Working with GDB

Abstract

Sometimes, the process of investigating problems by reading logs and examining the overall behavior of a high performance computing system will not yield fruitful results. The root cause of the issue affecting your customer base or your applications will remain a mystery. At this stage, you will need to dig in deeper, and this means analyzing what affected processes do. This article will explore debugging techniques at a geeky level.

Article

Working with GDB

Now, let's talk debug. So you wrote a piece of code and you want to compile it and run it. Or you have a binary and you just run it. The only problem is, the execution fails with a segmentation fault. For all practical purposes, you call it a day. To make things worse, your customer is screaming for help, because their critical piece of software is failing, and they are clueless as to what might be the root cause.

In this section, we will learn how to handle misbehaving binary code, how to examine its execution step by step, how to interpret errors and problems, and we will even step into the assembly code and hunt for problems there.

Prerequisites

We would like to emphasize that this will not be easy. Working with gdb is not something anyone can do at their leisure. There are many requirements you must meet before you can have a successful session.

Source Files

You can debug code without having access to source files. However, your task will be more difficult, because you will not be able to refer to the actual code and try to understand if there's any kind of logical fallacy in the execution. You will only be able to follow symptoms and try to figure out where things might be wrong, but not why.

Source Code Compiled with Symbols

On top of that, you will want source code with symbols, so you can map instructions in the binary program to their corresponding functions and lines in the source code. Otherwise, you will be groping in the dark.

Understanding of the Linux System

This is probably the most important element. First, you will need some core knowledge of the memory management in Linux. Then, the fundamental concepts like code, data, heap, stack, and so on. You should also be able to navigate /proc with some degree of comfort. You should also be familiar with the AT&T Assembly syntax, which is the syntax used in Linux, as opposed to Intel syntax, for example.

Simple Example

We will begin with a simple example: a null pointer. In layman's terms, a null pointer is a pointer to an address in the memory space that does not have a meaningful value and cannot be referenced by the calling program, for whatever reason. This will normally lead to an unhandled error, resulting in a segmentation fault. Here's our source code:

```
#include <stdio.h>

int main (int argc, char* argv[])
{
   int* boom=0;
   printf("hello %d",*boom);
}
```

Now, let's us compile it, with symbols. This is done by using the -g flag when running gcc.

```
gcc -g source.c -o binary.bin
```

And then we run it and get a nasty segmentation fault:

```
#./binary.bin

Segmentation fault
```

Now, you may want to try to debug this problem using standard tools, like perhaps strace, ltrace, maybe lsof, and a few others. Normally, you would do this, because having a methodical approach to problem solving is always good, and you should start with simple things first. However, we will purposefully not do that right now to simplify things. As we advance in the article, we will see more complex examples and the use of other tools, too.

All right, so now we need to start using the GNU Debugger. We will invoke the program once again, this time through gdb. The syntax is simple:

```
#gdb binary.bin

GNU gdb (GDB) SUSE (7.3-0.6.1)

Copyright (C) 2011 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 "x86_64-suse-linux".

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>...

Reading symbols from /tmp/binary.bin...done.

(gdb)
```

For the time being, nothing happens. The important thing is that gdb has read symbols from our binary. The next step is to run the program and reproduce the segmentation fault. To do this, simply use the command run inside gdb.

```
(gdb) run

Starting program: /tmp/binary.bin



Program received signal SIGSEGV, Segmentation fault.

0x0000000000400557 in main (argc=1, argv=0x7fffffffe458) at file.c:6
```

```
6         printf("hello %d",*boom);

(gdb)
```

We see several important details. First, we see that our program crashes. The problem is in the sixth line of source code, as shown in the image, our printf line. Does this mean there's a problem with printf? Probably not, but something in the variable that printf is trying to use, most likely. The plot thickens.

Second, you may also see a message that separate debuginfo (symbols) for third-party libraries, which are not part of our own code, are missing. This means that we can hook into their execution, but we won't see any symbols. We'll see an example soon.

