

Linux Kernel Debugging

Leverage open source tools and advanced techniques to debug
Linux kernel or module issues



Linux Kernel Debugging

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: Linux Kernel Debugging

Early Access Production Reference: B17445

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80107-503-9

www.packt.com

Table of Contents

1. [Linux Kernel Debugging](#)
 - I. [Linux Kernel Debugging](#)
2. [1 A General Introduction to Debugging Software](#)
 - I. [Technical requirements](#)
 - i. [Cloning this book's code repository](#)
 - II. [Software debugging – what it is, origins, and myths](#)
 - III. [Software bugs – a few actual cases](#)
 - i. [Patriot missile failure](#)
 - ii. [ESA's unmanned Ariane 5 rocket](#)
 - iii. [Mars Pathfinder reset issue](#)
 - iv. [The Boeing 737 MAX aircraft – the MCAS and lack of training to the flight crew](#)
 - v. [Other cases](#)
 - IV. [Setting up the workspace](#)
 - i. [Running Linux as a native or guest OS?](#)
 - ii. [Running Linux as a guest OS](#)
 - iii. [Installing the Oracle VirtualBox guest additions](#)
 - iv. [Installing required software packages](#)
 - V. [A tale of two kernels](#)
 - i. [A production and a debug kernel](#)
 - ii. [Setting up our custom production kernel](#)
 - iii. [Setting up our custom debug kernel](#)
 - iv. [Seeing the difference – production and debug kernel config](#)
 - VI. [Debugging – a few quick tips](#)
 - i. [A programmer's checklist – seven rules](#)
 - VII. [Summary](#)
 - VIII. [Further reading](#)
 3. [2 Approaches to Kernel Debugging](#)
 - I. [Technical requirements](#)
 - II. [Classifying bug types](#)
 - i. [Types of bugs – the classic view](#)
 - ii. [Types of bugs – the memory view](#)
 - iii. [Types of bugs – the CVE/CWE security-related view](#)
 - iv. [Types of bugs – the Linux kernel](#)

- III. [Kernel debugging – why there are different approaches to it](#)
 - IV. [Summarizing the different approaches to kernel debugging](#)
 - i. [Development phase](#)
 - ii. [Unit testing and/or QA phases](#)
 - iii. [Categorizing into different scenarios](#)
 - V. [Summary](#)
 - VI. [Further reading](#)
- 4. [3 Debug via Instrumentation – printk and friends](#)
 - I. [Technical requirements](#)
 - II. [The ubiquitous kernel printk](#)
 - i. [Using the printk API's logging levels](#)
 - ii. [Leveraging the pr <foo> convenience macros](#)
 - iii. [Understanding where the printk output goes](#)
 - iv. [Practically using the printk format specifiers – a few quick tips](#)
 - III. [Leveraging the printk for debug purposes](#)
 - i. [Writing debug messages to the kernel log](#)
 - ii. [Debug printing – quick and useful tips](#)
 - iii. [Device drivers – use the dev_dbg\(\)](#)
 - iv. [Trying our kernel module on the custom production kernel](#)
 - v. [Rate limiting the printk](#)
 - IV. [Using the kernel's powerful dynamic debug feature](#)
 - i. [Dynamic debug via module parameters](#)
 - ii. [Specifying what and how to print debug messages](#)
 - iii. [Exercising dynamic debug on a kernel module on a production kernel](#)
 - V. [Remaining printk miscellany](#)
 - i. [Printing before console init – the early printk](#)
 - ii. [Designating the printk to some known presets](#)
 - iii. [Printing exactly once](#)
 - iv. [Emitting a printk from userspace](#)
 - v. [Easily dumping buffer content](#)
 - vi. [Remaining points – bootloader log peeking, LED flashing and more](#)
 - VI. [Summary](#)
 - VII. [Further reading](#)
- 5. [4 Debug via Instrumentation – Kprobes](#)
 - I. [Understanding kprobes basics](#)
 - i. [What we intend to do](#)
 - II. [Using static kprobes – traditional approaches to probing](#)

- i. [Demo 1 – static kprobe – trapping into the file open the traditional static kprobes way – simplest case](#)
 - ii. [Demo 2 – Static kprobe – specifying the function to probe via a module parameter](#)
- III. [Understanding the basics of the Application Binary Interface \(ABI\)](#)
- IV. [Using static kprobes – demo 3 and demo 4](#)
 - i. [Demo 3 – static kprobe – probing the file open and retrieving the filename parameter](#)
 - ii. [Demo 4 – Semi-automated static kprobe via our helper script](#)
- V. [Getting started with kretprobes](#)
 - i. [Kprobes miscellany](#)
- VI. [Kprobes – limitations and downsides](#)
 - i. [Interface stability](#)
- VII. [The easier way – dynamic kprobes or kprobe-based event tracing](#)
 - i. [Kprobe-based event tracing – minimal internal details](#)
 - ii. [Setting up a dynamic kprobe \(via kprobe events\) on any function](#)
 - iii. [Using dynamic kprobe event tracing on a kernel module](#)
 - iv. [Setting up a return probe \(kretprobe\) with kprobe-perf](#)
- VIII. [Trapping into the execve\(\) – via perf and eBPF tooling](#)
 - i. [System calls and where they land in the kernel](#)
 - ii. [Observability with eBPF tools – an introduction](#)
- IX. [Summary](#)
- X. [Further reading](#)
6. [5 Debugging Kernel Memory Issues – Part 1](#)
 - I. [Technical requirements](#)
 - II. [What's the problem with memory anyway?](#)
 - i. [Tools to catch kernel memory issues – a quick summary](#)
- III. [Using KASAN and UBSAN to find memory bugs](#)
 - i. [Understanding KASAN – the basics](#)
 - ii. [Requirements to use KASAN](#)
 - iii. [Configuring the kernel for generic KASAN mode](#)
 - iv. [Bug hunting with KASAN](#)
 - v. [Using the UBSAN kernel checker to find UB](#)
- IV. [Building your kernel and modules with Clang](#)
 - i. [Using Clang 13 on Ubuntu 21.10](#)
- V. [Catching memory defects in the kernel – comparisons and notes \(Part 1\)](#)
 - i. [Miscellaneous notes](#)
- VI. [Summary](#)

- VII. [Further reading](#)
7. [6 Debugging Kernel Memory Issues – Part 2](#)
- I. [Technical requirements](#)
 - II. [Detecting slab memory corruption via SLUB debug](#)
 - i. [Configuring the kernel for SLUB debug](#)
 - ii. [Leveraging SLUB debug features via the slab_debug kernel parameter](#)
 - iii. [Running and tabulating the SLUB debug test cases](#)
 - iv. [Interpreting the kernel's SLUB debug error report](#)
 - v. [Learning to use the slabinfo and related utilities](#)
 - III. [Finding memory leakage issues with kmemleak](#)
 - i. [Configuring the kernel for kmemleak](#)
 - ii. [Using kmemleak](#)
 - iii. [A few tips for developers regarding dynamic kernel memory allocation](#)
 - IV. [Catching memory defects in the kernel – comparisons and notes \(Part 2\)](#)
 - i. [Miscellaneous notes](#)
 - V. [Summary](#)
 - VI. [Further reading](#)
8. [7 Oops! Interpreting the kernel bug diagnostic](#)
- I. [Technical requirements](#)
 - II. [Generating a simple kernel bug and Oops](#)
 - i. [The procmap utility](#)
 - ii. [What's this NULL trap page anyway](#)
 - iii. [A simple Oops v1 – dereferencing the NULL pointer](#)
 - iv. [Doing a bit more of an Oops – our buggy module v2](#)
 - III. [A kernel Oops and what it signifies](#)
 - IV. [Devil in the details – decoding the Oops](#)
 - i. [Line-by-line interpretation of an Oops](#)
 - V. [Tools and techniques to help determine the location of the Oops](#)
 - i. [Using objdump to help pinpoint the Oops code location](#)
 - ii. [Using GDB to help debug the Oops](#)
 - iii. [Using addr2line to help pinpoint theOops code location](#)
 - iv. [Taking advantage of kernel scripts to help debug kernel issues](#)
 - v. [Leveraging the console device to get the kernel log after Oops'ing in IRQ context](#)
 - VI. [An Oops on an ARM Linux system and using netconsole](#)
 - i. [Figuring out the actual buggy code location \(on ARM\)](#)

VII. [A few actual Oops'es](#)

VIII. [Summary](#)

IX. [Further reading](#)

Linux Kernel Debugging

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You'll be notified when a new version is ready.

This title is in development, with more chapters still to be written, which means you have the opportunity to have your say about the content. We want to publish books that provide useful information to you and other customers, so we'll send questionnaires out to you regularly. All feedback is helpful, so please be open about your thoughts and opinions. Our editors will work their magic on the text of the book, so we'd like your input on the technical elements and your experience as a reader. We'll also provide frequent updates on how our authors have changed their chapters based on your feedback.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book. Join the exploration of new topics by contributing your ideas and see them come to life in print.

Linux Kernel Debugging

1. A General Introduction to Debugging Software
2. Approaches to Kernel Debugging
3. Debug via Instrumentation – `printk` and friends
4. Debug via Instrumentation – Kprobes

5. Debugging Kernel Memory Issues – Part 1
6. Debugging Kernel Memory Issues – Part 2
7. Oops! Interpreting the kernel bug diagnostic
8. Lock Debugging
9. Tracing the Kernel Flow
10. Kernel Panic, Hangcheck, and Watchdogs
11. Using KGDB
12. Other Approaches to Kernel Debugging

1 A General Introduction to Debugging Software

Hello there! I welcome you on this, our journey, on learning how to go about debugging a really sophisticated, large, and complex piece of software that's proven absolutely critical to both enterprise business as well as tiny embedded systems and everything in between – **the Linux kernel**, and, to some extent, learn to debug the Linux user space ecosystem as well.

Let's begin this very first chapter, and our journey on kernel (and user mode) debugging, by first understanding a little more on what a **bug** really is, and the origins and myths of the term **debugging**. Next, a glimpse at some actual *real-world* software bugs will (hopefully) provide the required inspiration and motivation (to firstly avoid bugs and then to find and fix your bugs, of course). You will be guided on how to setup an appropriate workspace to actually work on a custom kernel and debug issues, including setting up a full-fledged *debug* kernel. We'll wrap up with some useful tips on debugging.

In this chapter we're going to cover the following main topics:

- Software debugging – what it is, origins, and myths
- Software bugs – a few actual cases
- Setting up the workspace
- Debugging – a few tips

Technical requirements

You will require a modern and powerful desktop or laptop. We tend to use **Ubuntu 20.04 LTS** as the primary platform for this book. Ubuntu desktop specifies the *recommended minimum system requirements* (<https://help.ubuntu.com/community/Installation/SystemRequirements>) for the installation and usage of the distribution; do refer it to verify that your system (even a guest) is up to it.

Cloning this book's code repository

The complete source code for this book is freely available on GitHub at <https://github.com/PacktPublishing/Linux-Kernel-Debugging>. You can work on it by cloning the Git tree using the following command:

```
git clone https://github.com/PacktPublishing/Linux-Kernel-Debugging
```

The source code is organized chapter-wise. Each chapter is represented as a directory in the repository – for example, `ch1/` has the source code for this chapter. A detailed description on installing a viable system is covered in the *Setting up the workspace* section.

Software debugging – what it is, origins, and myths

In the context of the software practitioner, a **bug** is a defect, an error, within the code. A key, and often large, part of our job as software developers is to hunt them down and fix them, so that, as far as is humanely possible, the software is defect-free and runs precisely as designed.

Of course, to fix a bug, you first have to find it. Indeed, with non-trivial bugs, it's often the case that you aren't even aware there is a bug (or several) until some event occurs to expose it! Shouldn't we have a disciplined approach to finding bugs before shipping the product or project? Of course we do – it's the **Quality Assurance (QA)** process, more commonly known as **testing**. Though glossed over at times, testing remains one of the – if not the – most important facets of the software lifecycle (would you voluntarily fly in a new aircraft that's never been tested? Well, unless you're the lucky test pilot...).

Okay, back to bugs; once identified (and filed), your job as a software developer is to now identify what exactly is causing it, what the actual underlying *root cause* is. A large portion of this book is devoted to tools, techniques, and just thinking about how to exactly do this. Once the root cause is identified, and you have clearly understood the underlying issue, you can, in all probability, fix it. Yay!

This process of identifying a bug – using tools, techniques, some hard thinking to figure out its root cause – and then fixing it, is subsumed into the word **debugging**. Without bothering to go into details, there's a popular story regarding the origin of the word debugging: on a Tuesday at Harvard University (on Sept 9, 1947), Admiral Grace Hopper's staff discovered a moth caught in a

relay panel of a Mark II computer. As the system malfunctioned because of it, they removed the moth, thus *de-bugging* the system! Well, as it turns out: one, Admiral Hopper has herself stated that she didn't coin the term; two, its origins seem to be rooted in aeronautics. Nevertheless, the term **debugging** has stuck. The following image shows the picture at the heart of this story - the unfortunate but famous moth that inadvertently caught itself in the system that then had to be de-bugged!

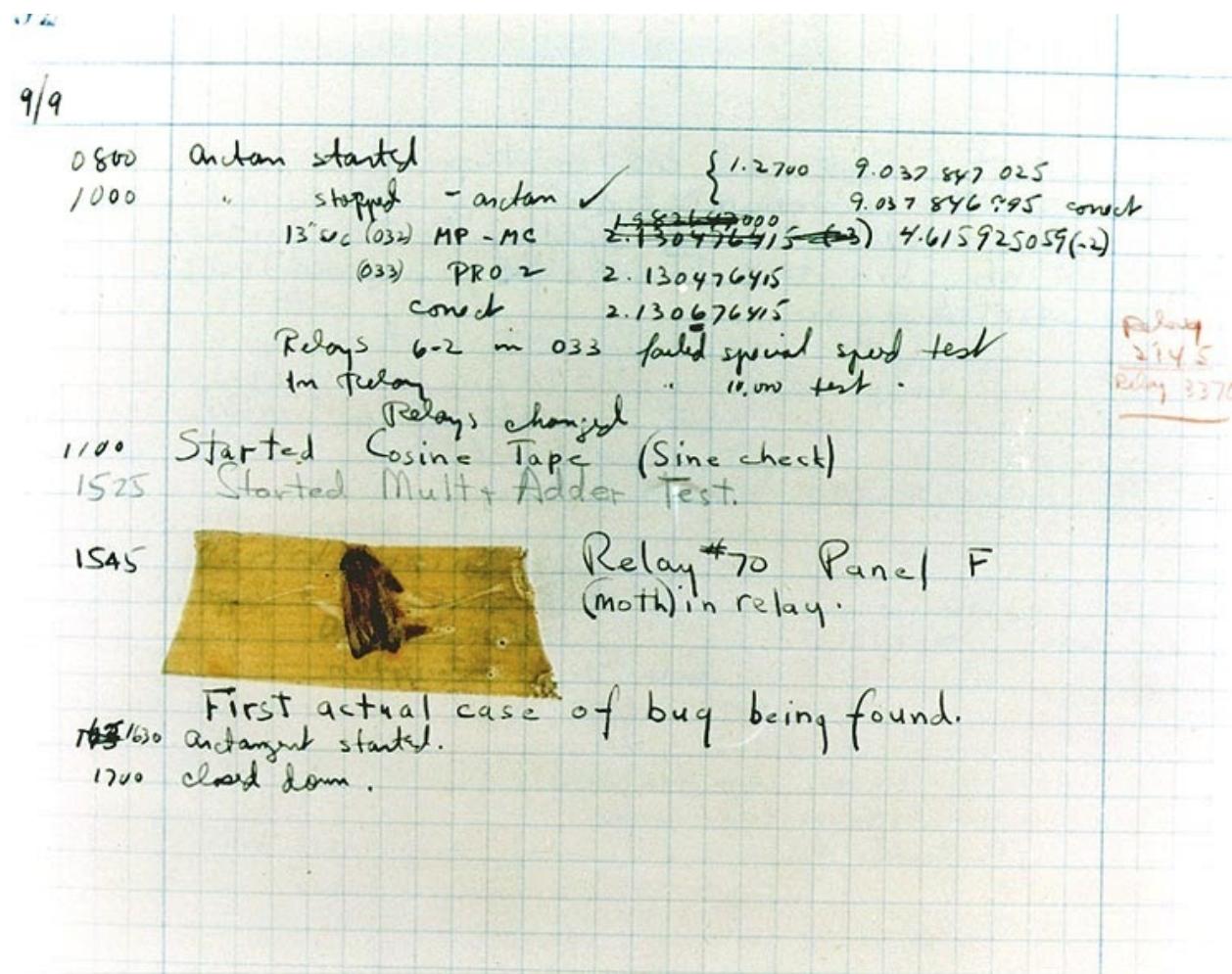


Figure 1.1 – The famous moth (By Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. - U.S. Naval Historical Center Online Library Photograph NH 96566-KN. Public Domain, <https://commons.wikimedia.org/w/index.php?curid=165211>)

Having understood what a bug and debugging basically is, let's move onto something interesting and important – we'll briefly examine a few real-world cases where a software bug (or bugs) has been the cause of some unfortunate and

tragic accidents.

Software bugs – a few actual cases

Using software to control electro-mechanical systems is not only common, it's pretty much all pervasive in today's world. The unfortunate reality though, is that software engineering is a relatively young field and that we humans are naturally prone to make mistakes; these factors can combine to create unfortunate accidents when software doesn't execute conforming to its design (which, of course, is called **buggy**).

Several *real-world* examples of this occurring exist; we highlight a few of them in the following sub sections. The brief synopsis given here is really just that – (too) brief; *to truly understand the complex issues behind failures like this, you do need to take the trouble to study the technical crash (or failure) investigation reports in detail* (do see the links in the *Further reading* section of this chapter). Here, I briefly mention and summarize these cases to: one, underline the fact that software failure, even in large, heavily tested systems, can and does occur, and two, to motivate all of us involved in any part of the software life cycle to pay closer attention, to stop making assumptions, and to do a better job in designing, implementing and testing the software we work on.

Patriot missile failure

During the Gulf War, the US had deployed a Patriot missile battery in Dhahran, Saudi Arabia. Its job was to track, intercept, and destroy incoming Iraqi Scud missiles. But, on 25 February 1991, a Patriot system failed to do so, causing the death of 28 soldiers and injury to about a hundred others. An investigation revealed that the problem's root was at the heart of the software tracking system. Briefly, the system uptime was tracked as a monotonically increasing integer value. It was converted to a *real* – floating point – value by multiplying the integer by 1/10 (which is a recurring binary expression evaluating to 0.0001100110011001100110011001100...). The trouble is, the computer's used a 24-bit register for this conversion, resulting in the computation being truncated at 24 bits; this caused a loss of precision which only became significant when the time quantity was sufficiently large.

This was exactly the case that day; the Patriot system had been up for about 100 hours; thus, the loss of precision during the conversion translated to an error of

approximately 0.34 seconds. Doesn't sound like much, except that a Scud missile's velocity is about 1,676 meters per second, thus resulting in a tracking error of about 570 meters. This was large enough for the Scud to be outside the Patriot tracking systems *range gate*, and was thus not detected.

Again, a case of loss of precision during conversion from an integer value to a real (floating point) number value.

ESA's unmanned Ariane 5 rocket

On the morning of 4th June 1996, the **European Space Agency's (ESA's)** Ariane 5 unmanned rocket launcher took off from the Guiana Space Centre, off the South American coast of French Guiana. A mere forty seconds into its flight, the rocket lost control and exploded. The final investigation report revealed that the primary cause ultimately came down to a software overflow error.

It's more complex than that; a brief summary of the chain of events leading to the loss of the rocket follows. (One realizes that, in most cases like this, it's not one single event that causes an accident; rather, it's a chain of several events). The overflow error occurred during the execution of code converting a 64-bit floating point value to a 16-bit signed integer; an unprotected conversion gave rise to an exception (**Operand Error**; the programming language was **Ada**); this in turn, occurred due to a much higher than expected internal variable value (BH – Horizontal Bias). The exception caused the shutdown of the **Inertial Reference System (SRI)** systems; this caused the primary **onboard computer (OBC)** to send erroneous commands to the nozzle deflectors resulting in full nozzle deflection of the boosters and the main Vulcain engine, which caused the rocket to veer dramatically off its flight path.

The irony is that the SRI's were, by default, not even supposed to function after launch; but due to a delay in the launch window the design specified that they remain active for 50 seconds after launch! An interesting analysis of why this software exception wasn't caught during development and testing (<https://archive.eiffel.com/doc/manuals/technology/contract/ariane/>) boils down to concluding that the fault lies in a **reuse error**:

"The SRI horizontal bias module was reused from a 10-year-old software, the software from Ariane 4."

MARS PATHFINDER TEST ISSUE

On July 4, 1997, NASA's Pathfinder lander touched down on the surface of Mars and proceeded to deploy its smaller robot cousin – the Sojourner rover, the very first wheeled device to embark upon another planet! The lander suffered from periodic reboots; the problem was ultimately diagnosed as being a classic case of **Priority Inversion** – a situation where a high priority task is made to wait for lower priority tasks. As such, this by itself may not cause an issue; the trouble is that the high priority task was left off CPU long enough for the watchdog timer to expire, causing the system to reboot.

An irony here was that there exists a well-known solution – enabling the **priority inheritance** feature of the semaphore object (allowing the task *taking* the semaphore lock to have its priority raised to the highest on the system – for the duration of its holding the lock – thus enabling it to complete its critical section and release the lock quickly, preventing starvation of higher priority tasks). The VxWorks RTOS defaulted to having it off and the **Jet Propulsion Laboratory (JPL)** team left it that way. Because they allowed the robot to continuously stream telemetry *debug* data to Earth, they were able to correctly determine this root cause and thus fix it – enabling priority inheritance. An important lesson here; as the team lead Glenn Reeves put it:

“we test what we fly and we fly what we test”

I'd venture that these articles (see the *Further reading* section) are a must-read for any systems software developer!

THE BOEING 737 MAX AIRCRAFT – THE MCAS AND LACK OF TRAINING TO THE FLIGHT CREW

Two unfortunate accidents, taking in all 346 lives, put the Boeing 737 MAX under the spotlight; the crash of the Lion Air Flight 610 from Jakarta into the Java Sea (29 Oct 2018) and the crash of Ethiopian Airlines Flight 302 from Nairobi into the desert (10 Mar 2019). These incidents occurred just 13 and 6 minutes after take-off, respectively.

Of course, the situation is complex; at one level, this is what has likely caused these accidents: once Boeing determined that the aerodynamic characteristics of the 737 MAX left something to be desired, they worked on fixing it via a

hardware approach. When that did not suffice, engineers came up with (what seemed) an elegant and relatively simple software fix, christened the **maneuvering characteristics augmentation system (MCAS)**. Two sensors on the nose continually measures the aircraft's **angle of attack (AoA)**; when the AoA is determined to be too high, this typically entails a pending stall (dangerous!); the MCAS kicks in, moving control surfaces on the tail elevator, causing the nose to go down and stabilizing the aircraft. But: for whatever reasons, the MCAS was designed to use only one of the sensors; if it did fail, the MCAS could automatically activate, causing the nose to go down and the aircraft to lose altitude; this is what seems to have actually occurred in both crashes.

