoint. Does not change (autoincrement)

What shows you where is the breakpoint set

Enb shows you if the breakpoint is currently enabled. If you disable it, if won't be hit until you enable it again.

Now some commands to manage breakpoints:

`disable 1` disables first breakpoint. Still in the table, but execution won't be stopped there

`delete 1` will delete first breakpoint

`condition 2 flag_fail==1` make fifth breakpoint hit only when =flag_fail= parameter is set to 1

`help breakpoints` can get you to the next level

Now, play with it a little bit. Erase all breakpoint (or disable them), create new one and make it hit only when `flag_fail` is a positive number. Then run the program again and you would get to the failing case immediately.

Remember the `next` command? It will just execute the source line you are at and stop executing before the next line of actual source file get executed. Consider you are in the `main()`, the actual line specifies an `alloc_this` call, and you do not want to just jump over to next line, you want to be able to see what is going on in the `alloc_this` call. You can use the `step` command to *step inside* the function. Using it would move you to the first line of the `alloc_this` function.

5.0.10 Variable value mangling

Now, we will do some magic to end-up this chapter.

I suppose you either made the conditional breakpoint as described above or any other kind of breakpoint with which you can get to the failing case of the `alloc_this()` function call.

Run the program and get into the failing case, but do not execute the failing `memcpy()` call – just get inside the `alloc_this` function. Check you have the failing case by inspecting the `flag_fail` value by the `print` command.

```
Now, do this:
(gdb) print flag_fail
$1 = 1
(gdb) set flag_fail = 0
(gdb) print flag_fail
$2 = 0
(gdb) cont
Continuing.
Some message -- alloc done
Program exited with code 01.
(gdb)
```

Wow, the program passed without crashing – you have fixed it! Okay, you have workarounded it by manual intervention. But more important is the knowledge that you can change the variable value, because the main purpose of this is *simulating error conditions* – so just the opposite we did. Imagine you are suspecting that some function may cause the failure but you do not have the exact data which sets some flags... using this you can set the flags easily. Another example is just exploring "what if this was set like this...". Combining

1. using breakpoints to get where you want
2. set the variable to the values you want
3. rerun how many times you want

You can get very good autumn bug harvest.

5.0.11 What you should remember

- how to run and re-run the program inside the debugger (`run`)
- how to set program cmdline arguments from the debugger (`set args ...`)
- what is a breakpoint and how to set it (`break`, `break ...`)
- you can set breakpoints at function or an exact line (in fact also to the exact address)
- how to manage the breakpoints (`info break`, `disable ...`, `delete ...`)
- continuation commands (`next`, `step`, `continue`)
- that you can mangle the variable values (`set <variable>`)
- you can do conditional breakpoints and much more – refer to `help breakpoints`

6 Failure number 2

This failure is a simple one, but you will learn two new things here.

Let's call `./example 2`. You get a corefile, run an gdb on it, execute `bt full` command. You should see something like this:

```
(gdb) bt full
#0 0xb7eec4db in strlen () from /lib/i686/cmov/libc.so.6
No symbol table info available.
#1 0x08048775 in main (argc=2, argv=0xbfadeac4) at example.c:67
p_ok = (void *) 0x0
p_clear = (void *) 0x804a048
p_stack = (void *) 0xbfade980
switcher = 2
```

The bug is hit inside the `strlen()` call. Because the `libc` is not compiled with debugging information, you cannot see the arguments. However, you do have full *call stack*. So look at the caller.

Looking at local variables of `main()`, there is nothing suspicious, only the `p_ok` is pointing to null. Now look at the guilty line.

```
(gdb) frame 1
#1 0x08048775 in main (argc=2, argv=0xbfadeac4) at example.c:67
67 fprintf(stderr, "Kernel mem length is %d\n", strlen((char *) 1));
```

Now, examine the argument value passed to strlen:

```
(gdb) print (char *) 1
$1 = 0x1 <Address 0x1 out of bounds>
```

Gotcha! The problem is in accessing invalid memory pointer. This error illustrates the case when there is a pointer error, but that the pointer is not stored in any variable, so you are not able to tell something just from the backtrace, you need corefile to inspect the arguments. (In fact, in this case, just looking at the source line number available from the full backtrace you can recognize that accessing to pointer address 1 is not correct, but this was just an illustration)

Now, let's have a look at another handy gdb feature. Stop the debugger, and run it on a binary (without the core file). Set commandline argument to 2, breakpoint on line 67, run the program.

