**Simon Goda, Doulos Embedded, 2014**

A memory scribble or some other access violation in user space is likely to cause an undefined instruction or a data abort exception which will trigger a "SIGSEGV" segmentation fault, e.g.:

target$ ./my\_app

Segmentation fault

target$

The basic information provided does not provide any clues as to what the problem is. In this article we take a brief look at some of the tools there available in a typical embedded Linux development environment which can help to track down the problem. We're assuming here embedded Linux running on a target board but these techniques could also be used for a host application.

## **Kernel Messages**

First check the kernel messages in the serial console connected to the target, or use dmesg directly on the target to retrieve these. Look for messages related to the problem application:

target$ ./my\_app

Segmentation fault

target$ dmesg

...

[ 1962.987529] myapp[3303]: segfault at 0 ip 00400559 sp 5bc7b1b0 error 6 in myapp[400000+1000]

...

Here we can see some information about the cause of the fault i.e. the instruction pointer address (ip) and the stack pointer address (sp). We can also see that the segfault has occurred at address 0.

## **Static Analysis**

There are some useful utilities you can use to make sense of this information by performing a static analysis of the application binary. In your toolchain the name of the tool may well be prefixed to indicate the target architecture they support e.g. arm-none-linux-gnueabi-**nm**.

**nm**

nm will provide a listing of all of the symbols and their addresses in the application binary file:

target$ nm myapp

00000000 a

00601034 b .bss

00601034 B \_\_bss\_start

00000000 n .comment

00601034 b completed.6366

00000000 a crtstuff.c

00000000 a crtstuff.c

00601030 d .data

00601030 D \_\_data\_start

00601030 W data\_start

...

Depending on which toolchain and binary file format you are using, it might be more use to compile the binary (if not done already) with the -g option. You can then use the nm -a option to ensure that any debug symbols are listed. We can then use the instruction pointer value from the dmesg output and see if we can understand which symbol it corresponds to.

**ldd**

ldd will show any dependencies on shared libraries, including their start address. Comparing the instruction pointer with these start addresses should show whether the problem is in fact in a shared library.

target$ ldd myapp

linux-vdso.so.1 => (0x7b5fe000)

libc.so.6 => /lib/libc.so.6 (0x8f400000)

/lib/ld-linux-armv7.so.2 (0x8ec00000)

**objdump**

objdump can be used to display a range of information about an object file. It is a very powerful tool which has numerous options (use the man page or --help). In this scenario we can use it to find the relevant line of assembly code for the instruction pointer address given in the dmesg output and see what instructions actually caused the problem. Here we use options -D (disassemble all) and -S (intermix source). This latter option is particularly useful but does require the application to have been compiled with debug symbols i.e. -g. We pipe the output into a file:

target$ objdump -DS myapp > dump.txt

Looking in that file we can see the instructions at the relevant ip address. Of course, some knowledge of the instruction set and architecture you are working with is required at this point to understand the disassembly. It may also help to compile without optimisations enabled as these can obfuscate the true functionality of the code:

...

40054f: 00

myfunction();

400550: e8 db ff ff ff callq 400530

/\* causing seg fault \*/

\*myptr = 4;

400555: 48 8b 45 f8 mov -0x8(%rbp),%rax

400559: c7 00 04 00 00 00 movl $0x4,(%rax)

...

The problem is obvious here, and if we refer back to our original dmesg output which told us that the segfault occurred at address 0 then it is not surprising to know that myptr was initialized to NULL.

## **Dynamic Debugging with GDB**

Another approach is to work 'dynamically' by using GDB (or your chosen debugger) to debug the faulty application. This could be done natively on the target or using gdbserver to debug remotely from the host. In both cases the application should be compiled with debug symbols enabled (-g) and, if possible, with all compilation optimization switched off.

We can then run the application in the debugger and get very useful information about the failure, particularly if GDB can find the corresponding source code:

$ gdb myapp

...

Reading symbols from /home/user/test/myapp/myapp...done.

(gdb) run

Starting program: /home/user/test/myapp/myapp

Starting Application

Program received signal SIGSEGV, Segmentation fault.

0x00400559 in main () at myapp.c:15

15 \*myptr = 4;

(gdb)

Here we have simply 'run' the program to see the source of the problem. In more complex applications you can debug more thoroughly using breakpoints, line by line stepping etc. to get a clearer picture of what is going wrong.

If this kind of 'live' debugging is not possible then another option is to generate a core dump file from the application which can then be loaded into GDB. The core dump file will contain useful information like the values of various system registers, the contents of memory etc.

To generate a core dump for the application, first set the maximum size of the core dump file e.g.

target$ ulimit -c 1024

Then run the app:

target$ ./myapp

Starting Application

Segmentation fault (core dumped)

target$ ll

...

-rw-------. 1 user user 253952 May 8 16:11 core.26593

...

You can then load the information into GDB:

target$ gdb myapp core.26593

...

Reading symbols from /home/user/test/myapp/myapp...done.

[New LWP 26593]

Core was generated by `./myapp'.

Program terminated with signal SIGSEGV, Segmentation fault.

#0 0x00400559 in main () at myapp.c:15

15 \*myptr = 4;

As before, in this trivial example the problem is obvious but it is possible to debug further in more complex cases by, for example, inspecting the system registers:

(gdb) info registers

...

r14 0x00

r15 0x00

rip 0x400559 0x400559 <main+25 style="margin: 0px; padding: 0px; background-repeat: no-repeat;">

...

or inspect the contents of memory at a particular address, e.g.:

(gdb) x/8b 0xfe2c1a30

0xfe2c1a30: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

An advantage of core dumps is that you don't need to have access to the running system to be able to debug the problem. The core dump file, the binary file and the relevant sources can be used independent of the target, as long as the appropriate tools are installed.