What we learn here is that we have symbols that gdb won't run automatically, and that we have a meaningful way of reproducing the problem. This is very important to remember, but we will recap this when we discuss when to run or not to run gdb.

Breakpoint

Running through the program does not yield enough meaningful information. We need to halt the execution just before the printf line. Enter breakpoints, just like when working with a compiler. We will break into the main function and then advance step by step until the problem occurs again, then rerun and break, and then execute commands one at a time just short of the segmentation fault.

To this end, we need the break command, which lets you specify breakpoints either at functions, your own or third-party loaded by external libraries, or at specific lines of code in your source—an example is on the way. Then, we will use the info command to examine our breakpoints. We will place the breakpoint in the main() function. As a rule of thumb, it's always a good place to start.

```
(gdb) break main

Breakpoint 1 at 0x40054b: file file.c, line 5.

(gdb) info breakpoint

Num     Type           Disp Enb Address            What

1       breakpoint     keep y   0x000000000040054b in main at file.c:5
```

Now we run the code again. The execution halts when we reach main().

```
(gdb) run

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /tmp/binary.bin



Breakpoint 1, main (argc=1, argv=0x7fffffffe458) at file.c:5

5         int* boom=0;

(gdb)
```

Step by Step

Now that we have stopped at the entry to main, we will step through code line by line, using the next command. Luckily for us, there isn't that much code to walk through. After just two steps, we encounter a segmentation fault. Good.

```
(gdb) next

6          printf("hello %d",*boom);

(gdb) next



Program received signal SIGSEGV, Segmentation fault.

0x0000000000400557 in main (argc=1, argv=0x7fffffffe458) at file.c:6

6          printf("hello %d",*boom);

(gdb)
```

We will now rerun the code, break in the main(), do a single next that will lead us to printf, and then we will halt and examine the assembly code, no less!

```
(gdb) run

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /tmp/binary.bin



Breakpoint 1, main (argc=1, argv=0x7fffffffe458) at file.c:5

5          int* boom=0;

(gdb) next

6          printf("hello %d",*boom);

(gdb)
```

Disassembly

Indeed, at this stage, there's nothing else the code can tell us. We have exhausted our understanding of what happens in the code. Seemingly, there doesn't seem to be any great problem, or rather, we can't see it yet.

So we will use the disassemble command, which will dump the assembly code. Just type disassemble inside gdb and this will dump the assembly instructions that your code uses.

```
(gdb) disassemble

Dump of assembler code for function main:

   0x000000000040053c <+0>:     push   %rbp
   0x000000000040053d <+1>:     mov    %rsp,%rbp

   0x0000000000400540 <+4>:     sub    $0x30,%rsp

   0x0000000000400544 <+8>:     mov    %edi,-0x14(%rbp)

   0x0000000000400547 <+11>:    mov    %rsi,-0x20(%rbp)

   0x000000000040054b <+15>:    movq   $0x0,-0x8(%rbp)

=> 0x0000000000400553 <+23>:    mov    -0x8(%rbp),%rax

   0x0000000000400557 <+27>:    mov    (%rax),%esi

   0x0000000000400559 <+29>:    mov    $0x400664,%edi

   0x000000000040055e <+34>:    mov    $0x0,%eax

   0x0000000000400563 <+39>:    callq  0x400430 <printf@plt>

   0x0000000000400568 <+44>:    leaveq

   0x0000000000400569 <+45>:    retq

End of assembler dump.
```

This is probably the most difficult part of the tutorial yet. Let's try to understand what we see here, again in very simplified terms.

On the left, we have memory addresses. The second column shows increments in the memory space from the starting address. The third column shows the mnemonic. The fourth column includes actual registers and values.