You are now just before the failure.

6.0.12 Calling functions from debugger

Gdb can do something like this:

```
(gdb) call strlen("text")
$1 = 4
```

So, try the real argument:

```
(gdb) call strlen((char *) 1)
Program received signal SIGSEGV, Segmentation fault.
0xb7e614db in strlen () from /lib/i686/cmov/libc.so.6
The program being debugged was signaled while in a function called from GDB.
GDB remains in the frame where the signal was received.
To change this behavior use "set unwindonsignal on"
Evaluation of the expression containing the function (strlen) will be abandoned.
```

Congratulation, you have hit the bug!

Now look at the backtrace, it gives you a hint that you manipulated the calls a little bit.

6.0.13 What you should remember

- `bt full` to show the local variables directly with the backtrace (useful if you want somebody do send you a backtrace output when debugging on a remote site)
- even the full backtrace is not enough sometimes
- you can inspect any expression value, not only the variables
- when running a program, you can call it's functions with arguments you want (does not work when inspecting core file)

7 Failure number 3

This failure is similar to failure number 2, but although this is the easiest chapter, you still have possibility to learn something new here.

Now run the bugger with `./example 3`

Look at the backtrace:

```
(gdb) bt
#0 0xb7e348eb in strlen () from /lib/tls/libc.so.6
#1 0xb7e0821e in vfprintf () from /lib/tls/libc.so.6
#2 0xb7e03d13 in cuserid () from /lib/tls/libc.so.6
#3 0xb7e0490f in vfprintf () from /lib/tls/libc.so.6
#4 0xb7e0d3c2 in fprintf () from /lib/tls/libc.so.6
#5 0x080486fb in main (argc=2, argv=0xbfd14c64) at example.c:71
```

You can see the invalid memory access was done in `strlen` function. But this is too deep in the call stack. Looking at our example, the guilty line is 71, which calls `fprintf` – according to the call stack.

Look around – select your `main()` frame, inspect locals and so (`frame`, `info locals`, `list`)

Because the `libc` has no *symbol table* available (try selecting the `fprintf` frame called from `main()` and looking at the local variables), you cannot see the arguments passed to the `fprintf` call.

Looking at the source line:

```
fprintf(stderr, "Kernel mem: %s\n", (char *) MACRO(1));
```

doesn't tell you that either. Okay, try the gdb print command:

```
print MACRO(1)
No symbol "MACRO" in current context.
print MACRO
No symbol "MACRO" in current context.
```

No, it just won't tell you the real fprintf argument value, because it is a macro, and macros are evaluated during preprocessing phase. In our case it does not matter that much because we are having a trivial macro and a very simple bug, but what can you do in more complicated situations?

7.0.14 Evaluating macros

It is possible to instruct the compiler to put even the macro processing information into the resulting binary. With gcc, you need to recompile the example with:

```
gcc -g3 -o example example.c
```

The `-g3` option will cause more debugging info than with `-g` will be compiled in. Refer to the gcc man page to learn more about it.

Now, rerun `./example 3` and issue `gdb example core`.

Backtrace is still the same, let's switch to the `main()` frame.

You have at least two possibilities how to get to the evaluated macro value, passed to the `fprintf` call.

First, you can use the gdb command `macro`:

```
macro expand MACRO(1)
```

Second, you can use common print command:

```
print MACRO(1)
```

Now you are sure, that the `fprintf` should print string including string placed at the address 2. This fails, because there is no accessible string stored there.