Further, many pilot crews weren't explicitly trained on managing the MCAS (some claimed they weren't even aware of it!). The luckless flights pilot's apparently did not manage to override the MCAS, even when no actual stall occurred.

Other cases

A few other examples of such cases are as follows:

- June 2002, Fort Drum: a US Army report maintained that a software issue contributed to the death of two soldiers. This incident occurred when they were training to fire artillery shells; apparently, unless the target altitude is explicitly entered into the system, the software assumes a default of zero. Fort Drum is apparently 679 feet ASL
- In November 2001, a British engineer, John Locker, noticed that he could easily intercept American military satellite feeds, live imagery of US spy planes over the Balkans. The almost unbelievable reason – the stream was being transmitted unencrypted, enabling pretty much anyone in Europe with a regular satellite TV receiver to see it! In today's context, many IoT devices have similar issues...
- Jack Ganssle, a veteran and widely known embedded systems developer and author, brings out the excellent TEM – The Embedded Muse – newsletter bi-monthly; every issue has a section entitled *Failure of the Week*, typically highlighting a hardware and/or software failure; do check it out!
- Read the web page on *Software Horror Stories* here (<http://www.cs.tau.ac.il/~nachumd/horror.html>); though old, it provides

many examples of software gone wrong with, at times, tragic consequences.

Again, if interested in digging deeper, I urge you to read the detailed official reports on these accidents and faults; the *Further reading* section has several relevant links.

By now, you should be itching to begin debugging on Linux! Let's do just that – begin – by first setting up the workspace.

Setting up the workspace

Firstly, you'll have to decide whether to run your test Linux system as a native system (*on the bare metal*) or as a guest OS; we cover the factors that will help you decide. Next, we (briefly) cover the installation of some software (the guest additions) for the case where you use a guest Linux OS, followed by the required software packages to install.

Running Linux as a native or guest OS?

Ideally, you should run a recent Linux distribution (Ubuntu, Fedora, and so on) on native hardware. We tend to use Ubuntu 20.04 LTS in this book as the primary system to experiment upon. The more powerful your system – in terms of RAM, processing power and disk space – the better! Of course, as we shall be debugging at the level of the kernel, crashes and even data loss (chances of the latter are small, but nevertheless...) can occur; hence, the system should be a *test* one with no valuable data on it.

If running Linux on native hardware – on the bare metal, as it were - isn't feasible for you, then a practical and convenient alternative is to install and use the Linux distribution as a guest OS on a **Virtual Machine (VM)**. It's important to install a recent Linux distribution.

Running a Linux guest as a VM is certainly feasible but (there's always a *but* isn't it!), it will almost certainly feel a lot slower than running Linux natively. Still, if you must run a Linux guest, it certainly works; it goes without saying – the more powerful your host system, the better the experience. There's also an arguable advantage to running your test system as a guest OS: even if it does crash (please do expect that to happen, especially with the deliberate (de)bugging we'll do with this book!), you don't even need to reboot the

hardware; merely reset the hypervisor software running the guest (typically Oracle VirtualBox).

Alternate hardware – using Raspberry Pi (and other) ARM-based systems

Though we specified that you can run a recent Linux distro either as a native system or as a guest VM, the assumption was that it's an x86_64 system. While that suffices, to get more out of the experience of this book (and simply to have more fun), I highly recommend you also try out the sample code and run the (buggy) test cases on alternate architectures. With many, if not most, modern embedded Linux systems being ARM based (on both 32-bit ARM and 64-bit Aarch64 processors), the Raspberry Pi hardware is extremely popular, relatively cheap and has tremendous community support, making it an ideal test bed. I do use it every now and then within this book, in the chapters that follow; I'd recommend you do the same!

All the details – installation, setup, and so on – are amply covered in the well documented Raspberry Pi documentation pages here:
<https://www.raspberrypi.org/documentation/>.

Ditto for another popular embedded prototyping board - TI's BeagleBone Black (affectionately, the BBB). This site is a good place to get started with the BBB: <https://beagleboard.org/black>.

Running Linux as a guest OS

If you do decide to run Linux as a guest, I'd recommend using Oracle VirtualBox 6.x (or the latest stable version) as a comprehensive and powerful all-in-one GUI hypervisor application appropriate for a desktop PC or laptop. Other virtualization software, such as VMware Workstation or QEMU, should also be fine. All of these are freely available and open source. It's just that the code for this book has been tested on Oracle VirtualBox 6.1. Oracle VirtualBox is considered **Open Source Software (OSS)** and is licensed under the GPL v2 (the same as the Linux kernel). You can download it from
<https://www.virtualbox.org/wiki/Downloads>. Its documentation can be found here: https://www.virtualbox.org/wiki/End-user_documentation.

The host system should be either MS Windows 10 or later (of course even

~~The host system should be either MS Windows 10 or later (of course, even Windows 7 will work), a recent Linux distribution (for example, Ubuntu or Fedora), or macOS.~~

The guest (or native) Linux distribution can be any sufficiently recent one. For the purpose of following along the material and examples presented in this book, I'd recommend installing **Ubuntu 20.04 LTS**. This is what I primarily use for the book.

How can you quickly check which Linux distribution is currently installed and running?

On Debian/Ubuntu, the `lsb_release -a` command should do the trick; for example, on my guest Linux:

```
$ lsb_release -a 2> /dev/null
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal
$
```

How can one check if the Linux currently running is on native hardware or is a guest VM (or a container)? There are many ways to do so; the script `virt-what` is one (we will be installing it); other commands include `hostnamectl(1)`, `dmidecode(8)` (on x86), `systemd-detect-virt(1)` (if `systemd` is the initialization framework), `lshw(1)` (x86, IA-64, PPC), `raw` ways via `dmesg(1)` (grepping for Hypervisor detected), and via `/proc/cpuinfo`.

In this book, I shall prefer to focus on setting up what is key from a kernel (and user) debug perspective; hence, we won't discuss the in-depth details on installing a guest VM (typically on a Windows host running Oracle VirtualBox) here. If you require some help on this, please refer to the many links to tutorials on precisely this within the *Further reading* section for this chapter. (FYI, these details, and a lot more, are amply covered in my previous book *Linux Kernel Programming, Chapter 1, Kernel Workspace Setup*).

Tip – using prebuilt VirtualBox images

The **OSBoxes** project allows you to freely download and use prebuilt VirtualBox (as well as VMware) images for popular Linux distributions.

See their site here: <https://www.osboxes.org/virtualbox-images/>.

In our case, you can download a prebuilt x86_64 Ubuntu 20.04.3 (as well as other) Linux image here: <https://www.osboxes.org/ubuntu/>. It comes with the guest additions preinstalled. The default username/password is osboxes/osboxes.org .

(Of course, for more advanced readers, you'll realize it's really up to you. Running an as-light-as-possible Linux on a Qemu (emulated) standard PC is a choice here).

Note that if your Linux system is installed natively on the hardware platform or you're using an OSBoxes Linux with the VirtualBox guest additions preinstalled, or you're using a Qemu-emulated PC, simply skip the next section.

Installing the Oracle VirtualBox guest additions

The *guest additions* are essentially software (para-virtualization accelerators) that quite dramatically enhances the performance, as well as the *look and feel*, of the experience of running a guest OS on the host system; hence, it's important to have it installed. (Besides acceleration, the guest additions provide conveniences like the ability to nicely scale the GUI window, and share facilities, such as, folders, the clipboard, and drag and drop between the host and the guest).

Before doing this though, please ensure you have already installed the guest VM (as mentioned previously. Also, the first time you login, the system will likely prompt you to update and possibly restart; please do so). Then, follow along:

1. Log in to your Linux guest VM (I'm using the login name letsdebug ; guess why!) and first run the following commands within a Terminal window (on a shell):

```
sudo apt update  
sudo apt upgrade  
sudo apt install build-essential dkms linux-headers-$(uname -r)
```

(Ensure you run each command above on one line).

2. Install the Oracle VirtualBox Guest Additions now. Refer to *How to Install VirtualBox Guest Additions in Ubuntu*: <https://www.tecmint.com/install->

[virtualbox-guest-additions-in-ubuntu/](#)

3. On Oracle VirtualBox, to ensure that you have access to any shared folders you might have setup, you need to set the guest account to belong to the `vboxsf` group; you can do so like this (you'll require to log in again, or sometimes even reboot, to have this take effect):

```
sudo usermod -G vboxsf -a ${USER}
```

The commands (step 1), after updating, has us install the `build-essential` package along with a couple of others; this ensures that the compiler (`gcc`), `make`, and other essential build utility programs are installed so that the Oracle VirtualBox Guest Additions can be properly installed straight after (in step 2).

Installing required software packages

To install the required software packages, perform the following steps (do note that, here, we assume the Linux distribution is our preferred one, Ubuntu 20.04 LTS):

1. Within your Linux system (be it a native one or a guest OS), first do the following:

```
sudo apt update
```

Now, to install the remaining required packages for the kernel build, run the following command in a single line:

```
sudo apt install bison flex libncurses5-dev ncurses-dev xz-util
```

(The `-y` option switch has `apt-get(8)` assume a `yes` answer to all prompts; careful though, this could be dangerous in other circumstances).

2. To install packages required for work we'll do in other parts of this book, run the following command in a single line:

```
sudo apt install bc bpfcc-tools bsdmainutils clang cmake cppcheck dwarves exuberant-ctags fakeroot flawfinder git gnome-system-monitor hwloc indent kernelshark libnuma-dev libjson-c-dev linux-tools-net-tools numactl openjdk-16-jre openssh-server perf-tools-unstable python3-distutils rt-tests smem sparse stress sysfsutils tldr-project tree tuna virt-what -y
```

Tip - a script to auto-install required packages

To make the (immediately above-mentioned) package install task simpler, you can make use of a simple bash script that's part of the GitHub repo for this book: `pkg_install4ubuntu_1kp.sh`. It's been tested on an x86_64 osboxes Ubuntu 20.04.3 LTS VM (running on Oracle VirtualBox 6.1).

Great; now that the required packages are all installed, let's proceed with understanding the next portion of our workspace setup – the need for two kernels!

A tale of two kernels

When working on a project or product, there obviously will be a Linux kernel that will be deployed as part of the overall system.

Information box

A quick aside: a working Linux system minimally requires a bootloader, a kernel, and root filesystem images.

This system that's deployed to the outside world is in general termed the **production system** and the kernel as the **production kernel**, as, of course, it's the one that runs while it's being used in the field (or on-premise, or at the customer location). Here, we'll limit our discussion to the kernel only. The configuration, build, test, debug, and deployment of the production kernel is, no doubt, a key part of the overall project.

Do note though, in many systems (especially the enterprise-class ones), the production kernel is often simply the default kernel that's supplied by the vendor (RedHat, SuSe, Canonical, or others). On most embedded Linux projects and products, this is likely not the case: the platform (or **Board Support Package (BSP)**) team or a vendor will select a base mainline kernel (typically from `kernel.org`) and work on it; this can include enhancements, careful configuration, and deployment of the custom-built production kernel.

For the purpose of our discussion, let's assume that we require to configure and build a custom kernel.

A production and a debug kernel

However (and especially after having read the earlier section *Software bugs – a few actual cases*), you will realize that there's always the off-chance that even the kernel – more likely the code you and your team added to it (the kernel modules, drivers, interfacing components) – has hidden faults, bugs. With a view to catching them before the system hits the field, thorough testing / QA is of prime importance!

Now, the issue is this: unless certain deeper checks are enabled within the kernel itself, it's entirely possible that they can escape your test cases. So, why not simply enable them? Well, one, these *deeper checks* are typically switched off by default in the production kernel's configuration. Two, when turned on, they do result in performance degradation, at times quite significantly.

So, where does that leave us? Simple, really: you should plan on working with at least two, and possibly three, kernels:

- One, a carefully tuned production kernel, geared towards efficiency, security, and performance
- Two, a carefully configured **debug kernel**, geared towards catching pretty much all kinds of bugs! Performance is not a concern here, catching bugs is
- Three (optional, case-by-case): a kernel with one or more very specific debug config options enabled and the rest turned off

The second one, the so-called **debug kernel** is configured in such a way that all required or recommended **debug options** are turned on, enabling you to (hopefully!) catch those hidden bugs. Of course, performance might suffer as a result, but that's okay; catching – and subsequently fixing – kernel-level bugs are well worth it. Indeed, in general, during development and (unit) testing, performance isn't paramount; catching and fixing deeply hidden bugs is!

The debug kernel is only used during development, test, and very possibly later, when bugs do actually surface. How exactly it's used later is something we shall certainly cover in the course of this book.

Also, this point is key: it usually is the case that the mainline (or vanilla) kernel that your custom kernel is based upon is working fine; the bugs are generally introduced via custom enhancements and **kernel modules**. As you will know,

we typically leverage the kernel's **Loadable Kernel Module (LKM)** framework to build custom kernel code – the most common being device drivers; it can also be anything else: custom network filters / firewall, a new filesystem or I/O scheduler. These are out-of-tree kernel components (typically some .ko files) that become part of the root filesystem (they're usually installed into `/lib/modules/$(uname -r)`). The debug kernel will certainly help catch bugs in your kernel modules as their test cases are executed, as they run.

The third kernel option – an in-between of the first two – is optional of course; from a practical real-world point of view, it may be exactly what's required on a given setup. With certain kernel debug systems turned on, to catch specific types of bugs that you're hunting (or anticipate) and the rest turned off, it can become a pragmatic way to debug even a production system, keeping performance high enough.

For practical reasons, in this book, we'll configure, build and make use of the first two kernels – a custom production one and a custom debug one, only; the third option is yours to configure as you gain experience on both the kernel debug features and tools as well as your particular product or project.

Which kernel release to use?

A key topic: the Linux kernel project is often touted as the most successful opensource project ever, with literally hundreds of releases and a release cadence that's truly phenomenal for such an enormous project (it averages a new release every 6 to 8 weeks!). Among them, which one should we use (as a starting point, at least)?

It's really important to use the *latest stable kernel version*, as it will include all the latest performance and security fixes. Not just that, the kernel community has different release *types*, which determines how long a given kernel release will be maintained (bug and security fixes applied, as they become known). For typical projects or products, selecting the latest stable **Long Term Stable (LTS)** kernel release, thus makes the best sense. Of course, as already mentioned, on many projects – typically the server / enterprise class ones – the vendor (RedHat, SuSe, and others) might well supply the production kernel to be used; here, for the purpose of our learning, we'll start from scratch, configure, and build a custom Linux kernel ourselves (as is often the case on embedded projects).

As of this writing, the latest LTS Linux kernel is **5.10** (particularly, version 5.10.60); I shall use this kernel throughout this book. (You will realize that by the time you’re reading this, it’s entirely possible, in fact very likely, that the latest LTS kernel has evolved to a newer version). Besides, a key point in our favor – the 5.10 LTS kernel will be supported by the community until December 2026, thus keeping it relevant and valid for a pretty long time!

So, great, let’s get to configuring and building both our custom production and debug kernels! We’ll begin with the production one.

Setting up our custom production kernel

(Here, I shall have to assume you are familiar with the general procedure involved in building a Linux kernel from source: obtaining the kernel source tree, configuring, and building it. In case you’d like to brush up on this, the *Linux Kernel Programming – Part 1* book covers this in a lot of detail. As well, do refer the tutorials and links in the *Further reading* section of this chapter).

Though this is meant to be our production kernel, we’ll begin with a rather simplistic default that’s based on the existing system (this approach is sometimes called the **tuned kernel config via the localmodconfig** one. FYI, this, and a lot more, is covered in depth in the *Linux Kernel Programming – Part 1* book). Once we’ve got a reasonable starting point, we’ll further tune the kernel for security. Let’s begin by performing some base configuration:

1. Create a new directory in which you work upon the upcoming production kernel:

```
mkdir -p ~/lkd_kernels/productionk
```

Bring in the kernel source tree of your choice. Here, as mentioned in *Which kernel release to use?*, we shall use the latest (at the time of writing this) **LTS Linux kernel, version 5.10.60**:

```
cd ~/lkd_kernels
wget https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linu
```

Notice that here we have simply used the `wget (1)` utility to bring in the compressed kernel source tree; there are several alternate ways (including using `git (1)`)

Note

As you'll know, the number in parentheses following the command name – for example, `wget (1)` – is the section within the manual or man pages where documentation on this command can be found).

2. Extract the kernel source tree:

```
tar xf linux-5.10.60.tar.xz --directory=productionk/
```

3. Switch to the directory it's just been extracted into (using `cd linux-5.10.60`) and briefly verify the kernel version information as shown in the following screenshot:

Figure 1.2 – Screenshot of the LTS kernel source tree

Every kernel version is christened with a (rather exotic) name; our 5.10.60 LTS kernel has an appropriately nice name (`Dare mighty things`), don't you think?

4. Configure for appropriate defaults. This is what you can do to obtain a decent, tuned starting point for kernel config based on the current config:

```
lsmod > /tmp/lsmod.now  
make LSMOD=/tmp/lsmod.now localmodconfig
```

Note

The preceding command might interactively ask you to specify some choices; just selecting the defaults (by pressing the *Enter* key) is fine for now. The end result is the kernel configuration file saved as `.config` in the root of the kernel source tree (the current directory).

We backup the config file as follows:

```
cp -af .config ~/lkd_kernels/kconfig_prod01
```

Tip

You can always do `make help` to see the various options (including

config) available to you.

Before jumping into the building our production kernel, it's really important to consider the security aspect. Let's first configure our kernel to be more secure, hardened.

Securing your production kernel

With security being a major concern, the modern Linux kernel has many security and kernel hardening features. The thing is, there always tends to be a trade-off between security and convenience/performance. Thus, many of these hardening features are *off* by default; several are designed as an opt-in system: if you want it, turn it on by selecting it from the kernel config menu (via the familiar `make menuconfig` UI). This makes sense to do, especially on a production kernel.

The question is: *how will I know which exactly config features regarding security to turn on or off?* There's literature on this and, better, some utility scripts which examine your existing kernel config and can make recommendations based on existing state-of-the-art security best practice! One such tool is Alexander Popov's `kconfig-hardened-check` Python script (<https://github.com/a13xp0p0v/kconfig-hardened-check>). Here's a screenshot of a portion of its output, when I ran it against my custom kernel configuration file:

```

$ git clone https://github.com/a13xp0p0v/kconfig-hardened-check
Cloning into 'kconfig-hardened-check'...
remote: Enumerating objects: 1339, done.
remote: Counting objects: 100% (123/123), done.
remote: Compressing objects: 100% (87/87), done.
remote: Total 1339 (delta 62), reused 90 (delta 35), pack-reused 1216
Receiving objects: 100% (1339/1339), 1.57 MiB | 880.00 KiB/s, done.
Resolving deltas: 100% (806/806), done.
$
$ ls kconfig-hardened-check/
LICENSE.txt MANIFEST.in README.md bin/ contrib/ default.nix kconfig_hardened_check/ setup.cfg setup.py*
$ ls kconfig-hardened-check/bin/
kconfig-hardened-check*
$
$ cd kconfig-hardened-check
$ bin/kconfig-hardened-check -p X86_64 -c ~/lkd_kernels/kconfig_prod01
[+] Config file to check: /home/letsdebug/lkd_kernels/kconfig_prod01
[+] Detected architecture: X86_64
[+] Detected kernel version: 5.10
=====
=====          option name          | desired val | decision |      reason      |   check result
=====

CONFIG_BUG           |     y     |defconfig | self_protection |    OK
CONFIG_SLUB_DEBUG   |     y     |defconfig | self_protection |    OK
CONFIG_GCC_PLUGINS   |     y     |defconfig | self_protection | FAIL: not found
CONFIG_STACKPROTECTOR_STRONG |     y     |defconfig | self_protection |    OK
CONFIG_STRICT_KERNEL_RWX |     y     |defconfig | self_protection |    OK
CONFIG_STRICT_MODULE_RWX |     y     |defconfig | self_protection |    OK
CONFIG_REFCOUNT_FULL |     y     |defconfig | self_protection | OK: version >= 5.5
CONFIG_IOMMU_SUPPORT |     y     |defconfig | self_protection |    OK
CONFIG_RANDOMIZE_BASE |     y     |defconfig | self_protection |    OK
CONFIG_THREAD_INFO_IN_TASK |     y     |defconfig | self_protection |    OK
CONFIG_VMAP_STACK    |     y     |defconfig | self_protection |    OK

```

Figure 1.3 – Partial screenshot – truncated output from the kconfig-hardened-check script

(We won't be attempting to go into details regarding the useful kconfig-hardened-check script here, as it's beyond this book's scope; do lookup the GitHub link provided to see details). Having followed most of the recommendations from this script, I generated a kernel config file:

```

$ ls -l .config
-rw-rw-r-- 1 letsdebug letsdebug 156646 Aug 19 13:02
.config
$
```

Note

My kernel config file for the production kernel can be found on the book's GitHub code repo here: https://github.com/PacktPublishing/Linux-Kernel-Debugging/blob/main/ch1/kconfig_prod01. (FYI, our custom debug kernel config fie, that we'll be generating in the following section, can be found within the same folder as well).

Now that we have appropriately configured our custom production kernel, let's build it; the following commands should do the trick (with `nproc` (1) helping us with the number of CPU cores onboard):

```
$ nproc  
4  
$ make -j8  
[ ... ]  
BUILD arch/x86/boot/bzImage  
Kernel: arch/x86/boot/bzImage is ready (#1)  
$
```

Information box

If you're working on a typical embedded project, you will require to install a toolchain and cross compile the kernel. Also, you'd normally set the environment variable `ARCH` to the machine type (for example, `ARCH=arm64`) and the environment variable `CROSS_COMPILE` to the cross compiler prefix (for example, `CROSS_COMPILE= aarch64-none-linux-gnu-`). Your typical embedded Linux builder systems – Yocto and Buildroot being very common – pretty much automatically take care of this.

