debugging

Abdul Rahim

24 July 2024

Asking the following questions can help to understand and identify the nature of the problem and how best to solve it:

- Is the problem easily reproducible?
- Is there a reproducer or test that can trigger the bug consistently?
- Are there any panic, or error, or debug messages in the dmesg when the bug is triggered?
- Is reproducing the problem time-sensitive?

Easily reproducible bugs with a test to trigger make it easier to debug, identify the problem, fix it and verify the fix. Time-sensitive problems could be a result of race conditions, and these are harder to debug and fix.

kernel panic

- kernel panic
  - A kernel panic is an action taken by an operating system upon detecting an internal fatal error from which it cannot safely recover and force the system to do a **controlled system hang / reboot** due to a detected **run-time system malfunction** (not necessarily an OOPS). The operation of the panic kernel feature may be controllable via **run-time sysconfig settings*** such as **hung task handling**.
- oops
  - oops are due to kernel exception handler getting executed including maros such as BUG ( ) which is defined as a invalid instruction. each exception has a unique number. Many oops result in kernel panic.

types of panics

panics are classified as:

1. hard panics: Aiee
2. soft panics: Oops

oops

when a kernel oops is encountered in a runnning kernel an OOPS message like:

```
1  [67.994624] Internal error: Oops: 5 [#1]
       XXXXXXXXXXXi
```

is displayed in the screen. An oops message contains:

- values of CPU registers
- address of the dunction that invoked the failure i.e.
  - PC(Program Counter)
  - stack
  - name of current process executing

how to debug oops

in linux `System.map` is a symbol table used by the kernel. System.map is required when symbol of an address or address of a symbol is required. The kernel does address to name translation when CONFIG_KALLSYMS is enabled so that tools like ksymoops are not required.

Note in the kernel backtraces in the logs, the kernel finds the nearest symbol to the address being analysed. Not all function symbols are available because of inlining, static, and optimisation so sometimes the reported function name is not the location of the failure.

using kernel stacktrace and objdump

1. check kernel stack trace, figure out the function address where the problem is occuring
2. run `objdump` on vmlinux and find out the instruction near the function. See the instructions near; verify the addresses match and find out the cause

using gdb

1. identify the line of code from panic/oops message;
2. run gdb list command; which will tell you the exact line where the error occured

```
(gdb) list *(function+0xoffset)
```

references

1. *panics*
2. *creating kernel oops*

decode stacktrace script

Panic messages can be decoded using the `decode_stacktrace.sh` tool.

```
Usage:
        scripts/decode_stacktrace.sh -r <release
            > | <vmlinux> [base path] [modules
            path]
```

Save (cut and paste) the panic trace in the dmesg between the two following lines of text into a .txt file.

```
------------[ cut here ]------------
------------[ end trace ]------------
```

Run this tool in your kernel repo. You will have to supply the [base path], which is the root of the git repo where the vmlinux resides if it is different from the location the tool is run from. If the panic is in a dynamically loaded kernel module, you will have to pass in the [modules path] where the modules reside.

```
scripts/decode_stacktrace.sh ./vmlinux <
   panic_trace.txt
```

It goes without saying that reading code and understanding the call trace leading up to the failure is an essential first step to debugging and finding a suitable fix.