There's a little arrow pointing at the memory address where our execution is right now. We are at offset 40054b, and we have moved the value that is stored 8 bytes below the base pointer into the RAX register. One line before that, we moved the value 0 into the RBP-8 address. So now, we have the value 0 in the RAX register.

```
   0x000000000040054b <+15>:    movq   $0x0,-0x8(%rbp)

=> 0x0000000000400553 <+23>:    mov    -0x8(%rbp),%rax
```

Our next instruction is the one that will cause the segmentation fault, as we have seen earlier while stepping through the code with the next command.

```
0x0000000000400557 <+27>:    mov    (%rax),%esi
```

So we need to understand what's wrong here. Let's examine the ESI register, which is supposed to get this new value. We can do this by using the examine or x command. You can use all kinds of output formats, but that's not important right now.

```
(gdb) x $rax

0x7ffff7ddaf40 <environ>:       0xffffe468

(gdb) x $esi
```

```
0xffffffffffffe458:     Cannot access memory at address 0xffffffffffffe458
```

And we get a message that we cannot access memory at the specified address. This is the clue right there, problem solved. We tried fondling memory that is not to be fondled. As to why we breached our allocation and how we can know that, we will learn soon.

Not So Simple Example

Now, we do something more complex. We'll create a dynamic array called pointer. We'll use the standard malloc subroutine for this. We will then loop, incrementing *i* values by 1 every iteration, and then let pointer exceed its allowed memory space, also known as *heap overflow*. Understandable as a lab case, but let's see this happen in real life and how we can handle problems like these. Most importantly, we will learn additional gdb commands.

Here's the source:

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

main()
{
   int *pointer;
   int i;
   pointer = malloc(sizeof(int));
   for (i = 0; 1; i++)
   {
      pointer[i]=i;
      printf("pointer[%d] = %d\n", i, pointer[i]);
   }
   return(0);
}
```

Let's compile:

```
gcc -g seg.c -o seg
```

When we run it, we see something like this:

```
./seg
...
pointer[33785] = 33785
pointer[33786] = 33786
pointer[33787] = 33787
Segmentation fault
```

Now, before we hit gdb and assembly, let's try some normal debugging. Let's say you want to try to solve the problem with one of the standard system admin and troubleshooting tools like strace. After having heard of strace earlier, you know the tool's worth and you want to attempt the simple steps first. Indeed, strace works well in most cases. But here, it's of no use.

```
15715 write(1, "pointer[33784] = 33784\n", 23) = 23
15715 write(1, "pointer[33785] = 33785\n", 23) = 23
15715 write(1, "pointer[33786] = 33786\n", 23) = 23
15715 write(1, "pointer[33787] = 33787\n", 23) = 23
15715 --- SIGSEGV (Segmentation fault) @ 0 (0) ---
15715 +++ killed by SIGSEGV +++
```

Nothing useful there, really. In fact, no classic tool will give you any indication what happens here. So we need a debugger, gdb in our case. Load the program.

```
gdb /tmp/seg
```

Breakpoint

Like before, we set a breakpoint. However, using main() is not going to be good for us, because the program will enter main() once and then loop, never going back to the set breakpoint. So we need something else. We need to break in a specific line of code.

To determine the best place, we could run the code and try to figure out where the problem occurs. We can also take a look at our code and make an educated guess. This should be somewhere in the for loop of course. So perhaps, the start of it?

```
(gdb) break 10

Breakpoint 1 at 0x4005a9: file /tmp/seg.c, line 10.
```

Condition

All right, but this is not good enough. We will have a breakpoint at every entry to our loop, and from the execution run, we see there are going to be some 30K+ iterations. We cannot possibly manually type cont and hit Enter every time. So we need a condition, an if statement that will break only if a specific condition is met.

From our sample run, we see that the problem occurs when *i* reaches the value of 33787, so we'll place a conditional break some one or two loop iterations before that. Conditions are set per breakpoint. Notice the breakpoint number, after it is set, because we need that number to set a condition.