As you can see, as a thumb-rule, we set the number of jobs to execute as twice the number of CPU cores available via `make`'s `-j` option switch. The build should complete in a few minutes; once done, let's check that the compressed and uncompressed kernel image files have been generated:

```
$ ls -lh arch/x86/boot/bzImage vmlinuz  
-rw-rw-r-- 1 letsdebug letsdebug 9.1M Aug 19 17:21  
arch/x86/boot/bzImage  
-rwxrwxr-x 1 letsdebug letsdebug 65M Aug 19 17:21 vmlinuz*  
$
```

Note that it's always only the first one, `bzImage` – the compressed kernel image – that we shall boot from. Then what's the second image, `vmlinuz`, for? *Very relevant here*: it's what we shall (later) often require when we need to perform kernel debug! It's the one that holds all the symbolic information, after all. Our production kernel config will typically cause several kernel modules (LKMs) to be generated within the kernel source tree. They have to be installed into a well-known location (`/lib/modules/$(uname -r)`); this is achieved by doing, as root:

```
$ sudo make modules_install
[sudo] password for letsdebug: xxxxxxxxxxxxxxxxx
  INSTALL arch/x86/crypto/aesni-intel.ko
  INSTALL arch/x86/crypto/crc32-pclmul.ko
[ ... ]
  DEPMOD 5.10.60-prod01
$ ls /lib/modules/
5.10.60-prod01/ 5.11.0-27-generic/ 5.8.0-43-generic/
$ ls /lib/modules/5.10.60-prod01/
build@ modules.alias.bin modules.builtin.bin modules.dep.bin modu:
$
```

Final step, we make use of an internal script to generate the `initramfs` image and setup the bootloader (in this case, on our `x86_64`, it's GRUB) by simply running:

```
sudo make install
```

For details and conceptual understanding of the `initramfs`, as well as some basic GRUB tuning, do see the [Linux Kernel Programming - Part 1](#) book. We also provide useful references within the *Further reading* section for this chapter.

Now all that's left to do is reboot your guest (or native) system, interrupt the bootloader (typically by holding the `Shift` key down during early boot) and selecting the newly built production kernel:

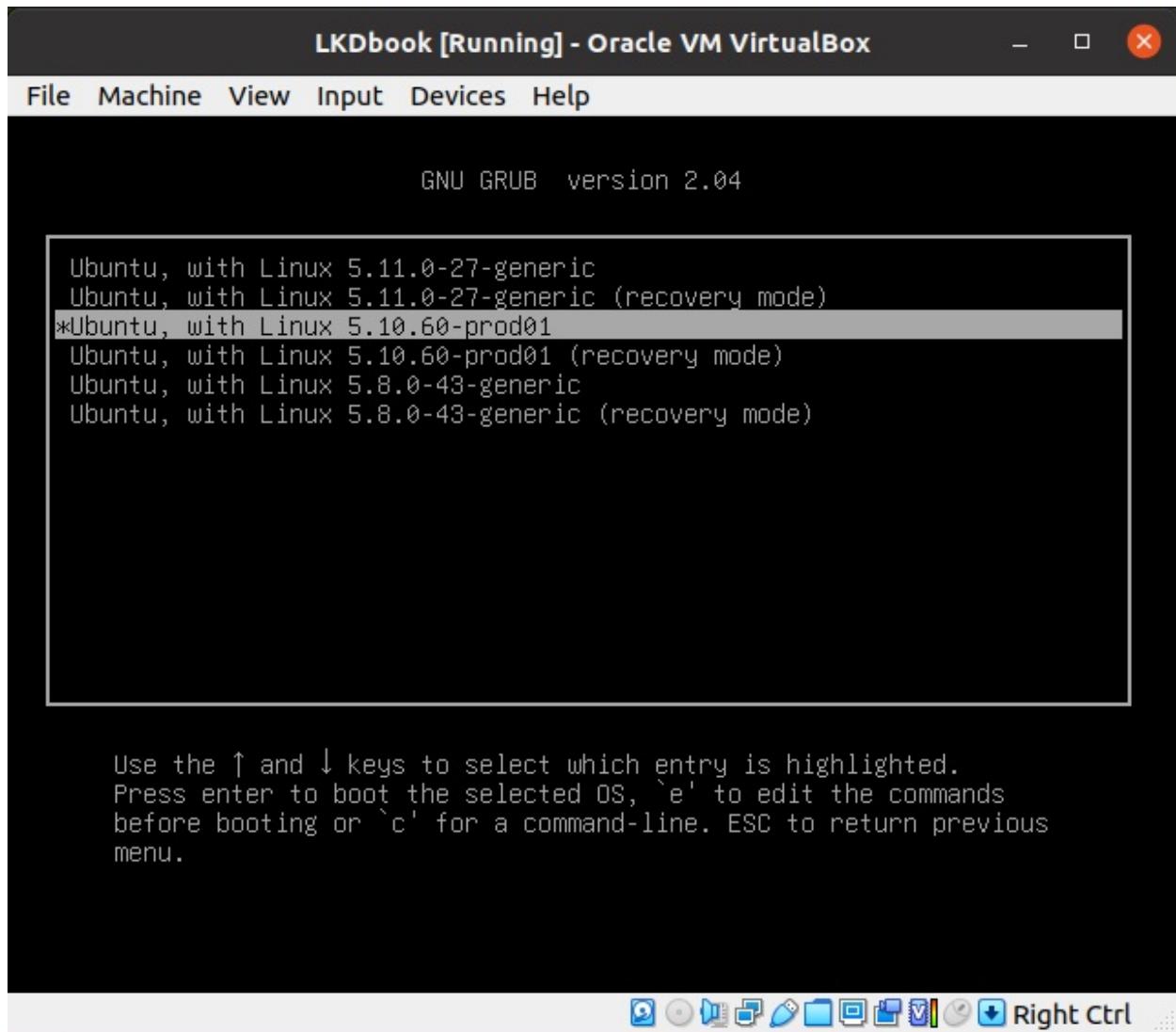


Figure 1.4 – Screenshot showing the GRUB bootloader screen and the new production kernel to boot from

As you can see from the preceding screenshot, I'm running the system as a guest OS via Oracle VirtualBox; holding the *Shift* key down during early boot got this bootloader screen to show up; I scrolled down, selected the new production kernel and pressed *[Enter]* to boot into it.

Voila, we're now running our (guest) system with our brand new production kernel:

```
$ uname -a
Linux dbg-LKD 5.10.60-prod01 #1 SMP PREEMPT Thu Aug 19 17:10:00 IST
$
```

Information box

The new kernel should run just fine with the existing root filesystem – the libraries and applications are loosely coupled with the OS, allowing different versions of the kernel (one at a time, of course) to simply mount and use them. Also, you may not get all the bells and whistles; for example, on my guest OS with our new production kernel, the screen resizing, shared folders, and so on. Features are missing. How come? They depend on the guest additions whose kernel modules haven't been built for this kernel. In this case, I find it a lot easier to work on the guest using the console over SSH. To do so, I installed the **dropbear** lightweight SSH server on the guest and then logged in over SSH from my host system. Windows users might like to try an SSH client like **putty**. (In addition, you might require setting up another bridged mode network adapter on the Linux guest).

You can (re)check the current kernel's configuration by looking up `/boot/config-$(uname -r)`. In this case, it should be that of our production kernel, tuned towards security and performance.

Tip

To have the GRUB bootloader prompt always show up at boot: make a copy of `/etc/default/grub` (to be safe), then edit it, adding the line `GRUB_HIDDEN_TIMEOUT_QUIET=false` and (possibly) commenting out the line `GRUB_TIMEOUT_STYLE=hidden`

Change the `GRUB_TIMEOUT` value from `0` to `3` (seconds). Run `sudo update-grub` to have the changes take effect, and reboot to test.

So, good, you now have your guest (or native) Linux OS running a new production kernel. During the course of this book, you shall encounter various kernel-level (and some user-mode) bugs while running this kernel. Identifying the bug(s) will often involve your booting via the debug kernel instead. So, let's now move onto creating a custom debug kernel for the system. Read on!

Setting up our custom debug kernel

As you have already setup a production kernel (as described in detail in the previous section), I won't repeat every step in detail here, just the ones that differ:

1. Firstly, ensure you have booted into the production kernel that you built in the previous section; this is to ensure that our debug kernel config uses it as a starting point:

```
$ uname -r  
5.10.60-prod01
```

2. Create a new working directory and extract the same kernel version again. It's important to build the debug kernel in a separate workspace from that of the production one; true, it takes a lot more disk space but it keeps them clean and from stepping on each other's toes as you perhaps modify their configs:

```
mkdir -p ~/lkd_kernels/debug
```

3. We already have the kernel source tree (we earlier used `wget` to bring in the 5.10.60 compressed source); let's reuse it, this time extracting it into the debug kernel work folder:

```
cd ~/lkd_kernels  
tar xf linux-5.10.60.tar.xz --directory=debugk/
```

4. Switch to the debug kernel directory and setup a starting point for kernel config – via the `localmodconfig` approach – just as we did for the production kernel. This time though, the config will be based on that of our custom production kernel, as that's what is running right now on the system:

```
cd ~/lkd_kernels/debugk/linux-5.10.60  
lsmod > /tmp/lsmod.now  
make LSMOD=/tmp/lsmod.now localmodconfig
```

5. As this is a debug kernel, we now configure it with the express purpose of turning on the kernel's debug infrastructure as much as is useful. (Though we do not care that much for performance and/or security, the fact is that as we're inheriting the config from the production kernel, the security features are enabled by default).

The interface we use to configure our debug kernel is the usual one:

```
make menuconfig
```

Much (if not most) of the kernel debug infrastructure can be found in the last main menu item here – the one named Kernel hacking :

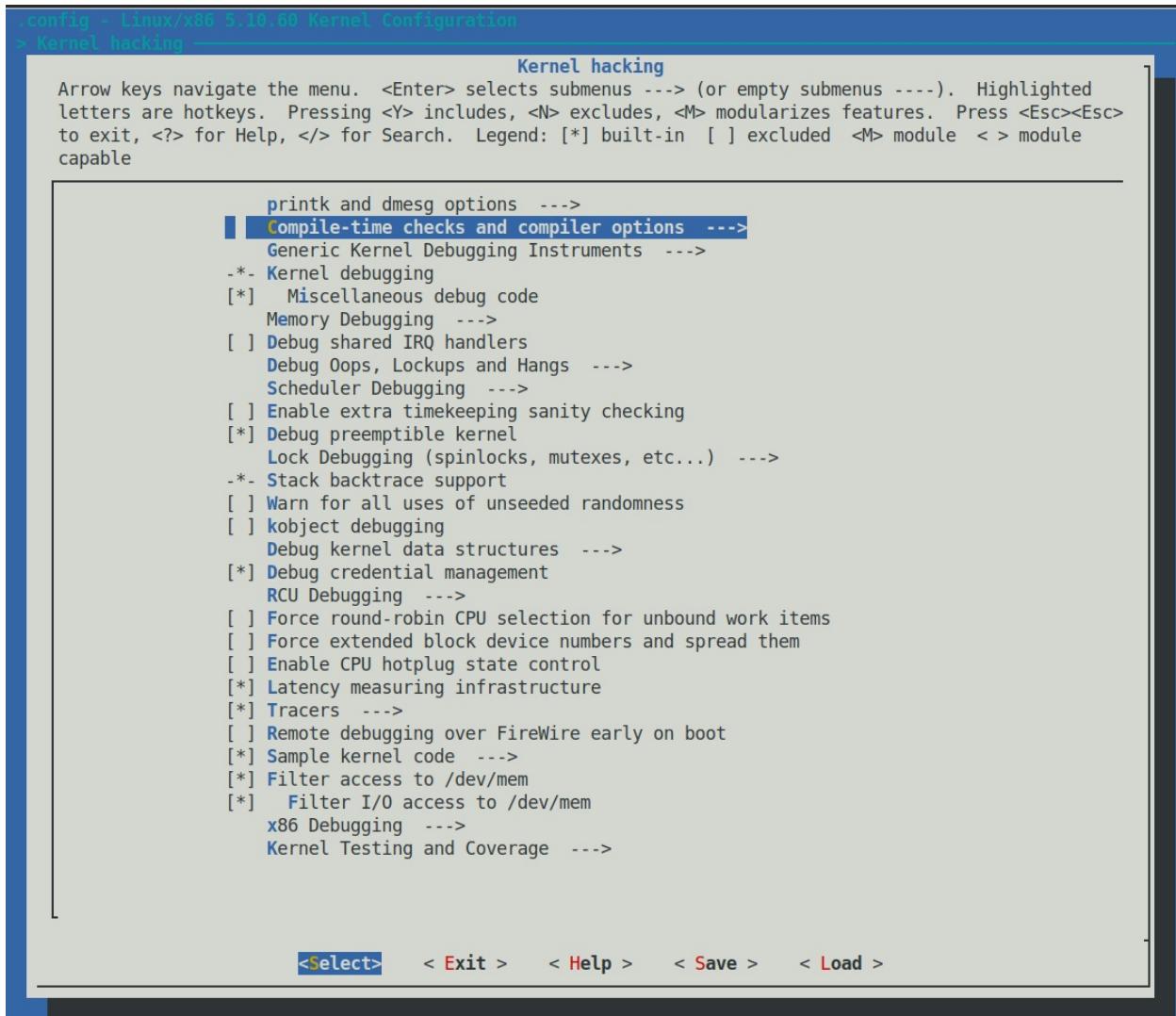


Figure 1.5 – Screenshot: make menuconfig / Kernel hacking – the majority of kernel debug options live here

There are just too many kernel configs relating to debugging to discuss individually here and now; several of them will be an important kernel debug feature that we shall explain and make use of in the chapters that follow. The following table summarizes some of the kernel config variables that we set or clear, depending on whether the config is for the debug or the production kernel. It is by no means exhaustive.

Not all of the config changes we make are within the Kernel hacking menu; others are changed as well (see the merged rows in the table that specify from which menu they originate as well as the Kconfig file(s) that they originate from). Further, the <D> in the Typical value ... columns indicates that the *decision* is left to you (or the platform/BSP team) as the particular value to use does depend on the actual product or project, it's **High Availability (HA)** characteristics, security posture, and so on.

Tip

You can search within the `make menuconfig` UI for a given config variable (`CONFIG_XXX`) by typing the key / (just as in vi!) and then typing the string to search for.

Kernel Config item	Meaning in brief	Type
General setup : init/Kconfig		
<code>CONFIG_LOCALVERSION</code>	Append a string to kernel version; useful (for example, -kdbg01)	<[
<code>CONFIG_IKCONFIG</code>	Allow complete kernel config to be stored in-kernel; can extract via scripts/extract-ikconfig (or, see the next one); very useful	Or me
<code>CONFIG_IKCONFIG_PROC</code>	Access the in-kernel config file via /proc/config.gz (for example, extract with gunzip -c /proc/config.gz)	Or
<code>CONFIG_KALLSYMS_ALL</code>	Loads <i>all</i> symbols into the kernel image	<[
Processor type and features :		
<code>arch/<arch>/Kconfig</code>		

Processor type and features :
`arch/<arch>/Kconfig`

CONFIG_CRASH_DUMP	Enable a crash-dump capable kernel, triggered via the kexec() on kernel bug/Oops	<[
CONFIG_RANDOMIZE_BASE	The Kernel Address Space Layout Randomization (KASLR) feature support; randomizes the physical address at which the kernel image is decompressed [...] as a security feature that deters exploit attempts ...	Or
General architecture-dependent options :		
arch/Kconfig		
CONFIG_KPROBES	General architecture-dependent options / Kprobes: allow to hook into almost any kernel function/address; useful for non-intrusive instrumentation and debug	<[
CONFIG_STACKPROTECTOR_STRONG	Intelligently add stack-protection canary logic via compiler; useful to detect Buffer overFlow (BoF) attacks	<[
CONFIG_ARCH_MMAP_RND_BITS	Number of bits to use to determine the random offset to the base address of process memory regions; higher is good for security	32
CONFIG_VMAP_STACK	Enable vmapped (vmalloc() Or allocated) kernel stacks with guard pages	
Executable file formats / CONFIG_COREDUMP	Enable core dumping	<I
Enable loadable module support :		
init/Kconfig		
CONFIG_MODULE_SIG_FORCE	Only loads modules with a valid signature; used with a kernel lockdown LSM	<I

CONFIG_MODULE_SIG_ALL	Auto sign all kernel modules during the make modules_install step	On
CONFIG_UNUSED_SYMBOLS	Enable unused but exported kernel symbols; a <i>bridge</i> that should soon get removed	Off
Device Drivers / Network device support:		
drivers/net/Kconfig		
CONFIG_NETCONSOLE	Network console (netconsole) logging support; log kernel printk's over the network	On
CONFIG_NETCONSOLE_DYNAMIC	Ability to dynamically reconfigure logging targets	<I
Kernel Hacking :		
lib/Kconfig.debug, lib/Kconfig.*		
printk and dmesg options :		
lib/Kconfig.debug		
CONFIG_DYNAMIC_DEBUG	Dynamic debug feature for debug printk + logging; (this auto-selects the core CONFIG_DYNAMIC_DEBUG_CORE feature as well)	On
Compile-time checks and compiler options		
: lib/Kconfig.debug		
CONFIG_DEBUG_INFO	compile the kernel and modules with debug info (Off
	gcc -g)	
CONFIG_DEBUG_BUGVERBOSE	BUG() prints filename and line number + instruction pointer register and Oops trace	<I
CONFIG_DEBUG_INFO_BTF	Generates dedup-ed BPF Type Format (BTF) info; useful for running eBPF in future (requires pahole v1.16 or later installed [1])	<I

Generic Kernel Debugging Instruments :

<code>lib/Kconfig[.debug .kgdb .ubsan]</code>	Enable all magic SysRq functionality by setting this bitmask (0x0=off , 0x1=all on, default is 0x01b6)	<[
<code>CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE</code>]
<code>CONFIG_DEBUG_FS_DISALLOW_MOUNT</code>	Debugfs additional protection layer on production; with this, API works but the filesystem isn't visible (not mounted)	O
<code>CONFIG_KGDB</code>	Remote debug kernel via GDB	O
<code>CONFIG_UBSAN</code>	Enables the Undefined Behavior sanity checker	<[
Memory Debugging :		
<code>lib/Kconfig[.debug .kasan .kgdb]</code>		
<code>CONFIG_DEBUG_PAGEALLOC</code>		
<code>CONFIG_DEBUG_WX</code>	Page memory allocation on 192.168.1.20 where cs are tracked; can help in detecting some types of memory corruption	O
<code>CONFIG_DEBUG_KMEMLEAK</code>	Warn on any w+x memory mappings seen at boot (writeable memory should, in general, not be executable, that is, w^x should apply)	O
<code>CONFIG_SCHED_STACK_END_CHECK</code>		
<code>CONFIG_KASAN</code>	Kernel memory leak detector	O
<code>CONFIG_SCHED_STACK_END_CHECK</code>	Stack overrun check when schedule() is called; minimal runtime overhead	O
<code>CONFIG_KASAN</code>	Enable Kernel Address SANitizer – adds compile-time instrumentation; extremely useful in catching many memory bugs (OOB, UAF, and so on)	O
Debug Oops, Lockups and Hangs :		
<code>lib/Kconfig.debug</code>		
<code>CONFIG_PANIC_ON_OOPS</code>	Enablers panic on any Oops (kernel bug)	O

CONFIG_PANIC_TIMEOUT	timeout (seconds) after which system reboots; requires arch-level reboot support. Value n == 0 => wait forever, n >= 1 => reboot after ‘n’ seconds
Lock debugging (spinlocks, mutexes, etc...)	
: lib/Kconfig.debug	
CONFIG_PROVE_LOCKING	Prove locking correctness via Of the very sophisticated lockdep lock validator; also detects possibility of deadlock
CONFIG_LOCK_STAT	Track lock contention code Of regions
CONFIG_DEBUG_ATOMIC_SLEEP	Noisy warnings on any sleep Of performed within an atomic section of code
<Various> : lib/Kconfig.debug	
CONFIG_BUG_ON_DATA_CORRUPTION	Debug kernel data structures: Of If data corruption found in a kernel structure, call BUG()
CONFIG_DEBUG_CREDENTIALS	Debug credential management: Debug checks to struct cred; useful for security as well
CONFIG_LATENCYTOP	Latency measuring infrastructure: Enable to use latencyTOP tool
CONFIG_STRICT_DEVMEM	Filter access to /dev/mem : If Off, userspace apps can memory map any memory region – user and kernel
Tracers : kernel/trace/Kconfig	
CONFIG_FUNCTION_TRACER	Enable Ftrace – tracing of every kernel function; disabled at runtime by default
CONFIG_FUNCTION_GRAPH_TRACER	Enables tracing of the call graph as well; useful to function tracer

<code>CONFIG_DYNAMIC_FTRACE</code>	<code>profile/debug</code>	<code><[</code>
<code><Early printk> : arch-dependent ;</code> <code>CONFIG_EARLY_PRINTK on x86</code>	Dynamic function tracing; no performance overhead when function tracing is disabled (the default)	
	x86 Debugging: Useful to see early kernel printk's before console device is initialized	<code>Of</code>
Kernel Testing and Coverage : <code>lib/Kconfig*, lib/kunit/Kconfig</code>		
<code>CONFIG_FAULT_INJECTION</code>	Enable the kernel's fault-injection framework	<code>Of</code>
Security : <code>security/Kconfig, security/*/Kconfig*</code>		
<code>CONFIG_SECURITY_DMESG_RESTRICT</code>	If On, only root can read kernel printk's via dmesg(1)	<code>Or</code>
<code>LOCK_DOWN_KERNEL_FORCE_CONFIDENTIALITY</code>	Kernel in lockdown (via an LSM), mode set to confidentiality; if On, modules can't be (un)loaded	<code>Or</code>

Table 1.1 – Summary of a few kernel config variables, meaning, and value

Besides the `<D>` value, the other values shown in the preceding table are merely my recommendation; they may or may not be suitable for your particular use case.

[1] Installing `pahole` v1.16 or later: `pahole` is part of the `dwarves` package. However, on Ubuntu 20.04 (or older) it's version 1.15 which causes the kernel build - when enabled with `CONFIG_DEBUG_INFO_BTF` – to fail. This is as `pahole` version 1.16 or later is required. To address this on Ubuntu 20.04, we've provided the v1.17 Debian package in the root of the GitHub source tree. Install it manually as follows:

```
sudo dpkg -i dwarves_1.17-1_amd64.deb
```

Being able to view (query) the currently running kernel's configuration can prove to be a very useful thing, especially on production systems. This can be done by looking up (grepping) `/proc/config.gz` (a simple `zcat /proc/config.gz | grep CONFIG_<FOO>` is typical). The pseudo-file

`/proc/config.gz` contains the entire kernel config (it's practically equivalent to the `.config` within the kernel source tree). Now, this pseudo-file is only generated by setting `CONFIG_IKCONFIG=y`. As a safety measure on production systems, we set this config to the value `m` on production, implying that it's available as a kernel module (called configs). Only once you load this up does the `/proc/config.gz` file become visible; and of course, to load it up you require root access...

Here's an example of loading it up and then querying the kernel config (for this very feature!):

```
$ ls -l /proc/config.gz  
ls: cannot access '/proc/config.gz': No such file or directory
```

Ok, to begin with (on production) it doesn't show up. So do this:

```
$ sudo modprobe configs  
$ ls -l /proc/config.gz  
-r--r--r-- 1 root root 34720 Oct  5 19:35 /proc/config.gz  
$ zcat /proc/config.gz |grep IKCONFIG  
CONFIG_IKCONFIG=m  
CONFIG_IKCONFIG_PROC=y
```

Ah, it now works just fine!