7.0.15 Things to remember

- recompiling with `-g3` option will compile in information for evaluating the macros
- `macro expand` expands the macro in the same way preprocessor does
- `print` works for evaluating the macro value

8 Failure number 4

8.0.16 Overwritten stack

Failure number 4 is a bit advanced topic. It does not need any extra skills or so, but because it tries to illustrate something which can be classified as a type of buffer overflow and how it could happen that your program produced an (completely) useless coredump, it needs a bit understanding of a an (i386) stack.

But even if you are not interested in above (but you should be, because the overwritten stack case can happen to you possibly under a completely different conditions), you will learn a bit of a gdb *beginner magic* here.

However, i must note that depending on a several things, you have a chance of not overwriting the call stackwhen running on your hardware. You can try to increase the the number of the `for` cycles at line 58, but event then, depending on the situation, you may actually end up with hitting `SIGSEGV` without overwriting the stack. Just try it. If you still cannot overwrite the stack, don't be sad, you can learn the promised magic without overwriting stack.

Enough of talks, run `./example 4`

If you haven't forgot to unset the corefile limit, you should get a coredump file.

Go on, examine it with gdb. After running gdb, your `bt` output could look like:

```
(gdb) bt
#0 0x20656c70 in ?? ()
#1 0x74786574 in ?? ()
#2 0x0804000a in ?? ()
#3 0x0000003b in ?? ()
#4 0x0000003b in ?? ()
#5 0xbfb1aa80 in ?? ()
#6 0x08049a10 in ?? ()
#7 0xbfb1aa58 in ?? ()
#8 0x080483f0 in _init ()
#9 0xb7e3d050 in __libc_start_main () from /lib/i686/cmov/libc.so.6
#10 0x080484e1 in _start ()
```

Hmm, interesting, isn't it? Where oh where is my `main()` frame? Yes, it is definitely interesting, but... useless. You can see something like this (if i omit some significant library difference between the analyzed machine and the machine you are running gdb on) if *the backtrace gets overwritten.*

Let's show how that happened.

Start gdb, set commandline argument to `4`, and before running the program, set some breakpoints – e.g. first one to `main()`, second to `alloc_this()`.

Get ready some editor/viewer on the example source file.

Now, run it and continue several times after you get the `SIGSEGV` error.

You should see something like this:

```
(gdb) r
Starting program: /home/bazil/actual/gdb/example 4
Breakpoint 1, main (argc=2, argv=0xbfd05524) at example.c:32
32 int switcher = 0;
(gdb) c
Continuing.
```

```
Breakpoint 2, alloc_this (flag_clear=0, flag_onstack=0, flag_fail=0, size=59)
at example.c:14
14 void * p_bug = NULL;
(gdb) c
Continuing.
Some message -- alloc done
Breakpoint 2, alloc_this (flag_clear=1, flag_onstack=0, flag_fail=0, size=59)
at example.c:14
14 void * p_bug = NULL;
(gdb) c
Continuing.
Some message -- alloc done
Breakpoint 2, alloc_this (flag_clear=1, flag_onstack=1, flag_fail=0, size=59)
at example.c:14
14 void * p_bug = NULL;
(gdb) c
Continuing.
Some message -- alloc done
Program received signal SIGSEGV, Segmentation fault.
0x20656c70 in ?? ()
(gdb)
```

You got to the same unusable state as with the core file. Try `bt`, you will see the mess.

What now? Look at the output above. You can say that at least some of the `alloc_this()` calls went fine. Well, you can see it was the last one in the source file, but in the real situation, it could be the x'th one (neither first, nor last of the `alloc_this()` calls). You probably want to inspect the situation on the last successful `alloc_this` call.

Let's try some promised beginner magic now.

Facts:

1. you have the `alloc_this()` breakpoint set
2. you know that after some `alloc_this()`, there is a failure which overwrites the stack

Needs:

1. you want to inspect the situation at the last successful enter on `alloc_this()`
2. you do not want to interrupt on each `alloc_this()` call

Okay. To be able to inspect the situation at the last successful call to `alloc_this()`, you need to identify that call first.

Let's go the fastest way.

You can specify a commands to be executed on each breakpoint hit.

So.

So what?

It's easy, try to think a little bit before reading further.

