Kernel Debugging

Team Emertxe

Introduction to Kernel Debugging

- Kernel debugging is critical for diagnosing issues in Linux kernel code.
- It involves identifying and fixing errors in:
    - Device drivers
    - System calls
    - Kernel modules

Challenges:

- No direct access to kernel memory
- Crashes can make systems unresponsive
- Requires specialized tools and methods

Debugging Methods

Debugging Methods Overview

1. **Print-Based Debugging** (using printk)
2. **Crash Dump Analysis** (using tools like kdump, crash)
3. **Dynamic Probes** (Kprobes, Uprobes, Ftrace)
4. **Interactive Debugging** (using KGDB, GDB)
5. **Static Code Analysis** (tools like sparse, smatch)
6. **Real-Time Debugging** (using perf, trace-cmd)
7. **Address2line Method**

Kernel Logs and Printk

- printk() is the most common method for kernel debugging.
- Logs are available in /var/log/messages or dmesg.

Advantages:

- Simple to use
- Works in most environments

Disadvantages:

Requires code modifications

Not suitable for time-sensitive bugs

Example:

```
printk(KERN_INFO "Driver loaded successfully\n");
```

EMERTXE

Crash Dump Analysis

- Used to analyze the state of the kernel when it crashes.
- Tools:
  - **kdump**: Captures kernel memory during a crash
  - **crash**: Analyzes crash dumps

Steps to Enable kdump:

1. Install kdump package
2. Configure /etc/kdump.conf
3. Enable and start kdump service

Example Command:

```
crash /var/crash/vmcore /usr/lib/debug/vmlinux
```

Dynamic Probes

Kprobes:

- Allows you to insert probes into running kernel code.
- Used to gather diagnostic data without recompiling the kernel.

Uprobes:

- Similar to Kprobes but for user-space applications.

Ftrace:

- Provides function tracing within the kernel.

KGDB (Kernel GNU Debugger)

- KGDB allows you to debug the kernel using GDB.
- Requires two machines:
  - **Host**: Runs GDB
  - **Target**: Runs the kernel being debugged

Setup Steps:

1. Enable KGDB in kernel configuration.
2. Connect host and target via serial or network.
3. Start GDB on the host machine.

Static Analysis Tools

Sparse: Identifies common programming errors in kernel code.

Smatch: Static analysis tool tailored for the Linux kernel.

Advantages:

1. Finds issues without running the code.
2. Identifies potential security vulnerabilities.

EMERTXE

Real-Time Kernel Debugging

- Tools like perf, trace-cmd, and systemtap enable real-time debugging.
- Useful for performance analysis and monitoring live systems.

Address2line Method

- address2line converts addresses from a crash dump or dmesg output to the corresponding source file and line number.
- Helps map kernel panic or OOPS addresses to specific code locations.

Steps to Use address2line:

- Identify the address from dmesg or crash dump.
- Use address2line with the kernel image:

Example Command:

```
address2line -e /usr/lib/debug/vmlinux-$(uname -r) 0xffffffff810c33a0
```

Advantages:

- Quickly pinpoints the source of the issue.
- No need to modify the kernel code.

EMERTXE

Kernel Debugging Using QEMU and GDB

Introduction to GDB

The **GNU Debugger (GDB)** is a powerful tool used to debug programs by allowing you to:

- Set breakpoints
- Inspect variables
- Step through code
- Analyze crashes

For **kernel debugging**, GDB can connect to a running QEMU virtual machine to debug the kernel code in real-time.

GDB for Kernel Debugging

When using GDB for kernel debugging:

- You need to run the Linux kernel inside QEMU with debugging enabled.
- GDB connects to QEMU using a remote protocol (gdbserver).
- You can set breakpoints and tracepoints inside the kernel code and custom drivers.

Common GDB Commands

- gdb <binary>: Start GDB with a specified binary.
- target remote :1234: Connect GDB to a remote target (QEMU in this case).
- break <function>: Set a breakpoint at a function.
- continue: Resume program execution.
- next: Step to the next line of code.