```
break 10
Breakpoint 1 at ...
```

And then:

```
(gdb) condition 1 i == 33786

(gdb) info breakpoint

Num     Type           Disp Enb Address            What

1       breakpoint     keep y   0x00000000004005a9 in main at /tmp/seg.c:10

        stop only if i == 33786
```

If you had multiple breakpoints and you wanted to set multiple conditions, then you would invoke the correct breakpoint number. Now we're ready to roll; hit run and let the for loop churn for a while.

```
pointer[33782] = 33782

pointer[33783] = 33783

pointer[33784] = 33784

pointer[33785] = 33785



Breakpoint 1, main () at /tmp/seg.c:11

11            pointer[i]=i;

(gdb)
```

Now we walk through the code, step by step using the next command.

```
Breakpoint 1, main () at /tmp/seg.c:11

11            pointer[i]=i;

(gdb) next

12            printf("pointer[%d] = %d\n", i, pointer[i]);

(gdb)

pointer[33786] = 33786

9          for (i = 0; 1; i++)

(gdb)

13         }

(gdb)

11            pointer[i]=i;

(gdb)

12            printf("pointer[%d] = %d\n", i, pointer[i]);

(gdb)

pointer[33787] = 33787

9          for (i = 0; 1; i++)

(gdb)

13         }

(gdb)
```

```
11            pointer[i]=i;

(gdb)



Program received signal SIGSEGV, Segmentation fault.

0x00000000004005bc in main () at /tmp/seg.c:11

11            pointer[i]=i;

(gdb)
```

We know the problem occurs after pointer[i]=i is set, when the *i* value is 33787. Which means we will rerun the program and then stop just short of executing the pointer[i]=i line of code after a successful print of pointer[33787] = 33787. Now, the next time we reach this point, we create the assembly dump.

```
(gdb) disassemble

Dump of assembler code for function main:

   0x000000000040058c <+0>:     push   %rbp

   0x000000000040058d <+1>:     mov    %rsp,%rbp

   0x0000000000400590 <+4>:     sub    $0x10,%rsp

   0x0000000000400594 <+8>:     mov    $0x4,%edi

   0x0000000000400599 <+13>:    callq  0x400478 <malloc@plt>

   0x000000000040059e <+18>:    mov    %rax,-0x10(%rbp)

   0x00000000004005a2 <+22>:    movl   $0x0,-0x4(%rbp)

=> 0x00000000004005a9 <+29>:    mov    -0x4(%rbp),%eax

   0x00000000004005ac <+32>:    cltq

   0x00000000004005ae <+34>:    shl    $0x2,%rax

   0x00000000004005b2 <+38>:    mov    %rax,%rdx

   0x00000000004005b5 <+41>:    add    -0x10(%rbp),%rdx

   0x00000000004005b9 <+45>:    mov    -0x4(%rbp),%eax

   0x00000000004005bc <+48>:    mov    %eax,(%rdx)

   0x00000000004005be <+50>:    mov    -0x4(%rbp),%eax

   0x00000000004005c1 <+53>:    cltq

   0x00000000004005c3 <+55>:    shl    $0x2,%rax

   0x00000000004005c7 <+59>:    add    -0x10(%rbp),%rax

   0x00000000004005cb <+63>:    mov    (%rax),%edx
   0x00000000004005cd <+65>:    mov    -0x4(%rbp),%esi

   0x00000000004005d0 <+68>:    mov    $0x4006e4,%edi

   0x00000000004005d5 <+73>:    mov    $0x0,%eax

   0x00000000004005da <+78>:    callq  0x400468 <printf@plt>

   0x00000000004005df <+83>:    addl   $0x1,-0x4(%rbp)

   0x00000000004005e3 <+87>:    jmp    0x4005a9 <main+29>

End of assembler dump.

(gdb)
```

We know the problem occurs at offset 4005bc, where we mov %eax value into %rdx. This is similar to what we saw earlier. But we need to understand what happens before that, one or two instructions back.