Food for thought

Did you notice? In *Table 1.1*, I've set the production kernel's value for `CONFIG_KALLSYMS_ALL` as `<D>`, implying it's up to the system architects to decide whether to keep it On or Off. Why? Shouldn't *all kernel symbols* be disabled (off) in a production system? Well, that's the common decision. Recall our brief on the Mars Pathfinder mission – where it initially failed due to a priority inversion issue. The tech lead of the software team at JPL, Glenn Reeves, made a very interesting statement in his now famous response to Mike Jones

(https://www.cs.unc.edu/~anderson/teach/comp790/papers/mars_pathfinder_lo)

The software that flies on Mars Pathfinder has several debug features within it that are used in the lab but are not used on the flight spacecraft (not used because some of them produce more information than we can send back to Earth). These features were not "fortuitously" left enabled but remain in the software by design. We strongly believe in the "test what you

fly and fly what you test" philosophy.

Sometimes, keeping debug features (and of course, logging) turned on in the production version of the system, can be immensely helpful!

For now, don't stress too much on exactly what each of these kernel debug options mean and how you're to use them; we shall cover most of these kernel debug options in the coming chapters. The entries in *Table 1.1* are meant to kickstart the configuration of your production and debug kernels and get a brief idea regarding their effect.

Once you're done generating the new debug kernel config, let's back it up as follows:

```
cp -af .config ~/lkd_kernels/kconfig_dbg01
```

Build it, as before `make -j8 all` (Adjust the parameter to `-j` based on the number of CPU cores on your box). When done, check out the compressed and uncompressed kernel image files:

```
$ ls -lh arch/x86/boot/bzImage vmlinuz
-rw-r--r-- 1 letsdebug letsdebug 18M Aug 20 12:35 arch/x86/boot/bzImage
-rwxr-xr-x 1 letsdebug letsdebug 1.1G Aug 20 12:35 vmlinuz
$
```

Did you notice? The size of the `vmlinuz` uncompressed kernel binary image file is huge; how come? All the debug features plus all the kernel symbols account for this large size...

Finish off with installing the kernel modules, initramfs, and bootloader update, as earlier:

```
sudo make modules_install && sudo make install
```

Great; now that you're done configuring both the production and debug kernels, lets briefly examine the difference between the configurations.

Seeing the difference – production and debug kernel config

It's enlightening – and really, it's the key thing within this particular topic - to *see the differences between our original production and the just-built debug*

kernel configuration. This is made easy via the convenience script `scripts/diffconfig`; from within the debug kernel source tree, simply do this to generate the `diff`:

```
scripts/diffconfig ~/lkd_kernels/kconfig_prod01 ~/lkd_kernels/kcon1
```

View the output file in an editor, seeing for yourself the changes we wrought in configuration. There are indeed many deltas – on my system, the `diff` file exceeds 200 lines. Here’s a partial look of the same on my system (I use the ellipse `[...]` to denote skipping some output):

```
$ cat kconfig_diff_prod_to_debug.txt
-BPF_LSM y
-DEFAULT_SECURITY_APPARMOR y
-DEFAULT_SECURITY_SELINUX n
-DEFAULT_SECURITY_SMACK n
[ ... ]
```

The `-` (minus sign) prefixing each of the above lines indicates that we removed this kernel config feature from the debug kernel. Continuing with the output:

```
DEBUG_ATOMIC_SLEEP n -> y
DEBUG_BOOT_PARAMS n -> y
DEBUG_INFO n -> y
DEBUG_KMEMLEAK n -> y
DEBUG_LOCK_ALLOC n -> y
DEBUG_MUTEXES n -> y
DEBUG_PLIST n -> y
DEBUG_RT_MUTEXES n -> y
DEBUG_RWSEMS n -> y
DEBUG_SPINLOCK n -> y
[ ... ]
```

In the preceding code snippet, you can clearly see the change made from the production to the debug kernel; for example, the first line tells us that the kernel config named `DEBUG_ATOMIC_SLEEP` was disabled in the production kernel and we’ve no enabled it (`n->y`) in the debug kernel! (Note that it will be prefixed with `CONFIG_`, that is, it will show up as `CONFIG_DEBUG_ATOMIC_SLEEP` in the kernel config file itself).

Here, we can see how the suffix to the name of the kernel – the config directive named `CONFIG_LOCALVERSION` – has been changed between the two kernels, besides other things:

```
LKDTM n -> m
LOCALVERSION "-prod01" -> "-dbg01"
LOCK_STAT n -> y
MMIOTRACE n -> y
MODULE_SIG y -> n
[ ... ]
```

The + prefix to each line indicates the feature that has been added to the debug kernel:

```
+ARCH_HAS_EARLY_DEBUG y
+BITFIELD_KUNIT n
[ ... ]
+IKCONFIG m
+KASAN_GENERIC y
[ ... ]
```

In closing, it's important to realize:

- The particulars of the kernel configuration we're performing here – for both our production and debug kernels - *is merely representative*; your project or product requirements might dictate a different config
- Many, if not most, modern embedded Linux projects typically employ a sophisticated builder tool or environment; Yocto and Buildroot are two common de facto examples. In such cases, you will have to adapt the instructions given here to cater to using these build environments (in the case of Yocto, this can become a good deal of work in specifying an alternate kernel configuration via a BB-append-style recipe).

By now, am furtively hoping you've absorbed this material and indeed, built yourself two custom kernels – a production and a debug one. If not, I request you to please do so before proceeding further along.

So, great, well done! By now, you have both custom 5.10 LTS production and debug kernels ready to rip. We'll certainly make use of them in the coming chapters. Let's finish this chapter with a few debug 'tips' that I hope you'll find useful.

Debugging – a few quick tips

I'll start off by saying this: debugging is both a science and an art, refined with experience, the mundane hands-on debugging through to identifying a bug and its

experience, we humans often struggle enough to identify a bug and its root cause, and (possibly) to fix it. I'm of the opinion that the following few debug tips are really nothing new; that said, we do tend to get caught up in the moment and often miss the obvious. I hope you'll find these useful and return to these tips time and again!

- **Assumptions - just say NO!**

Churchill famously said “*Never, never, never, give up*”. We say “*Never, never, never, make assumptions*”.

Assumptions are, very often, the root cause behind many, many bugs and defects. Think back, re-read the section *Software bugs – a few actual cases!*

In fact (hey, am partially joking here), just look at the word ‘ASSUME’: *it just begs to say: “Don’t make an ASS out of U and ME” !*

Using assertions in your code (we shall cover this), is a great way to catch assumptions.

- **Don’t lose the forest for the trees!**

At times, we do get lost in the twisted mazes of complex code paths; in these circumstances, it’s really easy to lose sight of the large idea, the objective of the code. Try and *zoom out*, think of the bigger picture. It often helps spot the faulty assumption(s) that led to the error(s). Well written documentation can be a life saver.

- **Think small:** When faced with a difficult bug, try this: build / configure / get the *smallest possible version* of your problem (statement) to execute causing the issue or bug you’re currently facing to surface; this often helps you track down the root cause of the problem. In fact, very often (in my own experience), the mere act of doing this – or even just the detailed jotting down of the problem you face – triggers your seeing the actual issue and its solution in your mind!
- **“It requires twice the brain power to debug a piece of code than to write it”:** This paraphrased quote is by Brian Kernighan in the book *The Elements of Programming Style*. So, should we not use our full brain power while writing code? Ha, of course you should... But, debugging *is* typically

harder than writing code. The real point is this: take the trouble to first carefully do your groundwork; write a brief very high-level design document, write what you expect the code to do, at a high level of abstraction. Then move into the specifics (with a so-called Low Level Design doc). Good documentation will save you one day (and blessings shall be showered upon you!).

Reminds me of another quote: *An ounce of design is worth a pound of refactoring*, Karl Wiegers.

- **Employ a “Zen Mind, Beginner’s Mind”:** Sometimes, the code can become too complex (spaghetti-like; it just *smells*). In many cases, just giving up and starting from scratch again, if viable, is perhaps the best thing to do.

This *Zen - Beginner’s Mind* also implies that we at least temporarily stop our (perhaps over-egotistical) thought patterns (*I wrote this so well, how can it be wrong!?*) and look at the situation from the point of view of somebody completely new to it. A good night’s rest can do wonders.

It is, in fact, one key reason why a colleague reviewing your code can spot bugs you’d never see!

- **Variable naming, comments:** I recall a Q&A on Quora revealing that *the hardest thing a programmer does* is naming variables well! This is truer than it might appear at first glance. Variable names stick; choose yours carefully. As with commenting, don’t go overboard either: a local variable for a loop index? `int i` is just fine (`int theloopindex` is just painful). The same goes for comments: they’re there to explain the rationale, the design, behind the code, what it’s designed and implemented to achieve, *not* how the code works. Any competent programmer can figure that out.
- **Ignore logs at your peril!** It’s self-evident perhaps, but we can often miss the obvious when under pressure... carefully checking kernel (and even app) **logs** often reveals the source of the issue you might be facing. Logs are usually able to be displayed in reverse-chronological order and give you a view of what actually occurred; Linux’s `systemd journalctl(1)` utility is powerful; learn to leverage it!
- **Testing can reveal the presence of errors but not their absence:** A truism, unfortunately. Still, testing and QA is simply one of the most

critical parts of the software process; ignore it at your peril! Time and the trouble taken to write exhaustive test cases – both positive and negative – pays off large dividends in the long run, helping make the product or project a grand success. Negative test cases and *fuzzing* are critical for exposing (and subsequently fixing) security vulnerabilities in the codebase. One hundred percent code coverage is the objective!

- **Incurring technical debt:** Every now and then, you realize deep down that though what you've coded works, it's not been done well enough (perhaps there still exist corner cases that will trigger bugs or undefined behavior); that nagging feeling that perhaps this design and implementation simply isn't best. The temptation to quickly check it in and hope for the best can be high, especially as deadlines loom! Please don't; there is really a thing called **technical debt**. It will come and get you.
- **Silly mistakes:** If I had a penny for each time I've made really silly mistakes when developing code, I'd be a rich man! For instance, I once spent nearly half a day racking my head on why my C program would just refuse to work correctly; until I realized am editing the correct code but compiling an old version of it – I was performing the build in the wrong directory! (I am certain you've faced your share of such frustrations). Often, a break, a good night's sleep, can do wonders.
- **Empirical model:** The word *empirical* means to validate something (anything) by actual and direct observation or experience rather than relying on theory. So, don't believe the book (this one is an exception of course!), don't believe the tutorial, the tutor or author: try it out and see for yourself!

Years (decades, actually) back, on my very first day of work at a company I joined, a colleague emailed me a document that I still hold dear: *The Ten Commandments for C Programmers*, by Henry Spencer (<https://www.electronicsweekly.com/open-source-engineering/linux/the-ten-commandments-for-c-programmers-2009-04/>). Do check it out. In a clumsily similar manner, I present a quick checklist for you.

A programmer's checklist – seven rules

Very important! Did you remember to:

- Check all APIs for their failure case

Compile with warnings on (definitely with `-Wall` and possibly `-Wextra`)

and eliminate all warnings as far as is possible

- Never trust (user) input; validate it
- Eliminate unused (or dead) code from the codebase immediately
- Test thoroughly; 100% code coverage is the objective. Take the time and trouble to learn to use powerful tools: memory checkers, static and dynamic analyzers, security checkers (checksec, lynis, and several others), fuzzers, code coverage tools, fault injection frameworks, and so on. Don't ignore security!
- With regard to kernel and especially drivers, after eliminating software issues, be aware that (peripheral) hardware issues could be the root cause of the bug, don't discount it out of hand! (One learns this the hard way)
- Do *not* assume anything (*ASSUME: makes an ASS out of U and ME*); using assertions helps catch assumptions, and thus bugs

We shall certainly be elaborating on several of these points in the coming material.

Summary

Firstly, congratulations on completing this, our first chapter; getting started is half the battle! You began by learning a bit about how the word **debug** came to be – equal parts myth, legend and truth...

A key section was the brief description of some complex real-world cases of software gone wrong (several of them very unfortunate tragedies), where a software bug (or bugs) proved to be a key factor behind the disaster.

You understood that we're using the latest (at the time of this writing) 5.10 LTS kernel and how to setup the workspace (on x86_64, using either a native Linux system or Linux running as a guest OS). We covered the configuring and building of two custom kernels – a **production** and a **debug** one, with the production kernel geared towards high performance and security whereas the debug one is configured with several (most) kernel debug features turned on, in order to help catch bugs. I will assume you've done this for yourself, as future chapters will depend on it.

Finally, and I think very important, a few debugging tips and a small checklist, wrapped up this chapter. I urge you to read through the tips and checklist often.

In the next chapter, you will learn the basics of user mode debugging, how to use a few very useful tools, instrument your code and leverage a *better* Makefile. See you there soon!

Further reading

- Real-world stories of software going wrong – software horror stories:
 - *SOFTWARE HORROR STORIES*: An old page, but still (mostly) valid and very interesting! Many, many incidents have been covered here; do take a gander (<http://www.cs.tau.ac.il/~nachumd/horror.html>)
 - Patriot missile battery failure: <https://www-users.cse.umn.edu/~arnold/disasters/patriot.html>
 - Ariane 5 launcher crash:
 - The official report - ‘ARIANE 5 – Flight 501 Failure’, by the Inquiry Board: <http://sunnyday.mit.edu/nasa-class/Ariane5-report.html>
 - An excellent article on the same: *Design by Contract: The Lessons of Ariane*, Jean-Marc Jézéquel, Bertrand Meyer (the creator of the Eiffel programming language): <https://archive.eiffel.com/doc/manuals/technology/contract/ariane/>
 - Mars Pathfinder reset issues:
 - *Priority inversion*: https://en.wikipedia.org/wiki/Priority_inversion
 - *What really happened on Mars ?*, Glenn Reeves detailed reply to Mike Jone’s summary of the issue : https://www.cs.unc.edu/~anderson/teach/comp790/papers/mars_p
 - *What the Media Couldn’t Tell You About Mars Pathfinder*, Tom Durkin, 1998; PDF: <https://people.cis.ksu.edu/~hatcliff/842/Docs/Course-Overview/pathfinder-robotmag.pdf>
 - *Now showing on satellite TV: secret American spy photos*, The Guardian, 13 June 2002: <https://www.theguardian.com/media/2002/jun/13/terrorismandthemedia>
 - *Software problem kills soldiers in training incident*, 13 June 2002 : <http://catless.ncl.ac.uk/Risks/22.13.html#subj2.1>
 - Boeing 737 MAX and the MCAS:
 - *The inside story of MCAS: How Boeing’s 737 MAX system gained power and lost safeguards*, The Seattle Times, 22 June 2019: [The inside story of MCAS: How Boeing’s 737 MAX system gained power and lost safeguards](#)

<https://www.seattletimes.com/seattle-news/times-watchdog/the-inside-story-of-mcas-how-boeings-737-max-system-gained-power-and-lost-safeguards/>

- *Boeing 737 Max: why was it grounded, what has been fixed and is it enough?*, The Conversation, 28 Nov 2020:
<https://theconversation.com/boeing-737-max-why-was-it-grounded-what-has-been-fixed-and-is-it-enough-150688>
- Jack Ganssle's TEM (The Embedded Muse) newsletter – back issues:
<http://www.ganssle.com/tem-back.htm>; excellent newsletters, do check it out
- Kernel and system workspace setup:
 - Various good online articles and tutorials on installing Linux as a guest VM on Oracle VirtualBox can be found at:
https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Further_Reading.md#chapter-1-kernel-development-workspace-setup---further-reading
 - *Easy way to determine the virtualization technology of a Linux machine?*, StackExchange:
<https://unix.stackexchange.com/questions/89714/easy-way-to-determine-the-virtualization-technology-of-a-linux-machine>
 - Ubuntu Linux – the System Requirements page:
<https://help.ubuntu.com/community/Installation/SystemRequirements>
 - Kernel documentation: *Configuring the kernel* (<https://www.kernel.org/doc/html/latest/admin-guide/README.html#configuring-the-kernel>)
 - Article: *How to compile a Linux kernel in the 21st century*, S Kenlon, Aug 2019: <https://opensource.com/article/19/8/linux-kernel-21st-century>
 - Information on initrd / initramfs and the GRUB bootloader: from the *Further reading* notes from the *Linux Kernel Programming – Part 1* book : https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Further_Reading.md#chapter-3-building-the-linux-kernel-from-source---further-reading
- Customizing the GRUB bootloader:
 - *How do I add a kernel boot parameter?*
<https://askubuntu.com/questions/19486/how-do-i-add-a-kernel-boot-parameter>. Do realize, this tends to be x86_64 and Ubuntu specific...
- *The Ten Commandments for C Programmers*, Henry Spencer:
<https://www.electronicsweekly.com/open-source-engineering/linux/the-ten-commandments-for-c-programmers>

[commandments-for-c-programmers-2009-04/](#)

- Interesting:
 - *What is a coder's worst nightmare?*, Quora; answer by Mick Stute:
<https://www.quora.com/What-is-a-coders-worst-nightmare>
 - *Reflections on Trusting Trust*, Ken Thompson:
https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_Reflecti

2 Approaches to Kernel Debugging

We're here! From this chapter, for the remainder of the book, you shall delve into the exciting world of Linux kernel debugging.

Even a casual perusal of topics related to kernel debugging will quickly have you realize that there are many approaches to it, and correspondingly, many tools and techniques that can and are brought to bear on the problem. In this relatively short chapter, we'll first check out some ways of classifying bugs by type. Classifying defects or bugs by type will help you gain a high-level understanding of them and where they fall, and at times overlap, better. We shall classify bugs by various types or views: the classic view, by memory issues, by the security related view and finally, by typical issues caused within the Linux kernel.

Next, we consider why there are various approaches to kernel debugging, and then summarize exactly what these approaches are and when it's generally appropriate to use which approach. These topics will help lay the foundation to the remainder of the book, where we'll delve into learning how to employ each of these kernel debug approaches or techniques.

In this chapter we're going to cover the following main topics:

- Classifying bug types
- Kernel debugging – why there are different approaches to it
- Summarizing the different approaches to kernel debugging

Technical requirements

The technical requirements and workspace remain identical to what's described in *Chapter 1, A General Introduction to Debugging Software*, under the *Technical requirements* and *Setting up the workspace* sections.

Classifying bug types

As you will know, defects (bugs) can be quite readily classified into different

types. Here, I attempt to do so (it's nothing new, really), with an added twist: we shall look at common bug classes through different *lenses* or viewpoints – first, the classic (typical, academic) manner, next, focused on memory-type bugs, and then the security-related view of bugs. Having seen this, we'll further refine this classification to what you'll typically see when working with the Linux kernel. Do note that there can be, and often are, overlaps within these classifications. Let's begin with the first one – classifying defects/bugs in the classic manner.

Types of bugs – the classic view

The classic way of viewing the types of defects or bugs that can occur in a software program is as follows:

- **Logic or implementation errors:**
 - Includes off-by-one errors, infinite loops/recursion
 - **Arithmetic errors:** Includes loss of precision errors (recall the Patriot missile and the Ariane 5 incidents!), arithmetic underflow or overflow, division by zero
 - **Syntax defects:** this is quite obvious; defects like (in C) using the equals = operator instead of the == operator (modern compilers and static analysis tools should certainly catch these)
- **Resource leakage and generic defects on resources:**
 - Includes the classic NULL pointer dereference bug, memory bugs: **Uninitialized Memory Reads (UMR)**, leakage, double-free, **Use After Free (UAF)**, **Out Of Bounds (OOB)** buffer overflow errors – read/write underflow/overflow, stack memory overflow, access violations, and so on
 - **Hardware:** Don't forget about the hardware! Faulty RAM, DMA issues, hardware freezes, microcode bugs, hardware interrupt misses/spurious interrupts, key bouncing, data endian errors, data packing/padding issues, instruction faults, and so on (see the *Further Reading* section for an interesting post)
- **Races:** Data races, locking issues causing deadlock, livelocks (as with too many hardware interrupts in too short a time period; network drivers layer often use the so-called **New API (NAPI**) to mitigate precisely this)
- **Performance defects:**
 - Includes data (cacheline) alignment issues, data races, deadlocks and livelocks
 - Poorly chosen APIs (for example, blind usage of the kernel's page /

slab allocator APIs – like the `get_free_pages()` / `kmalloc()` - can lead to highly suboptimal memory usage due to wild amounts of external fragmentation issues (*wastage*, really). Another: using moderate to highly contended locks with long critical sections is just begging for performance issues (the usage of lock-free algorithms and APIs is going to help! Perhaps via the Linux kernel's **percpu** and **Read Copy Update (RCU)** lock primitives)

- Races (mentioned in the earlier bullet point; as noted, overlaps in the bug classification can and do occur)
- **Input/Output (I/O):** Suboptimal and heavy reads and writes cause major performance bottlenecks; this applies to both the filesystem and network layers; it's important to realize: the actual performance bottlenecks are to do with suboptimal I/O usage and typically aren't CPU-related
- There are more ways to classify bugs; we shan't go into any detail here, we shall merely mention them:
 - by Interface-based and even teamwork-based side effects.

An interesting paper presented at a conference on software engineering in Melbourne (ICSE, 1992) showed that defects (bugs) are introduced and removed at various rates at different points in the SDLC; interestingly, relatively high bug insertion rates occur in both the design and coding phases; it helps highlight the need for better design/architecture of the system (see the *Further reading* section for the link to the paper, and more).

Let's move along to another way, or viewport, the *memory defects* one.

Types of bugs – the memory view

As defects due to memory-related bugs are simply so common with procedural (and non-managed) languages like C, we'll now view defects from the viewpoint of **memory corruption**:

- Incorrect memory accesses:
 - Using variables uninitialized; aka UMR bugs
 - Out-of-bounds memory accesses (read/write underflow/overflow bugs)
 - Use-after-free/use-after-return (out-of-scope) bugs
 - Double-free
- Memory leakage

- Data Races
- Fragmentation (internal implementation) issues:
 - Internal
 - External

All these common memory issues (except fragmentation) are generically classified as **Undefined Behavior (UB)**. Though fragmentation is a memory issue, it's not a bug in the sense that we're concerned with, so we don't delve into it further.

You'll have noticed that many bug classes are repeated from the previous classification; yes, of course, that's expected. The reason I re-classify defects via memory corruption is to highlight it – it's definitely among the more common root causes of software issues!

Next, let's view bugs through the viewport of security-related ones.

Types of bugs – the CVE/CWE security-related view

There's an open database of publicly known security issues; it's used by security researchers, academicians and industry to track security-related defects/bugs and helps folks to study and discuss them, build mitigations (fixes, patches) and thus respond to them in a consistent manner. Each security bug (and at times a whole bunch of them, forming a class) is assigned a number called a **Common Vulnerabilities and Exposures (CVE)** or a **Common Weaknesses and Enumeration (CWE)** number.