8.0.17 Little bit of magic – assigning commands to breakpoints

Let's suppose the `alloc_this()` breakpoint is a number 2. If you do something like:

```
(gdb) commands 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>bt
>continue
>end
```

It would cause that on *each* breakpoint 2 hit, backtrace would be printed and then `continue` command would be executed.

So, you will end up with the same overwritten stack, but the last backtrace will identify the position in the source file where the problem had arisen (at some other cases – like the problem arising on a call executed from cycle – you can use other commands to be executed as well – so you can display e.g. the cycle counter to identify the exact cycle run etc.)

Let's run it, possibly switching off the breakpoint 1 (in `main()`).

The end of gdb output should look like:

```
Breakpoint 2, alloc_this (flag_clear=1, flag_onstack=1, flag_fail=0, size=59)
at example.c:14
14 void * p_bug = NULL;
#0 alloc_this (flag_clear=1, flag_onstack=1, flag_fail=0, size=59)
at example.c:14
#1 0x08048678 in main (argc=2, argv=0xbff16f34) at example.c:44
Some message -- alloc done
Program received signal SIGSEGV, Segmentation fault.
0x20656c70 in ?? ()
(gdb)
```

Now you see, that the last successful `alloc_this()` call was from `example.c`, line 44.

Let's put a breakpoint on that line, disable the `alloc_this()` breakpoint and re-run the program.

You can step in the `alloc_this()` function, but to speed things up, lets go over by `next`. Repeat the `next` as long as you get to the failure:

```
Breakpoint 3, main (argc=2, argv=0xbfb27344) at example.c:44
44 p_stack = alloc_this(TRUE, TRUE, FALSE, text_size);
(gdb) n
Some message -- alloc done
46 if (!switcher) {
(gdb) n
51 memcpy((char *)p_ok, text, text_size);
(gdb) n
52 memcpy((char *)p_clear, text, text_size);
(gdb) n
54 if (switcher==1) (void) alloc_this(FALSE, FALSE, TRUE, text_size);
(gdb) n
56 if (switcher==4) {
(gdb) n
58 for (x=0; x<5; x++) {
(gdb) n
59 memcpy((char *)(p_stack+x*text_size), text, text_size);
(gdb) n
58 for (x=0; x<5; x++) {
(gdb) n
59 memcpy((char *)(p_stack+x*text_size), text, text_size);
(gdb) n
Program received signal SIGSEGV, Segmentation fault.
0x20656c70 in ?? ()
```

The last `memcpy` command is the guilty one.

Fine, it is identified, now we are going to look at it.

Delete all breakpoints and set a new one at line 59.

Set a display of some variables like:

```
(gdb) display x
```

```
(gdb) display p_stack
(gdb) display text
(gdb) display text_size
(gdb) display p_stack+x*text_size
```

Rerun the binary.

Now, for a last time in this failure, let the program die. Use `cont` commands. You will see something like this, only the pointer adresses will be different:

```
Breakpoint 4, main (argc=2, argv=0xbfd10534) at example.c:59
59 memcpy((char *)(p_stack+x*text_size), text, text_size);
5: p_stack + x * text_size = (void *) 0xbfd103f0
4: text_size = 59
3: text = "This is some very clever and not at all short sample text\n"
2: p_stack = (void *) 0xbfd103f0
1: x = 0
(gdb) c
Continuing.
Breakpoint 4, main (argc=2, argv=0xbfd10534) at example.c:59
59 memcpy((char *)(p_stack+x*text_size), text, text_size);
5: p_stack + x * text_size = (void *) 0xbfd1042b
4: text_size = 59
3: text = "This is some very clever and not at all short sample text\n"
2: p_stack = (void *) 0xbfd103f0
1: x = 1
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x20656c70 in ?? ()
```

Looking at the memcpy line and other facts, we can say:

- the program died after copying the `text` to the `0xbfd1042b` address
- the program died due to a `SIGSEGV` which is invalid access
- program tried to access some data which it has no privilege to access
- the stack is completely useless

The `SIGSEGV` you are seeing was not hit by the `memcpy` directly. In fact, the memcpy function wrote the text over some stack data and when the program was trying to access (or maybe execute, who knows) another data just after the memcpy finished, it did invalid memory access so it received the `SIGSEGV`.

