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");
```

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.

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.

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
```

EMERTXE

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.

Kdump Setup and Crash Analysis

What is Crash Analysis?

Crash analysis involves diagnosing and identifying the root cause of kernel panics or system crashes by analyzing memory dumps created during a crash. It is a critical process in debugging and improving system stability.

Importance of Crash Analysis

- **Improved Stability**: Pinpoints bugs or vulnerabilities in the kernel.
- **Debugging Custom Kernels**: Essential for developers working on custom kernel configurations.
- **Root Cause Analysis**: Helps trace issues in hardware, drivers, or kernel modules.
- **System Hardening**: Strengthens the system by eliminating critical bugs.

Prerequisites for Crash Analysis

1. **Kernel Configuration**: Enable crash dump features like CONFIG_KEXEC and CONFIG_CRASH_DUMP.
2. **Debug Symbols**: Use an unstripped vmlinux file to provide symbol information.
3. **Sufficient Memory**: Reserve memory for the crash kernel. Recommended to keep 4 times RAM size for /var/crash
4. **Tools**: Install kdump-tools, makedumpfile, and crash tools.

EMERTXE

How Crash Analysis Works

Crash Trigger: When a kernel panic occurs, the system switches to a reserved crash kernel (configured via kdump).

Crash Kernel Role: This lightweight crash kernel captures the memory of the running system and saves it as /proc/vmcore.

Dump File Creation: Tools like makedumpfile process /proc/vmcore and save it as a compressed dump file in /var/crash.

Analysis: The crash tool reads the dump file along with the unstripped vmlinux to analyze the root cause of the crash.

System Restart Twice

- **First Reboot**: The initial reboot transitions the system to the crash kernel. This kernel is minimal and captures the crash dump.
- **Second Reboot:** After saving the dump file, the system reboots into the primary kernel for normal operation.

Kdump Setup on Ubuntu

Verify Kernel ConfigurationEnsure the following kernel options are enabled in your custom kernel configuration file (.config):

```
CONFIG_KEXEC=y
CONFIG_KEXEC_FILE=y
CONFIG_KEXEC_CORE=y
CONFIG_CRASH_DUMP=y
CONFIG_PROC_VMCORE=y
```

After verifying the configuration, rebuild and install your kernel:

Kdump Setup on Ubuntu

After verifying the configuration, rebuild and install your kernel:

```
make -j$(nproc)
sudo make modules_install
sudo make install
```

Configure GRUB

Edit your GRUB configuration to reserve memory for the crash kernel.Edit the /etc/default/grub file:

```
sudo nano /etc/default/grub
```

Add the following:

```
GRUB_CMDLINE_LINUX="crashkernel=512M@0x20000000
nokaslr‟
```

Update GRUB:

```
sudo update-grub
```

Reboot the system:

Verify Crashkernel Reservation

verify that the crash kernel memory is reserved:

```
sudo dmesg | grep -i crashkernel
cat /proc/iomem | grep -i crash
```

Expected output:

```
[0.012853] crashkernel reserved:
0x0000000020000000 - 0x0000000040000000
(512MB)20000000-3fffffff : Crash kernel
```

Configure Kdump

Edit :

```
sudo vi /etc/kdump.conf
```

Ensure the following:

```
path /var/crash
core_collector makedumpfile -c --message-level
1 -d 15
```

Configure Kdump

Edit :

```
sudo vi /etc/default/kdump-tools
```

Ensure the following:

```
MAKEDUMP_ARGS="-c --message-level 1 -d 15"
```

Configure Kdump

Restart the service:

```
sudo systemctl daemon-reload

sudo systemctl restart kdump-tools
```

Ensure the following:

```
MAKEDUMP_ARGS="-c --message-level 1 -d 15"
```

Verify setup:

```
sudo kdump-config show


current state: ready to kdump
```

Trigger a Crash

To test the kdump setup, trigger a crash using the SysRq trigger:

```
echo c | sudo tee /proc/sysrq-trigger
```

After reboot, check if the dump file is created in:

```
ls /var/crash/
```

Crash Analysis Using

Ensure the crash tool is installed

```
MAKEDUMP_ARGS="-c --message-level 1 -d 15"
```

Run the crash tool:

```
sudo crash /path/to/vmlinux
/var/crash/<timestamp>/dump.<timestamp>
```

You should see the prompt `crash>`

Basic crash tool commands

Command	Description
bt	Show backtrace of the crashed kernel thread
ps	Display the list of processes
vm	Show virtual memory usage
mod	List loaded kernel modules
log	Show kernel messages (dmesg)
files	Show open files for a process
task	Show task structure for a process

Tracing the Root Cause

Check the backtrace:

```
crash> bt
```

Example output:

```
PID: 1    TASK: ffff888109c10000  CPU: 0
COMMAND: "swapper/0" #0 [ffff888109ce04b8]
crash_kexec at ffffffff8105d9d3 #1
[ffff888109ce0570] panic at ffffffff8105e1a5 #2
[ffff888109ce0620] sysrq_handle_crash at
ffffffff810ee0c3 #3 [ffff888109ce0630]
__handle_sysrq at ffffffff810ee46b
```

Tracing the Root Cause

Trace the Code Path:

```
crash> dis panic
```

Other commands:

```
crash> struct sysrq_key_op sysrq_crash_op
crash> log
crash> ps
crash> task PID
crash> mod
```

EMERTXE

Address2line Tool

What is Address2line?

- *address2line* is a command-line tool from the GNU Binutils suite.
- It converts memory addresses from crash dumps or core dumps into human-readable file names, function names, and line numbers.
- Essential for debugging **stripped binaries** and analyzing **core dumps**.

Why Use Address2line?

- Quickly maps memory addresses to source code lines.
- Helps debug core dumps, backtraces, and segmentation faults.
- Crucial for kernel debugging and embedded systems.
- Saves time in identifying crash locations.

How Does Address2line Work?

1. A program crashes and produces a memory address (e.g., 0x400636).
2. Use address2line to map the address to a file name and line number.
3. Example command: **`address2line -e my_program 0x400636`**

Output:

```
/home/user/project/main.c:25
```

This means the crash occurred on line 25 of main.c.

Kernel Debugging

Scenario: Kernel panic logs show memory addresses in the crash report.

Step 1: Check dmesg Logs

```
dmesg | grep "RIP"
```

Example output:

```
[12345.67890] RIP: 0010:[<ffffffff81234567>]
```

Step 2: Use Address2line to Decode the Kernel Address

```
address2line -e /usr/lib/debug/boot/vmlinux-$(uname -r)
ffffffff81234567
```

Output:

```
/home/user/linux/kernel/sched.c:150
```

EMERTXE

Real-World Use Cases

Use Case	Description
Debug core dumps	Map crash addresses to source code lines
Kernel crash analysis	Identify kernel bugs using memory addresses
Debug shared libraries	Locate bugs in shared `.so` files
Debug C++ exceptions	Demangle and decode C++ exception addresses
Embedded systems logs	Map embedded firmware crash logs to source code

Thanks