There are several websites that categorizes CWEs and CVEs; among them is the U.S. based **National Institute of Standards and Technology (NIST)** and **National Vulnerability Database (NVD)** here (<https://nvd.nist.gov/vuln/full-listing>). It (among other things) provides a comprehensive categorization of software defects; I urge you to lookup the site, and especially, this page showing a subset of the CWE structure (<https://nvd.nist.gov/vuln/categories/cwe-layout>).

It's not just the NIST NVD; several other sites do too. Among them are the **CVE Details** site (<https://www.cvedetails.com/>); it provides excellent explanations alongside the CVE number. MITRE provides this service as well; this is its FAQ page: <https://cve.mitre.org/about/faqs.html>.

As a good example, many security-related bugs boil down to nothing but an implementation weakness or vulnerability, that of the well-known (very often stack-based) **Buffer Overflow (BoF)**. The CWE MITRE site carries a detailed explanation of the same here: *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')* (<https://cwe.mitre.org/data/definitions/120.html>), along with example code that demonstrates the vulnerabilities! Do check it out...

It's really important to realize that, at heart, many *security issues are mostly software defects – bugs!*

Types of bugs – the Linux kernel

It's also useful and relevant to look at bug types from the viewpoint of the Linux kernel itself. Paraphrasing from Sergio Prado's presentation *Linux Kernel Debugging: Going Beyond Printk Messages* (https://www.youtube.com/watch?v=NDXYpR_m1CU) he classifies Linux kernel bugs as follows:

- Defects (bugs) that cause the system to lockup or hang
- Defects that cause the system to crash and/or panic
- Logic or implementation defects
- Resource leakage defects
- Performance issues

All right, we've done the (perhaps rather dry) task of classifying bugs. I know what you're perhaps thinking: this is all rather academic, perhaps a bit pointless? Well, the idea is that now that you understand how bugs can be classified, we shall get to the important point: based on classification (and other ways), which tools/techniques can you employ to debug them.

But first, we need to also understand that not all debugging techniques or approaches may be suitable to the task; the following section briefs you on this.

Kernel debugging – why there are different approaches to it

When the kernel has an error, a bug, no matter how trivial or non-trivial, the entire system is considered to be in a bad unrecoverable state and a **kernel panic**

typically ensures – a fatal condition wherein the system generates a brief diagnostic and then simply halts (or, it can be configured to reboot after a timeout). Debugging these scenarios is inherently harder, as, at least on the surface, it appears as though there is no diagnostic information to work with, and even if there were, the system is unresponsive, essentially dead; then how does one retrieve diagnostic information in order to analyze it?

What you will soon realize, is that even though there are several techniques, tools and approaches to kernel debugging, all of them aren't suitable to any and all scenarios – the tools or techniques you use are often dictated by the particular scenario you find yourself in.

So what are these scenarios? Broadly speaking, they include the following:

- **Development phase of the project:** You are in the process of developing the code, active development is ongoing; involves usage of both the custom debug and production kernels
- **Unit or individual developer testing and QA (integration/systems/acceptance) test phases:** You have developed a module or component and need to test it; involves usage of both the custom debug and production kernels
- **Post-mortem analysis:** The kernel has crashed; you need to try and figure out the root cause and fix it; involves usage of both the custom debug and production kernels
- **In-field or production:** The system is suffering from bugs and/or performance issues; you need to use appropriate tools to understand the underlying causes; usage of the custom production kernel (and debug kernel – where symbols are required – for some of the tools)

Finally, let's get to the nitty-gritty: the following section gives you a summary of the different approaches, actual tools, techniques, and APIs (if appropriate), to debugging the Linux kernel.

Summarizing the different approaches to kernel debugging

There are many approaches to kernel debugging; the one (or ones) to use depend upon the scenario. Here are the afore-mentioned scenarios and some approaches to kernel debugging when in them.



Development phase

Are you currently in the **development phase** of the project? If yes, the following approaches and techniques can help:

- Code-based debugging techniques can immediately help (although they're useful even later). These include:
 - Code-level instrumentation with `printk()` and friends.
 - Dynamic debug `printk`.
 - Generating a kernel-mode stack dump and interpreting it.
 - Using assertions within the code.
 - Setup and leverage debug hooks within the codebase; there are two typical ways to do this:
 - Via the **debugfs** pseudo filesystem
 - Via a special **ioctl(2)** hook function meant for debug purposes
- Single-stepping through the kernel (or module's) C code, setting breakpoints, watchpoints, examining the content of data, and so on: via the well-known **Kernel GDB (KGDB)** framework

Unit testing and/or QA phases

In the **unit testing and/or QA phases** (both unit and integration/systems/acceptance tests): here, you, in your capacity as an individual developer on the project, typically run unit tests against the code you've developed. Besides that, your team and/or a dedicated QA team might run a complete (perhaps automated) test suite against the project (interim) release, discover and report bugs back to the development team. The following tools and techniques should be used to try and catch possible bugs in these phases:

- **Dynamic analysis:** implies that you run tools that run on the live system, which perform checks on code paths as they're executed; these include:
 - Memory checkers: detecting memory issues or memory corruption (often the root cause of bugs) is critical
 - Undefined Behavior (UB) checkers; UB includes things like arithmetic underflows/overflows (including the well-known **Integer overFlow (IoF)** defect), invalid bit shifts, misaligned accesses, and so on

- Lock debugging tools and instrumentation
- **Static analysis:** involves employing tools that work upon the source code of the project (similar to the compiler). They can provide a great deal of insight into overlooked and possibly buggy, as well as security-wise risky code
- **Code coverage analysis:** this isn't really a debug technique; it's *to ensure that every line of code is actually exercised while testing is on. This is critical – only then can we have high confidence in the product.* Here, you typically employ code coverage tools (like **kcov** and **gcov**) to check which lines of code are actually executed during a given test run. (These techniques are typically / usually more applicable to individual developer unit testing than to system-level testing, though they can certainly be applied there as well)
- **Monitoring and tracing tools:** these can be employed in the development and testing/QA phases, and possibly even in production (in the field):
 - Kernel tracing infrastructure; this is a big area and includes:
 - **Ftrace and trace-cmd**
 - **Event Tracing**
 - The **Linux Tracing Toolkit: next generation (LTTng)**, kernelshark GUIs
 - **Perf**
 - **Enhanced Berkeley Packet Filter (eBPF)**
 - **SystemTap**
 - Usermode tracing infrastructure (often employing the powerful **strace** and **ltrace** utilities)
 - **Kernel probes (Kprobes)** - both static and dynamic
 - Watchdogs
 - Custom kernel panic handler
 - Magic SysRq handlers
- **Post-mortem analysis:** One of the common cases for most developers, is the *after-a-crash* case: (the capture and) analysis of a kernel diagnostic – called the kernel **Oops**:
 - Oops (kernel log files) analysis
 - Using **kdump** to collect a kernel dump image (loosely equivalent to the *core dump* produced by a process when it crashes due to a hardware-level fault), and the powerful **crash** application to interpret it
- **In production in-field runtime (mentioned for completeness):**
 - Any (or all) of the *monitoring and tracing tools* (mentioned in the previous bullet point)

- Debug hooks within the code (via debugfs, ioctl)
- Regular and dynamic debug printk’s
- Logging (via the **systemd** journal and app-based logging)
- Custom panic handler

Which kernel debug technology to use is not only dependent on the phases of the software lifecycle; some kernel debug techniques or technologies demand a significant amount of hardware and/or software resource availability:

- **Hardware constraints:** Some kernel debug techniques require significant amounts of hardware resource availability, which you may or may not be able to afford! For example, using the **kdump** technology requires significant amounts of RAM, network bandwidth and/or disk space; some tightly constrained embedded Linux systems just cannot afford this, whereas your typical server system can easily do so (the same goes for the well-known userspace Valgrind suite of tools; Address Sanitizer (ASAN) uses less resources...)
- **Software constraints:** just as with hardware, some systems have a self-imposed design limitation on what can be enabled in the kernel config which might preclude some debugging techniques. Again, kdump, and tracing infrastructure, are good examples of this.

A key point: dynamic analysis tools can only catch bugs within the code that they actually *see* and run; this leads us to understand that having test cases that cover all the code is extremely critical; as mentioned before (in *Chapter 1, A General Introduction to Debugging Software*) “*One hundred percent code coverage is the objective!*”.

Please do note, that though I’ve definitively categorized the tools and techniques, there certainly will arise cases where you can (and perhaps should) use a technique in a different scenario than has been shown above. *Keep it flexible, use what’s appropriate to the situation at hand.*

Categorizing into different scenarios

The tables that follow are an attempt at a catch-all of kernel debug approaches, tools, and techniques, categorized by different scenarios to use them in.

Do note:

- For now, just look at the available tools/techniques/technologies/APIs for different scenarios and use cases; don't worry about how exactly to use them. That, of course, is really the heart of the book, the coming chapters; the intent is to cover most of the ones mentioned here
- As already mentioned, the scenarios to use a given tool or technique are typical, not absolute; you might come across a use case that's different. I suggest you adapt and make use of whichever techniques seem appropriate.

We begin with a summary table (*Table 4.1*) for the scenario where **you're developing the kernel (or kernel / driver module) code**, the coding phase:

Debugging Approach	Tool(s) / Technique(s)	Specific
Code-based debug techniques: used within the kernel or module code	Instrumentation via the <code>printk()</code> and friends APIs	<code>printk()</code> , and so
Dynamic debug <code>printk(pr_debug() / dev_dbg() : can be <code>CONFIG_DYNAMIC_DEBUG</code> dynamically enabled/disabled per callsite)</code>		
Generating a kernel-mode stack dump and interpreting it	<code>[trace_]dump_stack()</code> ; kernel mode stack call trace interpretation	
Using assertions	<code>WARN[_ON[_ONCE]]()</code> , <code>WARN_TINT[_ONCE]()</code> , or a custom <code>assert()</code> macro	
Debug hooks	Via debugfs APIs, via a custom <code>ioctl()</code> method	
Interactive debug	Interactive debug using GDB, KGDB, KDB	Single-step, set structure (KDB) from additional verbs) with

Table 4.1 – Summary of kernel debug techniques for the “development / coding phase” scenario

Now let's check out a summary table of the kernel debugging tools and techniques that can be employed while **testing and QA phases**:

Debugging Approach	Tool(s) / Technique(s)	Specifics / APIs / tool names / front-ends
Dynamic Analysis	Kernel memory checker tools	Kernel Address SANitizer (KASAN), Undefined Behavior SANitizer (UBSAN), SLUB debug techniques, kmemleak
Undefined Behavior (UB) checkers (arithmetic over/underflows, ...)	UBSAN	
Lock debugging tools	Lockdep (kernel lock validator), various other kernel configs for lock debugging, locking statistics	
Static Analysis	Perform static analysis on kernel (or kernel module) source code	checkpatch.pl , sparse , smatch , Coccinelle , cppcheck , flawfinder , gcc
Code coverage	Perform code coverage analysis	kcov and gcov

Table 4.2 – Summary of kernel debug techniques for the “(Unit) testing / QA phases” scenario

Now that we’ve viewed several kernel debug tools and techniques by scenario, let’s view them by another couple of categories **tracing, monitoring, and profiling tools:**

Tool(s) / Technique(s)	Specifics / APIs / tool names / front-ends
Kernel tracing infrastructure	Ftrace , LTTng , perf-events and Perf , eBPF , SystemTap
User space tracing tools	strace and ltrace, uprobes*
Profiling tools	perf , perf-tools , and *bpfcc (eBPF)
In-production instrumentation	Static and Dynamic probes (Kprobes and kretprobes);

System monitoring

System-wide monitoring, panic handlers

Magic SysRq handlers

Table 4.3 – Summary of kernel debug techniques to do with system monitoring and tracing

This leaves only the **kernel image capture, Oops and post-mortem crash analysis tools and techniques**:

Tool(s) / Technique(s)	Specifics / APIs / tool names / front-ends
Kernel image dump generation and capture	Kdump
Kernel image dump analysis (or live kernel analysis)	crash and GDB (limited)
Analysis of kernel Oops within the kernel logs	Kernel Oops analysis
Logging	Kernel and user mode log analysis – captured via systemd journal; journalctl frontend
Netconsole : kernel log messages over the network	
Kernel / driver live patching	KGraft ; for example, can use live patching to add instrumentation

Table 4.4 – Summary of kernel debug techniques to do with Kernel image capture, Oops, post-mortem crash analysis and logging

I've included logging (and log analysis) as well in *Table 4.4* (instead of allocating an unnecessary separate table for it); *logs are a really important means* to ascertaining what happened on the system after a crash (for both user as well as kernel-level debugging).

Finally, the following table shows which kernel debug tools or techniques (or APIs) are effective for which types of kernel defects:

Type of kernel	Lockup Crash Logic /	Resource Performance
----------------	----------------------	----------------------

defect vs Debugging / Hang / Implementation Leakage Issues
techniques and tools Panic
to employ

	Code-based / interactive debugging ((dynamic) printk, netconsole, assertions, debugfs hooks, GDB, KGDB)	Y	Y	Y	?	N
Dynamic analysis (memory checkers: KASAN, UBSAN, SLUB debug, kmemleak; locking: lockdep, lock stats, ...)	Y	N	N		Y	Y
Static analysis (checkpatch.pl, sparse, smatch, cppcheck, Coccinelle)	N	N	Y		?	N
Monitoring and tracing tools (Ftrace, event tracing, LTTng, perf, eBPF, Kprobes; watchdog, panic handler, magic SysRq)	Y	?	?		?	Y
Post-mortem analysis (logging: kernel logs analysis, systemd logs; Oops interpretation, kdump, crash, GDB)	?	Y	?		N	N

Table 4.5 – Summary of kernel debug tools/techniques vs types of kernel defects

Legend:

- Y: yes, can/should be used

- N: no, avoid using it
- ?: it depends... **Your Mileage May Vary (YMMV)**

Again, these guidelines are definitely not written in stone; you should use your judgement and try different techniques as required.

Which kernel debug tools and techniques you enable on your production kernel is typically something you (or the platform/BSP team) have to decide, based on system constraints (both hardware and software), performance considerations, and so on. A good starting point for this is what we already covered back in *Chapter 1, A General Introduction to Debugging Software*, under the *A tale of two kernels* sections, which described in some detail how to go about configuring both a custom production as well as a custom debug kernel.

Summary

In this chapter you learned that there are many approaches to debugging the kernel. We even categorized them in a manner suitable to help you quickly decide which to use in what situation. This was one of the key points – every tool or technique will not be useful in every scenario or situation; for example, employing a powerful memory checker like KASAN to help find memory bugs is really useful during the development and unit testing phases but typically impossible during systems testing and production (as the production kernel will not be configured with KASAN enabled, but the debug kernel will).

You will also realize that both hardware and software constraints play a role in determining which kernel debug features can be enabled.

Further, we showed the various approaches, tools and techniques (even at times the API or tool names) to kernel debugging categorized via several tables; this can aid you in narrowing down your armory: which of them to use in which situation.

It's important to not be too rigid on deciding on which kernel debug tools/techniques to use for your project based solely on the tables we've shown here; keep it flexible and try different approaches for your situation until you find what clicks.

Good job on completing this chapter; in the next one we'll get down to brass

https://www.kernel.org/doc/html/v5.10.100/debugger.html

tacks and learn how to debug via the instrumentation approach!

Further reading

- *Estimating software fault content before coding*, Eick, Loader, et al, Proceedings of the International Conference on Software Engineering, June 1992: <https://dl.acm.org/doi/10.1145/143062.143090>
- A very in-depth and interesting academic article on UB: *Undefined Behavior in 2017*, Regehr, Cuoq, July 2017: <https://blog.regehr.org/archives/1520>
- NASA Study on Flight Software Complexity, March 2009: https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf. A deep and interesting read
- Security-related defect tracking via CWE/CVE; very useful to track security-related defects and gain an understanding of them
 - NIST NVD database – full listing: <https://nvd.nist.gov/vuln/full-listing>
 - CVE details: <https://www.cvedetails.com/>
 - CVE MITRE: <https://cve.mitre.org/>
- Hardware bug! *How a broken memory module hid in plain sight — and how I blamed the Linux Kernel and two innocent hard drives*, C Hollinger, Feb 2020: <https://towardsdatascience.com/how-a-broken-memory-module-hid-in-plain-sight-and-how-i-blamed-the-linux-kernel-and-two-innocent-ef8ce7560ecc>
- *Linux Kernel Debugging: Going Beyond Printk Messages* - Sergio Prado, Embedded Labworks, OSS/ELC Europe, May 2020, YouTube: https://www.youtube.com/watch?v=NDXYpR_m1CU. Excellent classification (and more) on kernel-level bugs
- *Debugging kernel and modules via gdb*, Linux kernel documentation: <https://www.kernel.org/doc/html/latest/dev-tools/gdb-kernel-debugging.html#debugging-kernel-and-modules-via-gdb>
- *The kernel debugging techniques for a device driver developer on arm64*, Christina Jacob, Oct 2019, Medium: <https://medium.com/@christina.jacob.koikara/the-kernel-debugging-techniques-for-a-device-driver-developer-on-arm64-fa984e4d2a09>

3 Debug via Instrumentation – `printf` and friends

Quick, think: how often have you interspersed `printf()` instances (or the equivalent) in your program in order to follow its progress as it executes code, and indeed, to see at approximately which point it (perhaps) crashes? Often, I’m guessing! Don’t feel bad at all, this is a really good debugging technique! It has a fancy name to boot: **instrumentation**.

What you’ve been doing is *instrumenting* your code, allowing you to see the flow (depending on the granularity of your print statements); this allows you to understand where it’s been. Often enough, this is all that’s required to debug many situations. Do recollect though, what we discussed in the previous chapter – a technique like instrumentation is typically useful in certain circumstances, not all. For example, a resource leak (such as a memory leak) defect, is difficult, if not impossible, to debug with instrumentation. For most other situations though, it’s a really useful technique!

In this chapter we’re going to understand how to instrument kernel (or driver) code, primarily using the powerful `printf()` – and friends – APIs. Further, we shall continue along this path in the following chapter as well, focusing our efforts on another kernel technology than can be used for instrumentation on production systems – **Kprobes**.

In this chapter, we shall focus upon and cover the following main topics:

- The ubiquitous kernel `printf`
- Leveraging the kernel `printf` for debug purposes
- Using the kernel’s powerful dynamic debug feature

These very practical topics are important: knowing how to efficiently debug via instrumentation can result in a quick cure for annoying bugs!

Technical requirements

The technical requirements and workspace remain identical to what's described in *Chapter 1, A General Introduction to Debugging Software*. The code examples can be found within the book's GitHub repository here: <https://github.com/PacktPublishing/Linux-Kernel-Debugging>.

The ubiquitous kernel `printf`

There's a good reason the famous and familiar **Kernighan and Ritchie (K&R)** *Hello, world* C program employs the `printf()` API: it's the preferred API via which any output is written to the screen (well, technically, to the **standard output** channel `stdout` of the calling process). After all, it's how we can actually *see* that our program is really doing something, right?

You will surely recall using this API when writing your very first C program; now, quick, did you write the code that incorporates the `printf()` function? No, of course not; then where is it? You know it: it's part of the (typically rather large) standard C library – **GNU libc (glibc)** on Linux. Pretty much every binary executable program on a Linux box automatically and dynamically links into this library; thus the `printf()` is pretty much always available! (On x86, doing `ldd $(which ps)` will have the useful `ldd` script show you the libraries that the `ps` app links into; one of them will be the standard C library `libc.so.*`; try it).

Except that the `printf()` isn't available within the kernel! Why? This itself is a key point: the Linux kernel does not use libraries – dynamically or statically – in the way user space applications do. There are what could perhaps pass as the equivalent: the `lib/` branch of the kernel source tree (peek at it here if you wish: <https://github.com/torvalds/linux/tree/master/lib>) contains many useful APIs that get built into the kernel image itself. Also, the kernel's framework for writing modules – the **Loadable Kernel Module (LKM)** – has facilities that kind of mimic the user mode library: the **module stacking** approach and the ability to link together several source files into a single kernel module object file (a `.ko`).

Information box

These facilities – the LKM framework, module stacking approach, the basic usage of the `printf()` API, and so on, are covered in detail in my earlier book *Linux Kernel Programming*.

So how is the kernel or driver developer expected to emit a message that can be seen and, even better, logged? Via the ubiquitous `printk()` API, that's how! We say this because the `printk()` (and friends) APIs can be used *anywhere* – within interrupt handlers (all sorts, `hardirq/softirq/tasklets`), process context, while holding a lock; it's SMP-safe.

For you, the reader of this book, I do assume that you understand the basic usage of the useful `printk()` API; so, I'll mostly skip over the very basics and instead explain a summary of typical basic usage, along with a few examples from the kernel codebase.

The `printk()` API's signature is as follows:

```
// include/linux/printk.h  
int printk(const char *fmt, ...);
```

If you're curious, the actual implementation is here within the kernel source: `kernel/printk/printk.c:printk()`.

Tip – browsing source trees

Efficiently browsing large code bases is an important skill; the modern Linux kernel source tree's **Source Lines Of Code (SLOCs)** are in excess of 20 million lines! Though you could go with the typical `find <ksrc>/ -name "*.[ch]" |xargs grep -Hn "<pattern>"` approach, it quickly gets tiresome.

Instead, please do yourself a big favor and learn to use powerful and efficient purpose-built code browsing tools like (exuberant!) **ctags** and **cscope** (you installed them when following directions in *Chapter 1, A General Introduction to Debugging Software*). In fact, for the Linux kernel they're built-in targets to the top-level `Makefile`; here's how you can build their index files for the kernel:

```
cd <kernel-src-tree>  
  
make -j8 tags  
  
make -j8 cscope
```

To build the indices for a particular architecture, set the environment

variable `ARCH` to the architecture name. For example, to build cscope indices for Aarch64 (ARM 64-bit): `make ARCH=arm64 cscope`

You'll find links to tutorials on using ctags and cscope in the *Further reading* section for this chapter.

Great; let's actually make use of the famous `printf()`; to do so, we'll begin by checking out the logging levels at which they can be emitted at.

Using the `printf` API's logging levels

Syntax-wise, the `printf` API usage is almost identical to that of the familiar `printf(3)`; the only immediate difference is the usage of a *logging level* prefixed with `KERN_<foo>` as the first token; here's a sample `printf` with the logging level set to `KERN_INFO`:

```
printf(KERN_INFO "Hello, kernel debug world\n");
```

First off, notice that the `KERN_INFO` is not a separate parameter; it's part of the format string being passed as the argument. Next, it's not a priority level; it's merely a marker to specify that this `printf` is being logged as an *informational* one. Utilities to view logs – like `dmesg(1)`, `journalctl(1)`, and even GUI tools like `gnome-logs(1)` – can subsequently be used to *filter log messages* by logging level.