```
   0x00000000004005bc <+48>:    mov    %eax,(%rdx)
```

Stepping through Assembly Dump

To this end, we will use the stepi command, which can walk the assembly dump, line by line. It's like next in a way, but you can control individual registers, so to speak. Take a look at the dump. The last line in the dump is the jump (jmp) instruction back to offset <main+29>, which brings us to mov 0x00000000004005a9 (%rbp), %eax. This is effectively our for loop. Now, when we hit stepi, we will execute line 4005ac. I omitted the line that reads cltq, because it merely extends the 2-byte EAX into a 4-byte value. That's because we're on a 64-bit system.

```
(gdb) stepi

0x00000000004005ac      11            pointer[i]=i;

(gdb) stepi

0x00000000004005ae      11            pointer[i]=i;
```

Now, we have several lines where the *i* value is incremented. But the crucial line is just one short of the segmentation fault. We need to understand what's inside those registers or whether we can access them at all.

```
(gdb) stepi

0x00000000004005b9      11            pointer[i]=i;

(gdb) stepi

0x00000000004005bc      11            pointer[i]=i;

(gdb) stepi



Program received signal SIGSEGV, Segmentation fault.

0x00000000004005bc in main () at /tmp/seg.c:11
```

```
11          pointer[i]=i;

(gdb)
```

And it turns out we can't. It's like we had earlier. But why? How can we know that this address is off limits? How do we know that?

Proc Mappings

In Linux, you can view the memory maps of any process through /proc/<pid>/maps, as we have learned earlier. It is important to understand what a sample output provides before we can proceed. Let’s recap briefly:

```
#cat /proc/self/maps | grep -iv lc

00400000-0040b000 r-xp 00000000 08:02 248                              /bin/cat

0060a000-0060b000 r--p 0000a000 08:02 248                              /bin/cat

0060b000-0060c000 rw-p 0000b000 08:02 248                              /bin/cat

0060c000-0062d000 rw-p 00000000 00:00 0                                [heap]

7ffff7a67000-7ffff7bd4000 r-xp 00000000 08:02 22                       /lib64/libc-2.11.3.so

7ffff7bd4000-7ffff7dd4000 ---p 0016d000 08:02 22                       /lib64/libc-2.11.3.so

7ffff7dd4000-7ffff7dd8000 r--p 0016d000 08:02 22                       /lib64/libc-2.11.3.so

7ffff7dd8000-7ffff7dd9000 rw-p 00171000 08:02 22                       /lib64/libc-2.11.3.so

7ffff7dd9000-7ffff7dde000 rw-p 00000000 00:00 0

7ffff7dde000-7ffff7dfd000 r-xp 00000000 08:02 788                      /lib64/ld-2.11.3.so

7ffff7fd7000-7ffff7fda000 rw-p 00000000 00:00 0

7ffff7ff3000-7ffff7ffa000 r--s 00000000 08:05 238067                   /usr/lib64/gconv/gconv-modules.cache

7ffff7ffa000-7ffff7ffb000 rw-p 00000000 00:00 0

7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0                        [vdso]

7ffff7ffc000-7ffff7ffd000 r--p 0001e000 08:02 788                      /lib64/ld-2.11.3.so

7ffff7ffd000-7ffff7ffe000 rw-p 0001f000 08:02 788                      /lib64/ld-2.11.3.so

7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0

7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                        [stack]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                [vsyscall]
```

The first line is the code (or text), the actual binary instructions. The second line shows data, which stores all initialized global variables. The third section is the heap, which is used for dynamic allocations, like malloc. Sometimes, it also includes the .bss segment, which stores statically linked variables and uninitialized global variables. When the .bss segment is small, it can reside inside the data segment.

After that, you get shared libraries, and the first one is the dynamic linker itself. Finally, you get the stack. The two last lines are the Linux gating mechanisms for fast system calls, which replace the int 0x80 system call that was used in the past. As you may notice, there are still more memory addresses above the last line, reserved by the kernel.

So here, at a glance, you can examine how your process resides in the memory. When a program is executed through gdb, you can view its memory allocations using the info proc mappings command.