Now, the question is, how come that the stack (or part of it) got overwritten? Originally, i wanted to go through the stack addresses, compare it to `p_stack` pointer and do some pointer math to show you what happened, but given the fact it is long enough now and your patience is probably lost by now, i will give you the start and you can inspect yourself. So:

- the problem is, that the `p_stack` is pointing to *stack*, not to *heap*. When the `alloc_this()` function is executed, the space for `p_stack` is allocated *on the stack* because there is an `alloca()` function used instead of a `malloc()`. You can generally use `alloca()` for quick allocation but you have to remember what you are doing. This code is an example of not remembering that.
- the p_stack variable value is in fact never valid, because the value is stored here *after* the execution of `alloc_this()` ends. At that time, all the stack space occupied by `alloc_this()` local storage is *considered free*. So, the `p_stack` points to

  - memory which can be accessed by the program without restriction
  - memory which lies on the program stack
  - memory which should never be accessed because semantically the `p_stack` pointer is invalid

- from now, it is only the matter of time when the real problem arises. It could be after some time (the `p_stack` pointer can be passed to any other function call which can overwrite it's own data, return pointer etc...) or just immediately like it is in the example: the for cycle calling memcpy is of course an error (because it is writing to the stack) but there is even a cumulative error (it is writing more space than was allocated – a check is missing)

One little help to get to the stack address: `info frame` will show you the address of the frame on the stack. In my case (after re-running the binary, addresses are a bit different):

```
(gdb) info frame
Stack level 0, frame at 0xbfd6a4f0:
eip = 0x8048715 in main (example.c:59); saved eip 0xb7db9050
source language c.
Arglist at 0xbfd6a4e8, args: argc=2, argv=0xbfd6a584
Locals at 0xbfd6a4e8, Previous frame's sp at 0xbfd6a4e4
Saved registers:
ebp at 0xbfd6a4e8, eip at 0xbfd6a4ec
(gdb) display
5: p_stack + x * text_size = (void *) 0xbfd6a47b
4: text_size = 59
3: text = "This is some very clever and not at all short sample text\n"
2: p_stack = (void *) 0xbfd6a440
1: x = 1
(gdb) print p_ok
$1 = (void *) 0x804a008
```

So you see that memory allocated on heap (`p_ok`) is at `0x804a008`, the stack frame 0 begins at `0xbfd6a4f0` and the p_stack points to `0xbfd6a440` (which is much closer to stack frame 0 than to the heap). The address which is the text going to be copied now is `0xbfd6a47b`, which is even closer to the frame 0. Rewriting of something important on the stack is only the matter of time...

8.0.18 Important things to remember

- commands can be assigned to breakpoints
- `display` sets items to be displayed after each command executed (if visible in current *scope*)
- `info frame` shows frame address on the stack
- `alloca()` allocates on stack
- after the stack is overwritten, a lot of strange things could happen, this was only one simple example

9 Failure number 5

Run the example with argument 5 to get the last prepared failure.

It should fail on a `SIGSEGV` again. Your backtrace could look like

```
#0 0xb7eabaf6 in free () from /lib/tls/libc.so.6
#1 0x08048712 in main (argc=2, argv=0xbfa6a9b4) at example.c:75
```

This failure does not tell you anything brand new (well, only a bit), it is mainly meant as an final exercise. If you remember the previous failure, just looking at the source code will tell you where is the problem now. But do not hurry, take the exercise first.

First, inspect the situation a bit. Look at the frame, locals, print the source code.

Then, let's say you decided you need to see what happens live.

Exit debugger, run it with binary only.

Set proper arguments.