The `printf` has eight available log levels (from `0` to `7`); you're expected to use the one appropriate to the situation at hand. We show them to you direct from the source; the comment to the right of each log level specifies the typical circumstances under which you're expected to use it:

```
// include/linux/kern_levels.h
[...]
#define KERN_EMERG    KERN_SOH "0" /* system is unusable */
#define KERN_ALERT     KERN_SOH "1" /* action must be taken immediate */
#define KERN_CRIT      KERN_SOH "2" /* critical conditions */
#define KERN_ERR       KERN_SOH "3" /* error conditions */
#define KERN_WARNING   KERN_SOH "4" /* warning conditions */
#define KERN_NOTICE    KERN_SOH "5" /* normal but significant conditions */
#define KERN_INFO      KERN_SOH "6" /* informational */
#define KERN_DEBUG     KERN_SOH "7" /* debug-level messages */
#define KERN_DEFAULT   ""          /* the default kernel log level */
[...]
```

You can see that the `KERN_<FOO>` log levels are merely strings ("0", "1", ..., "7") that get prefixed to the kernel message being emitted by `printf`; nothing more. (The `KERN_SOH` is simply the kernel **Start Of Header (SOH)** which is the value `\001`; the man page on the ASCII code, `ascii(1)`, shows that the numeric 1 (or `\001`) is the SOH character, a convention that is followed here).

What's the `printf` default log level?

Within the `printf()`, if the log level is not explicitly specified, what log level is the print emitted at? It's 4 by default, that is, `KERN_WARNING`. Note, though, that you are expected to always specify a suitable log level when using `printf`, or, even better, use the convenience wrapper macros of the form `pr_<foo>()` where `<foo>` specifies the log level (it's coming right up).

Further, the `kern_levels.h` header contains integer equivalents of the string `loglevel` we've just seen (`KERN_<FOO>`) as the macro's `LOGLEVEL_<FOO>` (fear not, we shall make use of it in the first example code that soon comes up!).

A quick introduction to the `pr_*(())` convenience macros will get us closer to the code; let's go!

Leveraging the `pr_<foo>` convenience macros

For convenience, the kernel provides simple wrapper macros over the `printf` of the form `pr_<foo>` (or `pr_*(())`) where `<foo>` specifies the log level; for example, in place of writing the code as:

```
printf(KERN_INFO "Hello, kernel debug world\n");
```

You can - and indeed should! - instead use:

```
pr_info("Hello, kernel debug world\n");
```

The kernel header `include/linux/printf.h` defines the following `pr_<foo>` convenience macros; *you're encouraged to use them* in place of the traditional `printf()`:

- `pr_emerg()`: `printf()` at log level `KERN_EMERG`
- `pr_alert()`: `printf()` at log level `KERN_ALERT`

- `pr_crit()`: `printk()` at log level `KERN_CRIT`
- `pr_err()`: `printk()` at log level `KERN_ERR`
- `pr_warn()`: `printk()` at log level `KERN_WARNING`
- `pr_notice()`: `printk()` at log level `KERN_NOTICE`
- `pr_info()`: `printk()` at log level `KERN_INFO`
- `pr_debug()` or `pr-devel()`: `printk()` at log level `KERN_DEBUG`

Here's an example of using the emergency `printk`!

```
// arch/x86/kernel/cpu/mce/p5.c
[...]
/* Machine check handler for Pentium class Intel CPUs: */
static noinstr void pentium_machine_check(struct pt_regs *regs)
{
    [...]
    if (lotype & (1<<5)) {
        pr_emerg("CPU#%d: Possible thermal failure (CPU on fire ?).\n"
    }
[...]
```

Is the processor on fire!? Whoops!

Fixing the prefix

In addition, there's a rather special macro, the `pr_fmt()`; it's used to generate a uniform format string for the `pr_*`() macros (and indeed for any `printk()`). So, by overriding it's definition, by (re)defining it as the *very first (non-comment) line* of a source file, you can guarantee prefixing a given format to all subsequent `pr_*`() macro and `printk()` API invocations. This can be very useful, especially in a debug context, allowing us to automatically prefix, say, the kernel module name, the function name and the line number to every single `printk`!

Let's check out an example; our very simple `printk_loglevels` kernel module demonstrates a couple of things:

- Using the `pr_fmt()` macro to prefix a custom string to every single `printk`
- Using the `pr_<foo>()` macros to emit `printk`'s at different logging levels

Don't forget

The code for this, and all, kernel / driver modules and demos presented in this book are available on its GitHub repo; for this particular demo, you can find the code here: https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch5/printk_loglevels. Next, when trying out the kernel modules here, please ensure that you *have booted into the custom debug kernel* (or even the default distro kernel is okay for now). Attempting to use our custom production kernel may not work – why not? This is as it's security configuration is possibly tight: it may not even allow you to try out a kernel module that isn't signed or if the signature can't be verified (more on this in the section *Trying our kernel module on the custom production kernel*).

Let's quickly check out the relevant code from the ch5/printk_loglevels/printk_loglevels.c file:

```
#define pr_fmt(fmt) "%s:%s():%d: " fmt, KBUILD_MODNAME, __func__, __  
#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
[...]  
static int __init printk_loglevels_init(void)  
{  
    pr_emerg("Hello, debug world @ log-level KERN_EMERG      [%d]\r  
    pr_alert("Hello, debug world @ log-level KERN_ALERT      [%d]\r  
    pr_crit("Hello, debug world @ log-level KERN_CRIT      [%d]\n'  
    pr_err("Hello, debug world @ log-level KERN_ERR      [%d]\n",  
    pr_warn("Hello, debug world @ log-level KERN_WARNING  [%d]\n'  
    pr_notice("Hello, debug world @ log-level KERN_NOTICE  [%d]\n'  
    pr_info("Hello, debug world @ log-level KERN_INFO      [%d]\n'  
    pr_debug("Hello, debug world @ log-level KERN_DEBUG      [%d]\r  
    pr-devel("Hello, debug world via the pr-devel() macro (eff (  
    return 0;                                /* success */  
}  
static void __exit printk_loglevels_exit(void)  
{  
    pr_info("Goodbye, debug world @ log-level KERN_INFO      [%d]\n'
```

A (partial) screenshot of trying out this code is shown as follows; do study the output:

```

-----
sudo insmod ./printk_loglevels.ko && lsmod|grep printk_loglevels
-----

Message from syslogd@dbg-LKD at Sep  8 16:23:49 ...
kernel:[53143.115411] printk_loglevels:printk_loglevels_init():34: Hello, debug world @ log-level KERN_EMERG [0]
printk_loglevels      20480  0
-----
sudo dmesg
-----
[53143.115411] printk_loglevels:printk_loglevels_init():34: Hello, debug world @ log-level KERN_EMERG [0]
[53143.115629] printk_loglevels:printk_loglevels_init():35: Hello, debug world @ log-level KERN_ALERT [1]
[53143.115802] printk_loglevels:printk_loglevels_init():36: Hello, debug world @ log-level KERN_CRIT [2]
[53143.115975] printk_loglevels:printk_loglevels_init():37: Hello, debug world @ log-level KERN_ERR [3]
[53143.116148] printk_loglevels:printk_loglevels_init():38: Hello, debug world @ log-level KERN_WARNING [4]
[53143.116154] printk_loglevels:printk_loglevels_init():39: Hello, debug world @ log-level KERN_NOTICE [5]
[53143.116160] printk_loglevels:printk_loglevels_init():40: Hello, debug world @ log-level KERN_INFO [6]
[53143.116167] printk_loglevels:printk_loglevels_init():41: Hello, debug world @ log-level KERN_DEBUG [7]
[53143.116173] printk_loglevels:printk_loglevels_init():42: Hello, debug world via the pr_devel() macro (eff @KERN_DEBUG) [7]
$ sudo rmmod printk_loglevels ; sudo dmesg |tail -n1
[53160.019525] printk_loglevels:printk_loglevels_exit():49: Goodbye, debug world @ log-level KERN_INFO [6]
$
```

Figure 5.1 – Screenshot showing output from our printk_loglevels kernel module

(By the way, I have used a simple wrapper bash script named `1km` – in the root of our source tree – to automate the build, load (`insmod(8)`), `lsmod(8)`, and `dmesg(1)`) of the kernel module; the invocation of the script isn't seen in the preceding screenshot though).

In the preceding code and screenshot, do notice:

- How, due to our `pr_fmt()` macro (in the first line of code), every `printk` is prefixed with the module name, function name, and line number
- The `pr_<foo>()` macros have emitted a `printk` at the relevant log level; even the log level integer equivalent is printed within parentheses on the extreme right
- Any `printk` at log level *emergency* (`KERN_EMERG`) is immediately displayed on all console devices; you can see the output in the preceding screenshot (see the line in the upper portion `Message from syslogd@dbg-LKD at ...`)
- The `dmesg` utility has the ability to conveniently color-code the log output, helping our human eyes to catch the more important kernel messages (so too does the powerful `journalctl` utility)
- To prevent serious **information leakage** issues – security, many recent distros configure the `CONFIG_SECURITY_DMESG_RESTRICT` to be on by default, thus requiring us to either use `sudo(8)` (or have the appropriate capability bits set) to view kernel logs via `dmesg`.

All right; now that we understand how to use the `printk()` API as well as the `pr_*`() macros, let's move onto figuring a key point: once emitted, where

exactly is the `printf()` / `pr_*`() / `dev_*`() output visible?

Understanding where the printf output goes

Without going into too many of the details (they're covered in my earlier *Linux Kernel Programming* book), let's quickly summarize this key point: we have issued several `printf`'s; where does the output actually go? The following table precisely shows this.

The first important thing to understand: unlike the `printf` user space family of APIs, the `printf` output does *not* go to `stdout`:

printf() When (and friends) outputs to	Additional info
Log buffer in memory (RAM)	Always <code>static char __log_buf[__LOG_BUF_LEN];</code> in RAM, volatile; designed as a ring buffer; gets overwritten when overflowed. Configurable via <code>CONFIG_LOG_BUF_SHIFT</code> (init/Kconfig); the default value of 17 yields a log buffer size of 128 KB (also affected by <code>CONFIG_LOG_CPU_MAX_BUF_SHIFT</code>)
Log file(s): modern	Always, by default on most systems (requires configuration)
Log file(s): traditional	Always, by default on most systems (requires traditional configuration)
Console device (we'll cover more on this in the <i>What</i>	On by default for log levels < 4 (that is, for <code>emerg/alert/crit/err</code>) on most systems (requires configuration) Controlled via kernel tunable <code>/proc/sys/kernel/printk</code>

*exactly is
the
console
device
section)*

Table 5.1 – Summary of where printk output locations

With modern Linux distros (including our x86_64 Ubuntu 20.04 LTS), **system daemon (systemd)** is the initialization framework used. Systemd is a pretty powerful (and intrusive!) framework, taking over many tasks on the OS; this includes bringing up system services, logging, core dump manipulation, the kernel/userspace udev feature, and more. The logging framework includes sophisticated features like log rotation, archival, and so on.

As well, on many modern distros, the traditional style logging does work along with the modern one; here, the files logged into for kernel printk's depends on the broad type of distro:

- Debian/Ubuntu type distros: /var/log/syslog
- Red Hat/Fedora/CentOS type distros: /var/log/messages

I'll also mention that the output of the kernel printk to the console device depends upon the log level that it's emitted at. The first number output by /proc/sys/kernel/printk specifies that all messages less than this value will appear on the console device (or devices). Recall, the lower the numeric value of the log level, the higher it's relative importance. For example, on our x86_64 Ubuntu 20.04 LTS:

```
$ cat /proc/sys/kernel/printk
4      4      1      7
```

The first number is 4, representing the log level below which messages will appear on the console (as well as get logged into the kernel log buffer and log files). In this case, we can conclude that all printk's at a logging level less than 4 – KERN_WARNING – will appear on the console. In other words, all printk's emitted at log levels KERN_EMERG , KERN_ALERT , KERN_CRIT , and KERN_ERR . This is useful as it displays only the more important log messages. Of course, as root, you can change this to be as you please.

Practically using the printk format specifiers – a few quick tips

Here's a few top-of-mind common `printk` *format specifiers* to keep in mind when writing portable code:

- For the `size_t`, `ssize_t` typedef's (which represent signed and unsigned integers respectively), use the `%zu` and `%zd` format specifiers respectively
- Kernel pointers:
 - use `%pk` for security (it will emit only hashed values, helps prevent info-leaks, a serious security issue)
 - `%px` for actual pointers (*don't do this in production!*)
 - `%pa` for physical (must pass it by reference)
- Raw buffer as a string of hex characters: `%*ph` (where `*` is replaced by the number of characters; use for buffers within 64 characters, use the `print_hex_dump_bytes()` routine for more); variations are available (see the kernel doc, link follows)
- IPv4 addresses with `%pI4`, IPv6 addresses with `%pI6` (variations too)

An exhaustive list of `printk` format specifiers, which to use when (with examples) is part of the official kernel documentation here:

<https://www.kernel.org/doc/Documentation/printk-formats.txt>. I urge you to browse through it!

Right, now that you understand the basics of using the `printk()` (and related `pr_*`() / `dev_*`()) macros), let's move onto more specifics on using the `printk` for the purpose of debugging.

Leveraging the `printk` for debug purposes

You might imagine that all one has to do to emit a debug message to the kernel log is simply to issue a `printk` at log level `KERN_DEBUG`. Yes, though there's (a lot) more to it; the `pr_debug()` (and `dev_dbg()`) macros are actually designed to be more than mere printers when the kernel's **dynamic debug** option is enabled. We shall learn this powerful aspect in the coming section *Using the kernel's powerful dynamic debug feature*.

In this section, let's first learn more on issuing a debug print, followed by slightly more advanced ways that help in the issuing of debug messages to the kernel log.

Writing debug messages to the kernel log

In the simple kernel module we covered in the previous section (`printk_loglevel`), let's relook at the couple of lines of code that emitted a kernel `printk` at the debug log levels:

```
pr_debug("Hello, debug world @ log-level KERN_DEBUG [%d]\n", LOGI  
pr-devel("Hello, debug world via the pr-devel() macro (eff @KERN_DEB
```

Both macros `pr_debug()` and `pr-devel()` issue a print to the kernel log at log level `KERN_DEBUG` *but only when the symbol DEBUG is defined!* If it isn't defined, they remain silent – no debug output appears. This is precisely what's required!

Module authors should avoid using the `pr-devel()` macro; it's meant to be used for kernel-internal debug `printk` instances whose output should never be visible in production systems.

Figure 5.1 revealed that the messages from them did indeed make it to the log; but in order to work, the `DEBUG` symbol needs to be defined; where was this done? – especially as it isn't defined in the code? The answer: we defined it within the module's `Makefile`; check it out (I've highlighted the key line below):

```
$ cd ch5/printk_loglevels ; cat Makefile  
[ ... ]  
# Set FNAME_C to the kernel module name source filename (without .c)  
FNAME_C := printk_loglevels  
PWD := $(shell pwd)  
obj-m += ${FNAME_C}.o  
# EXTRA_CFLAGS deprecated; use ccflags-y  
ccflags-y += -DDEBUG -g -ggdb -gdwarf-4 -Og -Wall -fno-omit-frame-pointer  
# man gcc: "...-Og may result in a better debugging experience"  
[ ... ]
```

By appending the value `-DDEBUG` to the `ccflags-y` variable, it gets defined in effect. The `-D` implies **define this symbol**; useful. Likewise, `-U` implies *undefine this symbol*. We typically employ these in the `Makefile` targets for the debug and production versions respectively, of the app, or as in this case, the kernel module. So, here, to generate the production version, simply comment out or delete the `ccflags-y` line; alternatively, and preferably, change it to:

```
cctags -y += -UDEBUG
```

This will undefine the symbol.

Important – building a kernel module for debug or production

The way your kernel module gets built is heavily influenced by the value that the `DEBUG_CFLAGS` variable gets set to. This variable is primarily set within the kernel's top-level `Makefile`; here, the value it obtains depends upon the kernel config `CONFIG_DEBUG_INFO`. When it's on (implying a debug kernel), various debug flags make their way into `DEBUG_CFLAGS` and thus your kernel module gets built with them. In effect, what I'm trying to emphasize here, is that the presence or absence of the `-DDEBUG` within your kernel module's `Makefile` (as we do here) does not much influence the way that your kernel module is built.

In effect, when you boot via your debug kernel and build your kernel modules, they automatically are built with symbolic info and various kernel debug options turned on; on the other hand, when booted via the production kernel, and (re)built therein, your kernel modules end up without debug information / symbols. As an example, when I built this kernel module (`printk_loglevels`) when on the debug kernel; then the `printk_loglevels.ko` file size was 221 KB, but when built on the production kernel, the size dropped to under 8 KB! (The lack of debug symbols and info, KASAN instrumentation, and so on, account for this major difference).

Quick tips: Doing `make v=1` to actually see all options passed to the compiler, can be very enlightening!.

Further, and *very useful*, you can leverage `readelf(1)` to determine the DWARF format debug information embedded within the binary

Executable and Linker Format (ELF) file. This can be particularly useful to figure out *exactly which compiler flags* your binary executable or kernel module has been built with. You can do so like this:

```
readelf --debug-dump <module_dbg.ko> | grep producer
```

Note that this technique typically works only when debug info is enabled; further, when working with a different target architecture (for example, arm), you'll need to run that toolchain's version: `${CROSS_COMPILE}readelf`. Do see the *Further reading* section for links to a series of articles on the **GNU**

Debugger (GDB) which describe this (and more) in detail (the second part in the series is the relevant one here).

Let's see an example of actual usage of the `dev_dbg()` within the kernel (drivers). An interesting, easy and very cool way of emitting output on typical embedded projects is via an **Organic Light-Emitting Diode (OLED)** device; they typically work over an **Inter Integrated Circuit (I2C)** bus, pretty much always available on embedded devices (like the popular Raspberry Pi or the BeagleBone). We'll take the SSD1307 OLED framebuffer driver as an example from this driver source file within the kernel source tree:

```
// drivers/video/fbdev/ssd1307fb.c
static int ssd1307fb_init(struct ssd1307fb_par *par)
{
    [ ... ]
    /* Enable the PWM */
    pwm_enable(par->pwm);
    dev_dbg(&par->client->dev, "Using PWM%d with a %lluns period\n"
            par->pwm->pwm, pwm_get_period(par->pwm));
}
```

As you can see, the first parameter to the `dev_dbg()` macro is a pointer to a device structure; here it happens to be embedded within an `i2c_client` structure (as this device is being driven over the popular I2C protocol), which itself is embedded within the driver's *context* structure (named `ssd1307fb_par`); this sort of thing is quite typical in drivers.

To make it more interesting, here's a photo of an SSD1306 OLED display panel in action (which the `ssd1307fb` driver can drive as well):

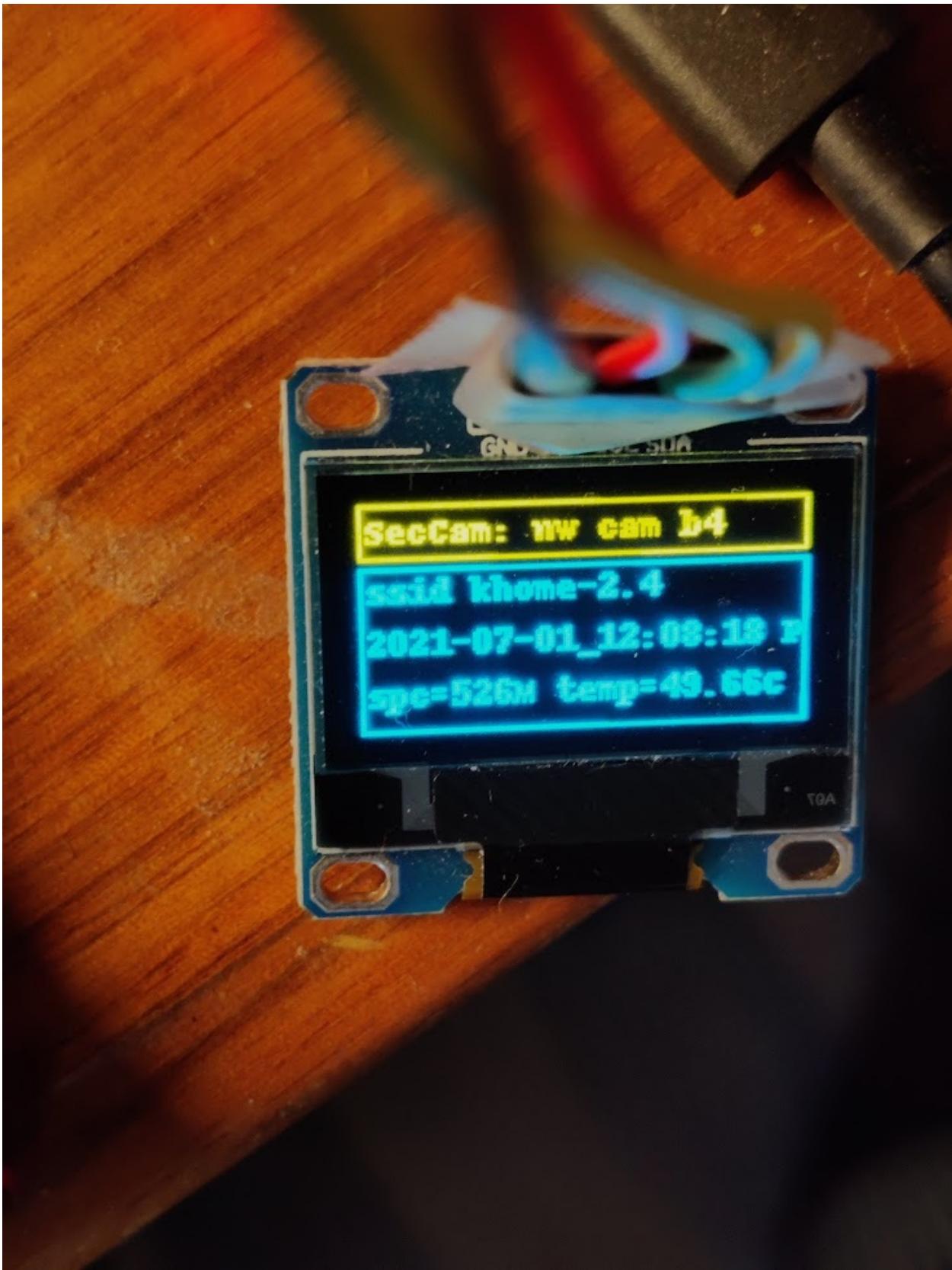


Figure 5.2 – An SSD1306 TFT display panel

Figure 5.2 An Arduino IDE display panel

As hinted at, there's much more we can do to leverage the kernel's dynamic debug framework... Before that though, and now that you know the basics of using the `printk` for debug, let's round this off with a few more practical tips on debugging with the `printk` and friends.