```
gdb) info proc mappings
process 44322
cmdline = '/tmp/seg'
cwd = '/tmp'
exe = '/tmp/seg'
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000          0                                 /tmp/seg
            0x600000           0x601000     0x1000          0                                 /tmp/seg
            0x601000           0x602000     0x1000     0x1000                                 /tmp/seg
            0x602000           0x623000    0x21000          0                                   [heap]
      0x7ffff7a67000     0x7ffff7bd4000   0x16d000          0                          /lib64/libc-2.11.3.so
      0x7ffff7bd4000     0x7ffff7dd4000   0x200000   0x16d000                          /lib64/libc-2.11.3.so
      0x7ffff7dd4000     0x7ffff7dd8000     0x4000   0x16d000                          /lib64/libc-2.11.3.so
      0x7ffff7dd8000     0x7ffff7dd9000     0x1000   0x171000                          /lib64/libc-2.11.3.so
      0x7ffff7dd9000     0x7ffff7dde000     0x5000          0
      0x7ffff7dde000     0x7ffff7dfd000    0x1f000          0                         /lib64/ld-2.11.3.so
      0x7ffff7fd7000     0x7ffff7fda000     0x3000          0
      0x7ffff7ff9000     0x7ffff7ffb000     0x2000          0
      0x7ffff7ffb000     0x7ffff7ffc000     0x1000          0                           [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x1e000                         /lib64/ld-2.11.3.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x1f000                         /lib64/ld-2.11.3.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000          0
      0x7ffffffde000     0x7ffffffff000    0x21000          0                           [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000          0                   [vsyscall]
```

Three lines are of interest: code, data, and heap. And for heap, we can see that the end address is 0x623000. We can't use that, so we get the segmentation fault. Back to C code; we will need to figure out what we did wrong.

```
          Start Addr           End Addr       Size     Offset objfile

            0x400000           0x401000     0x1000          0                                 /tmp/seg

            0x600000           0x601000     0x1000          0                                 /tmp/seg

            0x601000           0x602000     0x1000     0x1000                                 /tmp/seg

            0x602000           0x623000    0x21000          0                                   [heap]
```

We need to start counting bytes. In general, we use a single page for code, because our executable is small. We use a single page for data. And then, there's some heap space, a total of 0x21000, which is 132KB or more specifically 135168 bytes.

On the other hand, we ran through 33788 iterations of the for loop, each 4 bytes in size, as we're on a 64-bit system. Not 33787 as you may assume from the print output in our program run, but one more, because we started counting *i* at value 0.

So we get 135152 bytes, which is 16 bytes less that our heap. So you may ask, where did the extra 16 bytes go? Well, we can use the examine command again and check more accurately what happens at the start address.

```
(gdb) x /8xw 0x602000

0x602000:       0x00000000      0x00000000      0x00000021      0x00000000

0x602010:       0x00000000      0x00000001      0x00000002      0x00000003

(gdb)
```

We print eight 4-byte hexadecimal values. The first 16 bytes are the heap header and the count starts at address 0x501010. So we're all good here, and we know why we got our nasty segmentation fault. We can examine our source code and try to figure out what we did wrong. Two examples, two problems solved.

Other Useful Commands

When working with application cores, there are several other useful commands you may want to use.

The show command lets you show contents, as simple as that. The set command lets you configure variables. For example, you may want to see the initial arguments your program started with and then change them. In our heap overflow example, we could try altering the value of *i* to see if that affects the program.

```
(gdb) show args

Argument list to give program being debugged when it is started is "".

(gdb)
```

And:

```
(gdb) set args Chapter 6

(gdb) show args
```

```
Argument list to give program being debugged when it is started is "Chapter 6".
(gdb)
```

The syntax for setting variables is quite simple. For instance, set i=4. You can also set registers, but don't do this if you don't know what you're doing. The list command lets you dump your code. You can list individual lines, specific functions, or entire code. By default, you get ten lines printed, sort of like tail.