Wait, can we put the breakpoint inside the dynamic library? Try it:

```
(gdb) break free
Function "free" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

You may decide to make a *pending* breakpoint, or set break for `main()` and just when program starts and hits breakpoint at `main()` you may set the breakpoint at `free()`.

Now, run the program, and inspect each `free()` call a bit. Let's say you are paranoid, so set the frame to your `main()` at each `free()` call and look at the address to be freed.

When you get to the failing `free()` call, switch to the `main()` frame, look at the `p_stack` value and compare with stack frame address. You see the problem is that this pointer was not allocated by `malloc()` or any of its wrappers, because it is a pointer somewhere to the stack. Because it was not allocated via `malloc`, it cannot be freed.

That's all of the prepared failures i have got for you. However, don't miss the next chapter.

10 Where from now?

Let me give you one little advice: you can learn how to use gdb best by trying to use it. Where can you found real problems if you are not full-time developer and not solving customer problems?

First place is your own programs. Do not take care that they are (usually) just a small applications, just debug it if you hit the bug.

Second place to look are the real bugs – look at some opensource software (be it `GNU` or not) – there are lot of various sized applications with a lot of bugs reported. Pick some app and try to debug a known reported problem. By solving the problem you will help users of the application and you get some practice. On the other hand, you can start with already fixed problem, because there is usually no problem obtaining the old revision with the bug not yet fixed.

10.0.19 External links

If you are looking for some additional information (and you should be because this article is just to start you up), consider:

- internal gdb help system
- gdb documentation on gdb homesite -- `http://sourceware.org/gdb/documentation/`
- gdb quickref – try your favorite search engine, original site here or local copy here: `http://www.bzz.cz/mirror/gdbref.pdf`

Final words There is always a lot more what can be written about gdb or debugging in general, but i hope this short (compared to the doc) text helped you with learning gdb. Farewell!

–

Jan '*bazil*' Otte, 2007

should you need to contact me, write to *"incoming at <domain name – bzz cz>*".

11 Attachement – source code

You should be able to get this online at `http://www.bzz.cz/data/example.c`

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

#define FALSE 0
#define TRUE 1
#define MACRO(x) (x+1)

int text_size = 0;
char text[] = "This is some very clever and not at all short sample text\n";

void * alloc_this(int flag_clear, int flag_onstack, int flag_fail, size_t size) {
/* little allocation function, can alloc on heap and on stack,
   clear the memory and simulate an error (copy to NULL) */
    void * p_bug = NULL;
    /* allocate memory */
    void * mem_ptr = (flag_onstack) ? alloca(size) : malloc(size);

    /* simple failure */
    if (flag_fail) memcpy((char *) p_bug, text, text_size);

    /* clear memory if flag set and memory alloc succeeded */
    if (mem_ptr && flag_clear) mem_ptr = memset(mem_ptr, 0, size);

    /* just a clue -- you can break here */
    printf("Some message -- alloc done\n");

    return mem_ptr;
}

int main(int argc, char ** argv) {
    void * p_ok, * p_clear, * p_stack;
    int switcher = 0;

    text_size = strlen(text) + 1;

    /* get the switcher if any */
    if (argc>1) {
        switcher=atoi(argv[1]);
    }

    /* allocate it */
    p_ok = alloc_this(FALSE, FALSE, FALSE, text_size);
    p_clear = alloc_this(TRUE, FALSE, FALSE, text_size);
    p_stack = alloc_this(TRUE, TRUE, FALSE, text_size);

    if (!switcher) {
        memcpy((char *)p_ok, text, text_size*1000000);
    }

    memcpy((char *)p_ok, text, text_size);
    memcpy((char *)p_clear, text, text_size);
```

```
    if (switcher==1) (void) alloc_this(FALSE, FALSE, TRUE, text_size);

    if (switcher==4) {
        int x;
        for (x=0; x<5; x++) {
            memcpy((char *)(p_stack+x*text_size), text, text_size);
        }
    }

    free(p_ok); p_ok = NULL;
    free(p_clear);

    if (switcher == 2) {
        fprintf(stderr, "Kernel mem length is %d\n", strlen((char *) 1));
    }

    if (switcher == 3) {
        fprintf(stderr, "Kernel mem: %s\n", (char *) MACRO(1));
    }

    if (switcher>4) {
        if (p_stack) free(p_stack);
    }

    return switcher;
}
```