Debug printing – quick and useful tips

When working on the project or product, you'll perhaps need to generate some debug `printk`'s. The `pr_debug()` will get the job done (as long as the symbol `DEBUG` is defined). But think on this: to lookup the debug prints you will need to run `dmesg` over and over. Several tips on what you can do in this situation follow:

1. Clear the kernel log buffer (in RAM) with `sudo dmesg -c`; alternatively, `sudo dmesg -c` will first print the content and then clear the ring buffer; this way stale messages don't clog the system and you see only the latest ones when you run `dmesg`
2. Use `journalctl -f` to keep a *watch* on the kernel log (in a fashion similar to how `tail -f` on a file is used); try it out!
3. **Make the `printk` behave like the `printf` and see its output on the console!** We can do this by setting the console log level to the value `8`, thus ensuring that *all* `printk`'s (log levels `0` to `7`) will be displayed on the console device:

```
sudo sh -c "echo \"8 4 1 7\" > /proc/sys/kernel/printk "
```

(I often do this within a startup script when debugging kernel stuff. For example, on my Raspberry Pi, I keep a startup script that contains the following line:

```
[ $(id -u) -eq 0 ] && echo "8 4 1 7" > /proc/sys/kernel/printk
```

Thus, when it runs as root, this takes effect and all `printk` instances now directly appear on the `minicom(1)` (or whichever) console, just as `printf` output would).

Useful, yes!? But what about a very common case – when you're working on a device driver? The next section delves into the recommended way – how to use the `dev_dbg()` macro.

Device drivers – use the dev_dbg()

A key point for driver authors: when writing a device driver, you are expected to make use of the `dev_dbg()` macro to emit a debug message (and not the usual `pr_debug()`).

Why? The first parameter to this macro is `struct device *dev`, a pointer to `struct device`. This device structure is always present when writing a driver and serves to describe the device in detail; it's often embedded in a wrapper structure particular to the kind of driver being written. Printing via the `dev_dbg()` not only gets the debug `printk` across and into the kernel log (and possibly the console), it also typically has useful information prefixed to the message (like the name and (sometimes) class of the device, the major:minor numbers if appropriate, and so on).

An example from the kernel's **Network Block Device (nbd)** driver will serve to show how it's used (I searched, via `cscope`, for kernel code that calls `dev_dbg()` on the 5.10.60 kernel and got over 22,000 hits! An important reason is that it's used for *dynamic debug* as we'll shortly learn):

```
// drivers/block/nbd.c
dev_dbg(nbd_to_dev(nbd), "request %p: got reply\n", req);
```

Here the `nbd_to_dev()` inline function retrieves the device structure pointer from the `nbd_device` structure, where it's embedded.

In fact, this notion is carried to its logical conclusion: *when writing a driver, in place of the `pr_*`() macros, please use the equivalent `dev_*`() macros!* The header `include/linux/dev_printk.h` contains their definitions – the `dev_emerg()`, `dev_crit()`, `dev_alert()`, `dev_err()`, `dev_warn()`, `dev_notice()`, `dev_info()` and of course, as already covered, `dev_dbg()`. Everything remains as with the `pr_*`() macros except that the first parameter is a pointer to the device structure.

Trying our kernel module on the custom production kernel

As an experiment, boot into the custom production kernel we built back in *Chapter 1, A General Introduction to Debugging Software*. While running on

this production kernel, let's build and then attempt to load the kernel module (notice we're running as root):

```
# make
[...]
# dmesg -C; insmod ./printk_loglevels.ko ; dmesg
insmod: ERROR: could not insert module ./printk_loglevels.ko: Operation not permitted
[ 1933.232266] Lockdown: insmod: unsigned module loading is restricted
#
```

It fails due to the fact that, in our custom production kernel's configuration, we enabled the kernel **lockdown** mode (a recent kernel feature, from the 5.4 kernel, enabled via `CONFIG_SECURITY_LOCKDOWN_LSM=y`). This (and related) config options *disallow the loading of any kernel module that isn't signed or the signature cannot be validated by the kernel.*

This implies that we can't even test our kernel module on the production kernel? You can, in one of two ways:

- Actually sign the kernel module (official kernel documentation: Kernel module signing facility: <https://www.kernel.org/doc/html/v5.0/admin-guide/module-signing.html#kernel-module-signing-facility>).
(Also, FYI, with `CONFIG_MODULE_SIG_ALL=y` all kernel modules are auto-signed upon installation, during the `make modules_install` step of the kernel build).
- Or you can always disable these kernel configs, rebuild the kernel, reboot with it and then test; we do precisely this in the section *Disabling the kernel lockdown* that follows.

FYI, here's a link to the man page on the kernel lockdown feature: https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html.

All good with the debug prints, except – what is one to do when there are multiple voluminous printk's, especially in a high volume code path? The following section has you covered.

Rate limiting the printk

Let's take a plausible scenario: you're writing a device driver for some chipset

or peripheral device... Often, especially during development, and sometimes in order to debug in production, you of course intersperse your driver code with the now familiar `dev_dbg()` (or similar) macro. This works well until your code paths containing the debug prints turn out to run (very) often; what will happen? It's quite straight-forward:

- The kernel ring (circular) buffer isn't very large (typically between 64 KB to 256 KB, configurable at kernel build time); once full, it wraps around; this causes you to lose perhaps precious debug prints
- Debug (or other) prints in a very high volume code path – within interrupt handler routines and timers, for instance – can dramatically slow things down (especially on an embedded system with prints travelling across a serial line), even leading to a *livelock* situation (a situation where the system becomes unresponsive as the processor(s) are tied up working on logging stuff – console output, framebuffer scrolling, log file appends, and so on)
- The very same debug (or other) `printk` message being repeated over and over again umpteen times (for example, a warning or debug message within a loop) doesn't really help anyone
- Also, do realize that it's not just the `printk` (and similar) APIs that can lead to logging issues and failures; the usage of kprobes or indeed any kind of event tracing on high volume code paths can cause this same issue to crop up (we cover kprobes in the following chapter and tracing in later ones).

In such situations, you'll notice the message (typically from the `systemd-journald` process):

```
/dev/kmsg buffer overrun, some messages lost.
```

...or similar.

(By the way, if you're wondering what the `/dev/kmsg` character device node is all about, please do refer the kernel documentation here:
<https://www.kernel.org/doc/Documentation/ABI/testing/dev-kmsg>).

To mitigate exactly these situations, the community came up with the *rate-limited printk* – a means to throttle down and not emit prints (the same or different) when certain (tunable) thresholds have been exceeded!

We discuss these thresholds in just a moment... The kernel provides the

following macros to help you rate limit your prints/logging
(#include <linux/kernel.h>):

- `printk_ratelimited()` (*Warning!* Do not use it; the kernel warns against this)
- `pr_*_ratelimited()`: Where the wildcard `*` is replaced by the usual (`emerg`, `alert`, `crit`, `err`, `warn`, `notice`, `info`, `debug`)
- `dev_*_ratelimited()`: Where the wildcard `*` is replaced by the usual (`emerg`, `alert`, `crit`, `err`, `warn`, `notice`, `info`, `debug`)

Ensure you use the `pr_*_ratelimited()` macros in preference to `printk_ratelimited()`; driver authors should use the `dev_*_ratelimited()` macros.

But how exactly are the prints rate limited? The kernel provides two tunable thresholds via the usual control file interfaces within procfs (under the `/proc/sys/kernel` folder), named `printk_ratelimit` and `printk_ratelimit_burst` for this purpose. Here, we directly reproduce the sysctl documentation (from <https://www.kernel.org/doc/Documentation/sysctl/kernel.txt>) that explains the precise meaning of these two (pseudo) files:

```
printk_ratelimit:  
Some warning messages are rate limited. printk_ratelimit specifies 1  
A value of 0 will disable rate limiting.  
=====  
printk_ratelimit_burst:  
While long term we enforce one message per printk_ratelimit seconds,
```

On my x86_64 Ubuntu 20.04 LTS guest system, we find that their (default) values are as follows:

```
$ cat /proc/sys/kernel/printk_ratelimit  
5  
$ cat /proc/sys/kernel/printk_ratelimit_burst  
10
```

This implies that, by default, *a burst of up to 10 printk messages occurring within a 5-second time interval can make it through before rate limiting kicks in and further messages are suppressed (until the next time interval).*

The printk rate-limiter code, when it does suppress kernel printk instances, emits

a helpful message mentioning exactly how many earlier printk callbacks were suppressed.

We write a simple kernel module to test printk rate limiting (again, I only show the relevant snippets here):

```
// ch5/ratelimit_test/ratelimit_test.c
#define pr_fmt(fmt) "%s:%s():%d: " fmt, KBUILD_MODNAME, __func__, __
[...]
#include <linux/kernel.h>
#include <linux/delay.h>
[...]
static int num_burst_prints = 7;
module_param(num_burst_prints, int, 0644);
MODULE_PARM_DESC(num_burst_prints, "Number of printk's to generate :");
static int __init ratelimit_test_init(void)
{
    int i;
    pr_info("num_burst_prints=%d. Attempting to emit %d printk's in "
           "for (i=0; i<num_burst_prints; i++) {"
           "    pr_info_ratelimited("[%d] ratelimited printk @ KERN_INFO [%c]"
           "    mdelay(100); /* the delay helps magnify the rate-limiting effect */"
           "}"
           "return 0; /* success */"
}
```

If you build and run this module with defaults, not modifying the `num_burst_prints` module parameter (it defaults to the value 7), you can see that we emit seven rate-limited printk's in a short time interval. This, in spite of the 100 milliseconds delay (the delay is deliberate – you will soon see its effect).

Let's push it a bit: we test by passing the module parameter `num_burst_prints` setting its value to some number greater than the maximum allowed burst (the value of `/proc/sys/kernel/printk_ratelimit_burst`; 10 by default); we set it to 60. The screenshot shows what happens at runtime:

```

# make; rmmod ratelimit_test; dmesg -C; insmod ./ratelimit_test.ko num_burst_prints=60 ; dmesg ; echo -n "# of printk's actually seen: " ; dmesg |grep "ratelimited printk @" |wc -l

--- Building : KDIR=/lib/modules/5.10.60-prod01/build ARCH= CROSS_COMPILE= EXTRA_CFLAGS=-DDEBUG -g -ggdb -gdwarf-4 -Wall -fno-omit-frame-pointer -DDYNAMIC_DEBUG_MODULE ---

make -C /lib/modules/5.10.60-prod01/build M=/home/letsdebug/Linux-Kernel-Debugging/ch5/ratelimit_test modules
make[1]: Entering directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
make[1]: Leaving directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
[14855.679081] ratelimit_test:ratelimit_test_init():40: num_burst_prints=60. Attempting to emit 60 printk's in a burst:
[14855.681387] ratelimit_test:ratelimit_test_init():44: [0] ratelimited printk @ KERN_INFO [6]
[14855.782887] ratelimit_test:ratelimit_test_init():44: [1] ratelimited printk @ KERN_INFO [6]
[14855.883286] ratelimit_test:ratelimit_test_init():44: [2] ratelimited printk @ KERN_INFO [6]
[14855.983924] ratelimit_test:ratelimit_test_init():44: [3] ratelimited printk @ KERN_INFO [6]
[14856.084340] ratelimit_test:ratelimit_test_init():44: [4] ratelimited printk @ KERN_INFO [6]
[14856.184749] ratelimit_test:ratelimit_test_init():44: [5] ratelimited printk @ KERN_INFO [6]
[14856.285232] ratelimit_test:ratelimit_test_init():44: [6] ratelimited printk @ KERN_INFO [6]
[14856.385645] ratelimit_test:ratelimit_test_init():44: [7] ratelimited printk @ KERN_INFO [6]
[14856.486079] ratelimit_test:ratelimit_test_init():44: [8] ratelimited printk @ KERN_INFO [6]
[14856.586458] ratelimit_test:ratelimit_test_init():44: [9] ratelimited printk @ KERN_INFO [6]
[14860.688772] ratelimit_test_init: 40 callbacks suppressed
[14860.688773] ratelimit_test:ratelimit_test_init():44: [50] ratelimited printk @ KERN_INFO [6]
[14860.789403] ratelimit_test:ratelimit_test_init():44: [51] ratelimited printk @ KERN_INFO [6]
[14860.889742] ratelimit_test:ratelimit_test_init():44: [52] ratelimited printk @ KERN_INFO [6]
[14860.990279] ratelimit_test:ratelimit_test_init():44: [53] ratelimited printk @ KERN_INFO [6]
[14861.090667] ratelimit_test:ratelimit_test_init():44: [54] ratelimited printk @ KERN_INFO [6]
[14861.191045] ratelimit_test:ratelimit_test_init():44: [55] ratelimited printk @ KERN_INFO [6]
[14861.291560] ratelimit_test:ratelimit_test_init():44: [56] ratelimited printk @ KERN_INFO [6]
[14861.391897] ratelimit_test:ratelimit_test_init():44: [57] ratelimited printk @ KERN_INFO [6]
[14861.492243] ratelimit_test:ratelimit_test_init():44: [58] ratelimited printk @ KERN_INFO [6]
[14861.592568] ratelimit_test:ratelimit_test_init():44: [59] ratelimited printk @ KERN_INFO [6]
# of printk's actually seen: 20

```

Figure 5.2 – Screenshot showing our ratelimit_test LKM in action

The preceding screenshot should make it clear: we attempt to emit 60 printk's in a burst – but of course it's the rate-limited version of the printk (via the `pr_info_ratelimited()` macro). The kernel's limit gets hit after just 10 printk's (the default value of `/proc/sys/kernel/printk_ratelimit_burst`); thus, the kernel now prevents or suppresses further prints – this is clearly seen: you can see prints [0] to [9] - 10 of them being issued and then the message:

40 callbacks suppressed

After that, sufficient time elapsed (5 seconds here, as the `/proc/sys/kernel/printk_ratelimit` value is 5 by default) that the prints resumed! Our usage of the `mdelay(100)` helped create sufficient delay so that prints can resume... So, out of the 60 attempted prints, only 20 actually made it to the log (or console). This is a good thing and clearly demonstrates the point. As root, you can modify the rate-limit sysctl parameters to suit your requirement.

The Ftrace `trace_printk()`

The kernel's powerful Ftrace subsystem (which we shall cover in a later chapter), provides another way to mitigate high volume logging issues: the `trace_printk()` API. The syntax is identical to the regular `printf()` (not

`printk()`!). It has two major advantages over the typical `printk`: one, it's very fast (as it only writes to a RAM buffer); two, the size of the trace buffer is large by default and tunable by root.

So, in conclusion, if you have a code path with a high volume of `printk`'s, you can mitigate the potential ill effects by either employing the rate-limiting `printk` (and/or macros) or by using the `trace_printk()` (more on the latter in a later chapter).

So, by now you have the skills and knowledge to emit a debug `printk` (typically via the `pr_*[_ratelimited]()` or `dev_*[_ratelimited]()` macros)! It seems this is sufficient, until you learn about and start using the kernel's pretty awesome dynamic debug framework. This is precisely what follows; read on, learn!

Using the kernel's powerful dynamic debug feature

The **instrumentation** approach to debugging – interspersing your kernel (and module) code with many `printk`'s is indeed a good technique, it helps you narrow things down and debug them! Yes, but as you've no doubt realized, there can be a (pretty high) cost to this:

- It eats into your disk (or flash) space as logs get filled in; this can be especially problematic on constrained embedded systems; also, writing to disk is *much* slower than writing to RAM
- It's fast in RAM, but the ring buffer is not that large and would thus quickly get overwhelmed; older prints will soon be lost
- Even more important, on many production systems, a high volume of `printk`'s would have an adverse performance impact, creating bottlenecks and possible livelocks! Rate limiting helps with this, to some extent...

A solution would be to use the `pr_debug()` and/or the `dev_dbg()` APIs! They're especially useful during development and testing as it's really easy to turn these debug `printk`'s on or off: the presence of the `DEBUG` symbol implies the debug `printk` will run (and be logged), its absence implies it won't.

That's great; however, think on this: when running in production (using the production kernel), the `DEBUG` symbol will almost certainly be undefined by default. Now let's say you have a situation, while running in production, where

you want your debug prints for a given kernel module to appear, and thus get logged. Changing the code (or `Makefile`) to define the `DEBUG` symbol, then recompiling and re-installing it is very unlikely to be allowed during production.

So, what do you do (besides give up)? There are two broad approaches to *dynamically* toggling debug prints: one, via module parameters, and two, via the kernel's powerful built-in dynamic debug facility – the latter being the superior one and very much the focus of this section. First though, let's briefly check out the first option.

Dynamic debug via module parameters

One approach is to use a **module parameter** to hold a `debug` predicate; keep it off by default (the value `0`). You can define it like this:

```
static int debug;
module_param(debug, int, 0644);
```

This has the kernel setup the module's parameter named `debug` under the sysfs pseudo filesystem (at `/sys/module/<module_name>/parameters/debug`, with the owner and group as root and the octal permissions as specified in the third parameter to the `module_param` macro).

Interestingly, the i8042 keyboard and mouse controller driver (very often found in x86-based laptops) does precisely this; it defines this module parameter:

```
// drivers/input/serio/i8042.c
static bool i8042_debug;
module_param_named(debug, i8042_debug, bool, 0600);
MODULE_PARM_DESC(debug, "Turn i8042 debugging mode on and off");
```

This has the OS set up a module parameter named `debug` (notice the usage of the `module_param_named()` macro to achieve this), which is a Boolean and off (false) by default. A given module's parameters can be easily seen by leveraging the `modinfo(8)` utility; for example, let's lookup the parameters you can supply to the hid driver:

```
$ modinfo -p /lib/modules/5.10.60-prod01/kernel/drivers/hid/hid.ko
debug:toggle HID debugging messages (int)
ignore_special_drivers:Ignore any special drivers and handle all de
```

Once back to the i8042 driver once loaded in you can see its parameters.

Okay, back to the `i8042` driver, once loaded up, you can spot it's parameters.

```
$ ls -l /sys/module/i8042/parameters/debug  
-rw----- 1 root root 4096 Oct  3 07:42 /sys/module/i8042/parameters/debug
```

Of course, this sysfs-based pseudo file will only be seen after the module has been loaded into memory, for its lifetime.

Notice the permissions; in this case, only root can read or write to the debug pseudo file:

```
$ sudo cat /sys/module/i8042/parameters/debug  
[sudo] password for letsdebug: xxxxxxxxxxxx  
N
```

The root user can always turn it on dynamically, by writing the value `Y` (or `1`) into the sysfs pseudo-file representing it! This way, one can dynamically turn on or off debugging. So, here, to turn debugging on at runtime, do, as root of course:

```
# echo "Y" > /sys/module/i8042/parameters/debug
```

And turn it off again with:

```
# echo "N" > /sys/module/i8042/parameters/debug
```

Simple. In fact, think on this, you can easily extend this idea: one way to do so, is to use an integer debug parameter, which, depending on its value, will have the module emit debug messages at various levels of verbosity. (For example, `0` means all debug messages are off, `1` implies only a few key debug prints will be emitted, `2` implying more debug verbosity, and so on).

This general approach does work, but, with some significant drawbacks, especially when compared to the kernel's dynamic debug facility:

- Performance – you will require a conditional statement of some sort (`an if, case, ...`) to check whether a debug print should be emitted or not, every time; with levels of verbosity, more checking is required
- With the kernel's dynamic debug framework (which is covered next), you get several advantages:
 - The formatting of debug messages with useful information prefixed is part of the feature set, with a gentle learning curve

- Performance remains high, with next to no overhead when debugging is off (typically the default in production). This is achieved by sophisticated dynamic code patching techniques that the kernel employs (as is the case for Ftrace as well)
- It's always part of the mainline kernel (from way back, the 2.6.30 kernel), not requiring home-brewed solutions which may or may not be maintained or available.

So, for the remainder of this section, we shall focus on learning to use and leverage the kernel's powerful **dynamic debug** framework, available right since the 2.6.30 kernel. Read on!

When the kernel config option `CONFIG_DYNAMIC_DEBUG` is enabled, *it allows one to dynamically turn on or off debug prints that have been compiled into the kernel image as well as within kernel modules*. This is done by having the kernel always compile in all `pr_debug()` and `dev_dbg()` callsites; now, the really powerful thing is that you can not only enable or disable these debug prints, but do so at various levels of scope: at the scope of a given source file, a kernel module, function, even a line number.

This does imply that the kernel image will be larger; it's not by too much, approximately a 2% increase in kernel text size. If this is a concern (on a tightly constrained embedded Linux, perhaps), you can always just set the kernel config `CONFIG_DYNAMIC_DEBUG_CORE`. This enables the core support for dynamic `printk`'s but it only takes effect on kernel modules that are compiled with the symbol `DYNAMIC_DEBUG_MODULE` defined. Thus, our module `Makefile` always defines it; you could always comment it out if you don't wish to have dynamic debug facility for that kernel module. This is the relevant line within our module `Makefile`:

```
# We always keep the dynamic debug facility enabled; this
# allows us to turn dynamically turn on/off debug printk's
# later... To disable it simply comment out the following
# line
ccflags-y += -DDYNAMIC_DEBUG_MODULE
```

In fact, it's not just the `pr_debug()`; *all the following APIs can be dynamically enabled/disabled per callsite*: `pr_debug()`, `dev_dbg()`, `print_hex_dump_debug()`, and `print_hex_dump_bytes()`.

Specifying what and how to print debug messages

Specifying what and how to print debug messages

As with many facilities, control over the kernel's dynamic debug framework – deciding which debug printk's are enabled and what extraneous information is prefixed to them – is decided via a **control file**. Where's this control file then? It depends, if the debugfs pseudo filesystem is enabled within the kernel config (typically it is, with `CONFIG_DEBUG_FS=y`), and the kernel configs `CONFIG_DEBUG_FS_ALLOW_ALL=y` and `CONFIG_DEBUG_FS_DISALLOW_MOUNT=n` – usually the case for a debug kernel – then the control file is here: `/sys/kernel/debug/dynamic_debug/control`.

On many production environments though, for security reasons, the debugfs filesystem is present (functional) but invisible (it can't be mounted) via the `CONFIG_DEBUG_FS_DISALLOW_MOUNT=y`.

In this case, the debugfs APIs work just fine but the filesystem isn't mounted (in effect it's invisible). Alternately, debugfs might be disabled altogether by setting the kernel config `CONFIG_DEBUG_FS_ALLOW_NONE` to `y`; in either of these cases, an identical but alternate control file under the pseudo proc filesystem (procfs) should be used: `/proc/dynamic_debug/control`.