What is QEMU?

QEMU is an open-source emulator and virtualizer that allows you to:

- Run virtual machines on your host system.
- Emulate different architectures (x86, ARM, etc.).
- Connect GDB to a running virtual machine for debugging purposes.

Why use QEMU for kernel debugging?

- No need for physical hardware.
- Easy to set up and configure.
- Supports different architectures (x86, ARM, etc.).

Preparing Your Host System

```
sudo apt update

sudo apt install build-essential libncurses-dev
bison flex libssl-dev libelf-dev qemu qemu-
system gdb

sudo apt install qemu qemu-kvm qemu-system-x86

qemu-system-x86_64 --version
```

Linux Kernel - Setup

```
mkdir ~/kernel_debug

cd ~/kernel_debug

wget
https://cdn.kernel.org/pub/linux/kernel/v5.x/lin
ux-5.15.tar.xz

tar -xf linux-5.15.tar.xz

cd linux-5.15
```

Configure the Kernel for QEMU

make x86_64_defconfig

make menuconfig

Enable debugging options:

- Kernel hacking → Compile the kernel with debug info
- Kernel hacking → Enable kgdb
- Kernel hacking → Magic SysRq key

Save and exit.

Compile the Kernel

make -j$(nproc)

Transfer the Kernel to QEMU

The compiled kernel image will be located at:

arch/x86/boot/bzImage

Transfer the Kernel to QEMU

The bzImage file needs to be transferred to the directory from where QEMU will be launched. For simplicity, copy it to your ~/kernel_debug folder:

```
cp arch/x86/boot/bzImage ~/kernel_debug/
```

Root Filesystem (Rootfs) for QEMU

A root filesystem is required for the Linux kernel to boot properly. You can use a prebuilt root filesystem or create your own using buildroot.

Option 1: Download a Prebuilt Rootfs

Option 2: Create Your Own Rootfs Using Buildroot/Yocto

Option 3: Creating an Initramfs using mkinitramfs

Starting Linux with QEMU

Start QEMU

cd ~/kernel_debug

```
qemu-system-x86_64 -kernel bzImage -initrd
initramfs.cpio.gz -append "console=ttyS0 root=/dev/ram
rdinit=/bin/sh" -nographic -s -S
```

Explanation:

-nographic: Disables graphical output.

-s: Opens port 1234 for GDB.

-S: Stops the CPU at startup (waiting for GDB).

-initrd initramfs.cpio.gz: Specifies the initramfs image.

Connect GDB to QEMU

Open another terminal and run:

- `cd ~/kernel_debug`
- `gdb vmlinux`
- `target remote :1234`

You can pass the following arguments to QEMU for debugging:

```
-append "console=ttyS0 root=/dev/ram
rdinit=/bin/bash"
```

Breakpoints and Tracepoints in GDB

To Set a Breakpoint:

- break start_kernel
- continue

To Inspect Variables:

- print <variable_name>

Using Yocto - runqemu

What is QEMU?

- QEMU is an emulator that lets you run a virtual machine with a different architecture from your host system.
- Yocto integrates with QEMU to let you test your built images without real hardware.

What is runqemu?

runqemu is a Yocto tool that simplifies launching a virtual machine using QEMU with a Yocto-built Linux image.

Benefits of runqemu

- Fast and easy way to test your Yocto images.
- Automatically handles image and kernel selection.
- Supports network configurations and file sharing.

Basic runqemu Command

Syntax:

```
runqemu <image> <machine> <options>
```

Example:

```
runqemu core-image-minimal qemux86-64 nographic
```

Explanation:

- core-image-minimal: The image you built with Yocto.
- qemux86-64: The machine architecture.
- nographic: Run without a graphical interface.

Passing Extra Parameters to QEMU

Using qemuparms to Pass Custom QEMU Options

```
runqemu core-image-minimal qemux86-64 nographic
qemuparms="-s -S"
```

Explanation:

- -s: Starts QEMU's GDB server on port 1234.
- -S: Stops the virtual CPU at startup, waiting for GDB to connect.