```
(gdb) list
77      #else
78
79      /* This is a "normal" system call stub: if there is an error,
80         it returns -1 and sets errno.  */
81
82      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
83              ret
84      T_PSEUDO_END (SYSCALL_SYMBOL)
85
86      #endif
```

Another thing you may want to do is inspect stack frames in detail. We're already familiar with the info command, so what we need now is to invoke it for specific frames, as listed in the backtrace (bt) command. In our heap overflow example, there's only a single frame.

We break in main, run, display the backtrace, and then check info frame 0, as shown in the screenshot below. You get a wealth of information, including the instruction pointer (RIP), the saved instruction pointer from a previous frame, the address and the list of arguments, the address and the list of local variables, the previous stack pointer, and saved registers.

```
(gdb) break main
Breakpoint 1 at 0x400594: file /tmp/seg.c, line 8.
(gdb) run
Starting program: /tmp/seg


Breakpoint 1, main () at /tmp/seg.c:8
warning: Source file is more recent than executable.
8          pointer = malloc(sizeof(int));
(gdb) bt
```

```
#0  main () at /tmp/seg.c:8

 (gdb) info frame 0

Stack frame at 0x7fffffffe3a0:

 rip = 0x400594 in main (/tmp/seg.c:8); saved rip 0x7ffff7a85c16

 source language c.

 Arglist at 0x7fffffffe390, args:

 Locals at 0x7fffffffe390, Previous frame's sp is 0x7fffffffe3a0

 Saved registers:

 rbp at 0x7fffffffe390, rip at 0x7fffffffe398

(gdb)
```

We mentioned backtrace (bt) earlier, and indeed, it is a most valuable command and best used when you don't know what your program is doing. External commands can be executed using the shell command. For instance, showing the /proc/PID/maps can also be done by using the shell cat /proc/PID/maps instead of info proc mappings as we did before. If for some reason you cannot use either, then you might want to resort to readelf to try to decipher the binary. Just as we used next and stepi, you can use nexti and step. Let's not forget finish, jump, until, and call. The whatis command lets you examine variables.

This article is based on material found in the book *Problem-solving in High Performance Computing* by Will Arthur and David Challener. Visit the Intel Press web site to learn more about this book: https://noggin.intel.com/content/problem-solving-high-performance-computing.

Also see our Recommended Reading List for similar topics: https://noggin.intel.com/recommended-reading

About the Authors

Igor Ljubuncic

Igor Ljubuncic is a technical lead of a global Linux solutions team in Intel IT's Engineering Computing division. Igor has ten years of experience in the hi-tech industry, half of which he has spent as a physicist, focusing on image and signal processing and complex problem solving using statistical engineering and design of experiments methodologies. He has considerable experience in notable areas, like kernel crash debugging and analysis, performance optimization and system internals. In the recent years, Igor focuses on exploring and developing new technologies in the compute space. Igor holds a BA in Physics, and half a dozen industry certifications, including Six Sigma Green Belt, LPIC-1, GSEC, Cloudera Hadoop, and others. Igor has a prolific publication portfolio, including both technical and fiction books, articles in leading technical journals and magazines, numerous whitepapers, and open-source projects. Igor's Linux-oriented personal blog garners close to 1.5 million views every month from loyal readers.

Ravi A. Giri

Ravi A. Giri is a Solutions Architect in Intel IT's Engineering Computing division. He has ~14 years of experience in the industry and is responsible for the architecture and design of cloud computing solutions for Intel's product design teams and is a senior technologist based in Intel India, with numerous whitepapers and publications. He led Intel's Global Datacenter Monitoring and Automation effort as part of Intel's IT Datacenter Strategy where he has conceptualized &

developed several global computing metrics platforms, one of which won the InfoWorld 'Green 15' award in 2010 for enabling energy efficiency gains. He is currently the technical lead for the build out of Intel's next generation cloud computing environment for Intel's globally distributed Silicon design computing.