As with other pseudo filesystems, this *control* file under debugfs or procfs is a pseudo-file; it exists only in RAM. It gets populated and manipulated by kernel code. Reading its content will give you a comprehensive list of all debug printk (and/or `print_hex_dump_*`) callsites within the kernel. Thus, its output is typically pretty large (over here, we're on the custom debug kernel and can hence use the debugfs location for the control file); let's begin to interrogate it:

```
# ls -l /sys/kernel/debug/dynamic_debug/control
-rw-r--r-- 1 root root 0 Sep 16 12:26 /sys/kernel/debug/dynamic_debug/control
# wc -l /sys/kernel/debug/dynamic_debug/control
3217 /sys/kernel/debug/dynamic_debug/control
```

Notice it's only writable as root (and we're running as root). Let's look up the first few lines of output:

```
# head -n5 /sys/kernel/debug/dynamic_debug/control
# filename:lineno [module]function flags format
drivers/powercap/intel_rapl_msr.c:151 [intel_rapl_msr]rapl_msr_probe
drivers/powercap/intel_rapl_msr.c:94 [intel_rapl_msr]rapl_msr_read_i
sound/pci/intel8x0.c:3160 [snd_intel8x0]check_default_spdif_aclink =
sound/pci/intel8x0.c:3156 [snd_intel8x0]check_default_spdif_aclink =
..
```

```
#
```

The format of each entry is shown first; it's reproduced here:

```
filename:lineno [module]function flags format
```

Besides the `flags` member, all are obvious; the last one, `format` is the actual printf-style format string that the debug print uses. So, let's zoom into the first actual entry seen and examine it minutely, with a (hopefully) helpful diagram:

```
drivers/powercap/intel_rapl_ms... 151 [intel_rapl_ms... rapl_ms... = "failed to register powercap control_type.\012"
```

Figure 5.3 – The dynamic debug control file format specifier

Here's the detailed breakup, as per the control file output format specifier:

- `filename`: `drivers/powercap/intel_rapl_ms.c` : It's the full pathname of the source file
- `lineno`: `151` : It's the line number within the source file, the place in code where the debug print lives (so complicated; yup, I can be sarcastic)
- `[module]`: `[intel_rapl_ms]` : The name of the kernel module where the debug print lives; it's optional: if the debug print callsite is in a kernel module, this – the module name - shows up in square brackets
- `function`: `rapl_ms_probe` : The function containing the debug print
- `flags`: `=` : Ah, this is really the interesting, juicy bit, we explain it shortly (*Table 5.2*).
- `format`: `"failed to register powercap control_type.\012"` : The actual printf-style format string that's to be printed/logged.

Just to fully verify this, here's the actual code snippet of this example from the kernel codebase (version 5.10.60; I've highlighted the relevant line – # 151 – below):

```
// drivers/powercap/intel_rapl_ms.c
149     rapl_ms_priv.control_type = powercap_register_control_type();
150     if (IS_ERR(rapl_ms_priv.control_type)) {
151         pr_debug("failed to register powercap control_type.\n");
152         return PTR_ERR(rapl_ms_priv.control_type);
153     }
```

You can see how it perfectly matches the control file's understanding of it.

(Interestingly, you can use Bootlin's online kernel code browser to look it up as well:

https://elixir.bootlin.com/linux/v5.10.60/source/drivers/powercap/intel_rapl_msr. useful!).

The real magic lies in the so-called `flags` specifier; using `flags`, you can program the dynamic debug framework to emit the debug print (thus having it logged) along with various useful prefixes. The following table summarizes how to program and interpret the `flags` specifier:

Dynamic debug control file: flags specifier value	Meaning
=<foo>	
-	The debug print is currently off; this is the typical default
p	The debug print is currently on and is printed and logged; the following specifiers can be added along with this one:
m	Module name is prefixed (if it's within a kernel module)
f	Function name is prefixed
l	Line number in source file is prefixed; lines range can be specified in from-to format
t	If in process context (not in any kind of interrupt), the PID of the thread that runs this code path is prefixed

Table 5.2 – Dynamic debug framework ‘flags’ specifier

In addition, quite intuitively, you can use these symbols to:

- + : Add the flag(s) specified
- - : Remove the flag(s) specified
- = : Set to the flag(s) specified

A quick experiment: let's `grep` for the number of debug printk's currently

enabled (notice how I use `sed` to strip away the first line, as it's the format string explanatory line and not an actual entry):

```
# cat /sys/kernel/debug/dynamic_debug/control |sed '1d' |wc -l  
3216
```

So, here and now, we have a total of 3,216 debug prints recognized by the kernel's dynamic debug framework. Now let's `grep` the flags, only matching the ones that are turned off:

```
# grep " =_ " /sys/kernel/debug/dynamic_debug/control |sed '1d' |wc  
3174
```

So, of the total 3,216 debug printk's in the kernel right now, 3,174 of them are turned off, leaving only $3216 - 3174 = 42$ turned on (by the kernel/drivers/whatever). Let's verify this, by negating the sense of the `grep`:

```
# grep -n -v " =_ " /sys/kernel/debug/dynamic_debug/control |wc -l  
42
```

It's verified. Of the ones that are turned on, here's the last three:

```
# grep -v " =_ " /sys/kernel/debug/dynamic_debug/control |tail -n3  
init/main.c:1340 [main]run_init_process =p " with arguments:\012"  
init/main.c:1129 [main]initcall_blacklisted =p "initcall %s blackli:  
init/main.c:1090 [main]initcall_blacklist =p "blacklisting initcall
```

So, as their `flags` value is `=p`, (just) the debug print will be logged when the line of code is hit; nothing will be prefixed to it.

Next, how does one program the dynamic debug framework? Very simple: just write the command often via the `echo` shell built-in into the control file! Needless to say, it will only go through with root access (or, with the better and modern capabilities model, having a capability bit like `CAP_SYS_ADMIN` set). The command syntax essentially is:

“`match-spec* flags`”

`match-spec` is one of (this, direct from the kernel documentation on dynamic debug here: <https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html#command-language-reference>):

```
match-spec ::= 'func' strina |
```

```

      'file' string |
      'module' string |
      'format' string |
      'line' line-range
line-range ::= lineno   | '-'lineno |
              lineno'-' | lineno'-'lineno
lineno ::= unsigned-int

```

The `flags` specifier has already been covered – see *Table 5.2*. Here’s a table summarizing how to use the `match-spec` to form a command, with examples:

match- Example of a ‘command’ [format: match-spec* flags]	Meaning
spec	
func run_init_process +p	Turn on run_init_process
func	
string	
file init/main.c +pf	Turn on init/main.c function
file	
string	
module module usbhid =pmflt	Turn on named module of the target
string	
format "Parser recognised the format (ret %d)\x012" +p	Turn on format "Parser recognised the format (ret %d)\x012"
format	
string	(\x012
file kexec_file.c line 90-446 +pf	Turn on kexec_file.c line 90-446 (in)
line	
string	

Table 5.3 – Dynamic debug framework ‘match-spec’ specifiers with examples

Issue the command, or program it like this:

```
# echo -n "<command string>" > <control-file>
```

Where the <command string> parameter to echo is the command formed in the match-spec* flags format and the <control-file> is either <debugfs-mount>/dynamic_debug/control or /proc/dynamic_debug/control.

Adding to this, several match specifications can be given in a single command; you can think of them as being implicitly ANDed to form a match to a subset of debug prints. You can even batch several commands into a file and write the file to the control file. More examples are available in the kernel documentation page on dynamic debug here: <https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html#examples>.

Exercising dynamic debug on a kernel module on a production kernel

For most of us module authors, using this powerful dynamic debug framework on our kernel module when it's running in production, will be a useful thing. A demo will help you understand how to do so. To make the demo a bit more realistic, let's boot up via our custom production kernel to help mimic an actual production environment.

Disabling the kernel lockdown

However, what if - as recommended back in the first chapter – you've configured the custom production kernel for security, by enabling (among other things) the kernel lockdown mode by default (by setting CONFIG_LOCK_DOWN_KERNEL_FORCE_CONFIDENTIALITY=y). (Hey, if this isn't the case, and you can load up your (and other third party) kernel modules on the production kernel, then all's well for this experiment and you can skip this section).

This is good for security, preventing one from loading unsigned kernel modules (along with other safety measures). But now that we'd like to test our kernel module on the production kernel, we will have to tweak the production kernel's configuration, setting the following within the make menuconfig UI:

- Under *Security options | Basic module for enforcing kernel lockdown*

- *Enable lockdown LSM early in init*: Set it to `n` (off)
- *Kernel default lockdown mode*: Set it to `None`
- Next, save the config, rebuild and reboot via the new production kernel
- At the (GRUB) bootloader screen, press a key, edit the kernel command line parameters, appending `lockdown=none`. This disables kernel lockdown mode.

FYI, for more details, please refer the man page on kernel lockdown:
https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html.

Now let's get that debug printk working dynamically!

Demonstrating dynamic debug on a simple misc driver

For the purposes of this demo, I grab a simple `misc` class character device driver from my earlier book, *Linux Kernel Programming – Part 2* (the original code's here: https://github.com/PacktPublishing/Linux-Kernel-Programming-Part-2/tree/main/ch1/miscdrv_rdwr). Of course, we shall keep a copy in this book's GitHub repo as well...

Looking at the code, you will notice several instances of the `dev_dbg()` macro being invoked; obviously these are the debug prints that will only get logged when `DEBUG` is defined or we use the kernel's dynamic debug facility – the latter being what this demo is all about!

Here's a sample of the debug prints in the driver (due to space constraints I don't show all the code here of course, only a few relevant bits):

```
// ch5/miscdrv_rdwr/miscdrv_rdwr.c
#define pr_fmt(fmt) "%s:%s(): " fmt, KBUILD_MODNAME, __func__
static int open_miscdrv_rdwr(struct inode *inode, struct file *filp)
{
    struct device *dev = ctx->dev;
    char *buf = kzalloc(PATH_MAX, GFP_KERNEL);
    [...]
    dev_dbg(dev, " opening \"%s\" now; wrt open file:
                  f_flags = 0x%lx\n", file_path(filp,
                  buf, PATH_MAX), filp->f_flags);
    kfree(buf);
    [...]
}
static ssize_t write_miscdrv_rdwr(struct file *filp, const char __u8
```

```

{
    int ret = count;
    void *kbuf = NULL;
    [...]
    dev_dbg(dev, "%s wants to write %zu bytes\n",
            get_task_comm(tasknm, current), count);
    [...]
    ret = count;
    dev_dbg(dev, " %zu bytes written, returning...
                (stats: tx=%d, rx=%d)\n",
                count, ctx->tx, ctx->rx);
    [...]
}
[...]

```

Note that the `Makefile` file for this module will conditionally set the `DEBUG` symbol to be undefined (as we're building in production mode). Thus the debug prints will *not* make it to the console or kernel logs.

A quick `mount | grep -w debugfs` shows no output, implying that the debugfs filesystem isn't visible. This, again, is intentional, a security feature we enabled for our custom production kernel by setting `CONFIG_DEBUG_FS_DISALLOW_MOUNT=y`. Don't panic (yet); as mentioned, there's a solution; simply make use of the control file available here: `/proc/dynamic_debug/control`.

Grepping it for our module before it's inserted into memory, reveals no data, as expected:

```
# grep miscdrv_rdwr /proc/dynamic_debug/control
#
```

Okay; we now get it running. The following screenshot shows the source files via `ls`, the build (via our convenience `lkm` script), the resulting `dmesg` output and the device node this driver creates:

```

$ ls
Makefile miscdrv_rdwr.c rdwr_test_secret.c
$ ..../lkm miscdrv_rdwr
Version info:
Distro: Ubuntu 20.04.3 LTS
Kernel: 5.10.60-prod01
-----
sudo rmmod miscdrv_rdwr 2> /dev/null
-----
^--[FAILED]
-----
sudo dmesg -C
-----
make || exit 1
-----
strip: './miscdrv_rdwr.ko': No such file
--- Building : KDIR=/lib/modules/5.10.60-prod01/build ARCH= CROSS_COMPILE= ccflags-y=-UDEBUG -DDYNAMIC_DEBUG_MODULE ---
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
make -C /lib/modules/5.10.60-prod01/build M=/home/letsdebug/Linux-Kernel-Debugging/ch3/miscdrv_rdwr modules
make[1]: Entering directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
  CC [M] /home/letsdebug/Linux-Kernel-Debugging/ch3/miscdrv_rdwr/miscdrv_rdwr.o
  MODPOST /home/letsdebug/Linux-Kernel-Debugging/ch3/miscdrv_rdwr/Module.symvers
  CC [M] /home/letsdebug/Linux-Kernel-Debugging/ch3/miscdrv_rdwr/miscdrv_rdwr/mod.o
  LD [M] /home/letsdebug/Linux-Kernel-Debugging/ch3/miscdrv_rdwr/miscdrv_rdwr.ko
make[1]: Leaving directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
-----
sudo insmod ./miscdrv_rdwr.ko && lsmod|grep miscdrv_rdwr
-----
miscdrv_rdwr      20480  0
-----
sudo dmesg
-----
[ 1854.991485] miscdrv_rdwr:miscdrv_rdwr_init(): LLKD misc driver (major # 10) registered, minor# = 58, dev node is /dev/llkd_miscdrv_rdwr
$ ls -l /dev/llkd_miscdrv_rdwr
crw-rw-rw- 1 root root 10, 58 Dec 26 09:24 /dev/llkd_miscdrv_rdwr
$
```

Figure 5.4 – Screenshot of our miscdrv_rdwr loading up on our custom production kernel

Also notice (in the preceding screenshot) that:

- The kernel version is `5.10.60-prod01`, showing that we're running on our custom production kernel
- The value of the `ccflags-y` variable is `-UDEBUG -DDYNAMIC_DEBUG_MODULE`, as expected.

With the current settings, the debug prints do not get logged; let's try this out and see (*remember: Be Empirical!*):

```
# echo "DEBUG undefined, no logging?" > /dev/llkd_miscdrv_rdwr
# dmesg
[ 9177.333822] miscdrv_rdwr:miscdrv_rdwr_init(): LLKD misc driver (r
#
```

As is expected, the kernel log (seen via `dmesg`) shows only the earlier

`printk` (which being a `pr_info()` does show up); *none of the debug prints appear*. So, let's setup to make them appear!

Now that our kernel module is loaded up, let's `grep` the dynamic debug control file again:

```
# grep "miscdrv_rdwr" /proc/dynamic_debug/control
<...>/miscdrv_rdwr.c:303 [miscdrv_rdwr]miscdrv_rdwr_init =_ "A sample
<...>/miscdrv_rdwr.c:242 [miscdrv_rdwr]close_miscdrv_rdwr =_ " filenar
<...>/miscdrv_rdwr.c:239 [miscdrv_rdwr]close_miscdrv_rdwr =_ "%03d) %
[...]
#
```

Clearly, the dynamic debug control is aware that our module has debug prints; it's currently off, the `flags` value of `=_` proving this (for readability, I've truncated the pathname and shown only the first few lines of output).

Now let's set it up such that any and all debug prints from our `miscdrv_rdwr` kernel module will get logged via the dynamic debug framework:

```
# echo -n "module miscdrv_rdwr +p" > /proc/dynamic_debug/control
```

You'll need to do this only once per session; the value is retained until the module is removed or a power cycle (or reboot) occurs. Now let's retry the `grep` command; the following screenshot shows our setting the debug prints on – by using the `+p` flags specifier syntax; the subsequent `grep` shows that this has been noticed and set up:

```
# echo "module miscdrv_rdwr +p" > /proc/dynamic_debug/control
# grep "miscdrv_rdwr" /proc/dynamic_debug/control
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:303 [miscdrv_rdwr]miscdrv_rdwr_init =p "A sample print via the dev_dbg(): driver initialized\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:242 [miscdrv_rdwr]close_miscdrv_rdwr =p " filename: \042%\042\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:239 [miscdrv_rdwr]close_miscdrv_rdwr =p "%03d) %c%s%c:%d | %c%c%c%u /* %s() */\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:217 [miscdrv_rdwr]write_miscdrv_rdwr =p "%zu bytes written, returning.. (stats: tx=%d, rx=%d)\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:181 [miscdrv_rdwr]write_miscdrv_rdwr =p "%s wants to write %zu bytes\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:175 [miscdrv_rdwr]write_miscdrv_rdwr =p "%03d) %c%s%c:%d | %c%c%c%u /* %s() */\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:152 [miscdrv_rdwr]read_miscdrv_rdwr =p "%d bytes read, returning.. (stats: tx=%d, rx=%d)\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:118 [miscdrv_rdwr]read_miscdrv_rdwr =p "%s wants to read (upto) %zu bytes\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:117 [miscdrv_rdwr]read_miscdrv_rdwr =p "%03d) %c%s%c:%d | %c%c%c%u /* %s() */\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:91 [miscdrv_rdwr]open_miscdrv_rdwr =p " opening \042%\042 now; wrt open file: f_flags = 0x%x\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/miscdrv_rdwr/miscdrv_rdwr.c:91 [miscdrv_rdwr]open_miscdrv_rdwr =p "%03d) %c%s%c:%d | %c%c%c%u /* %s() */\012"
#
```

Figure 5.5 – Screenshot showing setting on of debug prints for our `miscdrv_rdwr` module

Let's reprint and study the first line of output:

```
# grep "miscdrv_rdwr" /proc/dynamic_debug/control
<...>/miscdrv_rdwr.c:303 [miscdrv_rdwr]miscdrv_rdwr_init =p "A sample
```

This shows us that:

- Source line 303 is a debug print callsite; it also shows the source file pathname, the module and the function name, and then the actual print format string
- More importantly, between the function name and the format string you can see =p ; this implies of course that this debug print's callsite is known and, when this line of code is hit, the print will be emitted and logged!

To verify that this works, let's exercise our driver a bit (lazy fellow):

```
# echo "DEBUG undefined, dynamic debug now ON for this module" > /dev/llkd_miscdrv_rdwr
# dmesg
[ 608.317065] miscdrv_rdwr:miscdrv_rdwr_init(): LLKD misc driver (major # 10) registered, minor# = 58, dev node is /dev/llkd_miscdrv_rdwr
[ 1010.813690] miscdrv_rdwr:open_miscdrv_rdwr(): 001) bash :1080 | ...0 /* open_miscdrv_rdwr() */
[ 1010.813705] misc_llkd_miscdrv_rdwr: opening "/dev/llkd_miscdrv_rdwr" now; wrt open file: f_flags = 0x8241
[ 1010.813744] miscdrv_rdwr:write_miscdrv_rdwr(): 001) bash :1080 | ...0 /* write_miscdrv_rdwr() */
[ 1010.813750] misc_llkd_miscdrv_rdwr: bash wants to write 54 bytes
[ 1010.813758] misc_llkd_miscdrv_rdwr: 54 bytes written, returning... (stats: tx=0, rx=54)
[ 1010.813772] miscdrv_rdwr:close_miscdrv_rdwr(): 001) bash :1080 | ...0 /* close_miscdrv_rdwr() */
[ 1010.813777] misc_llkd_miscdrv_rdwr: filename: "/dev/llkd_miscdrv_rdwr"
# -
```

Figure 5.6 – Dynamic debug in action!

It works indeed! The preceding screenshot clearly shows the debug printk's have actually run and been logged.

Now let's turn it off:

```
# echo -n "module miscdrv_rdwr -p" > /proc/dynamic_debug/control
# grep "miscdrv_rdwr" /proc/dynamic_debug/control |head -n1
<...>/miscdrv_rdwr.c:303 [miscdrv_rdwr]miscdrv_rdwr_init =_ "A sample
```

And retry:

```
# echo "DEBUG undefined, dynamic debug now OFF for this module" > /...
# dmesg
[...]
[ 1010.813777] misc_llkd_miscdrv_rdwr: filename: "/dev/llkd_miscdrv_rdwr"
#
```

As expected, no debug prints have appeared (the one in the log is the earlier one – see the timestamp).

One more experiment; we turn on display of the module name (m) and thread context PID (t : shows the thread PID that runs this driver code in process

context):

```
# echo -n "module miscdrv_rdwr +ptm" > /proc/dynamic_debug/control
```

Write to the device node and check dmesg :

```
# echo "DEBUG undefined, dynamic debug now ON for this module" > /dev/mem
# dmesg
[...]
[ 1010.813777] misc l1kd_miscdrv_rdwr: filename: "/dev/l1kd_miscdrv_rdwr"
[ 1457.376915] [1080] miscdrv_rdwr: miscdrv_rdwr:open_miscdrv_rdwr()
[ 1457.376931] [1080] miscdrv_rdwr: misc l1kd_miscdrv_rdwr: opening...
[...]
#
```

Aha! This time you can see the PID of the thread that performed the write in square brackets ([1080] ; it's in fact the PID of our bash shell, as the echo is a bash built-in!) followed by the name of the module.

Super; you now know how to activate and deactivate debug prints on a production system using the kernel's dynamic debug framework.

Activating debug prints at boot and module init

It's important to realize that any debug prints within the early kernel initialization (boot) code paths or the initialization code of a kernel module, will *not automatically be enabled*. To enable it:

- For core kernel code and any built-in kernel modules, that is, for activating debug prints during boot, pass the kernel command line parameter dyndbg="QUERY" or module.dyndbg="QUERY" ; where QUERY is the dynamic debug syntax (explained earlier). For example, dyndbg="module myfoo* +pmft" will activate all debug prints within the kernel modules named myfoo* with the display as set by the flags spec pmft .
- For activating debug prints at kernel module initialization, that is, when modprobe myfoo is invoked (by systemd , perhaps): there are several ways to do this, by passing along module parameters (with examples):
 - Via /etc/modprobe.d/*.conf (put this in the /etc/modprobe.d/myfoo.conf file): options myfoo dyndbg=+pmft
 - Via kernel command line:

- ```
myfoo.dyndbg="file myfoobar.c +pmf; func goforit +mpt"
```
- Via parameters to modprobe itself: modprobe myfoo dyndbg==pmft  
(This, the = and not the +, overrides any previous settings!)

Interesting: the dyndbg is an always-available kernel module parameter, even though you don't see it (even in /sys/module/<modname>/parameters). You can see it by grepping the dynamic debug control file or /proc/cmdline.

(FYI, details on passing parameters to and auto-loading kernel modules have been fully covered in my earlier *Linux Kernel Programming* book).

The official kernel documentation on dynamic debug is indeed very complete; be sure to have a look: <https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html#dynamic-debug>.

## Kernel boot-time parameters

As an important aside, the kernel has an enormous (and useful!) number of kernel parameters that can be optionally passed to it at boot (via the bootloader). See the complete list here in the documentation: *The kernel's command-line parameters*: <https://www.kernel.org/doc/html/v5.10/admin-guide/kernel-parameters.html> (here, we've shown the link for the 5.10 kernel documentation).

While on the topic of the kernel command line, several other useful options with regard to printk-based debugging exist, enabling us to enlist the kernel's help for debugging issues concerned with kernel initialization. For example, the kernel provides the following parameters in this regard (taken directly from the link):

```
debug
[KNL] Enable kernel debugging (events log level).
[...]
initcall_debug
[KNL] Trace initcalls as they are executed. Useful for working out
[...]
ignore_loglevel
[KNL] Ignore loglevel setting - this will print /all/ kernel messages
```

Useful indeed; do try them out! The sheer volume of information posted is surprising at first; try and carefully and patiently analyze it.

We're almost done here; let's complete this chapter with some miscellaneous but

useful printk-related logging functions and macros.

## Remaining printk miscellany

By now you're familiar with most of the typical and pragmatic means to leverage the kernel's powerful and ubiquitous printk and its related APIs, macros, and frameworks. Of course, innovation never stops (especially in the opensource universe); the community has come up with more and more ways (and tooling) to use this simple and powerful tool. Without claiming to cover absolutely everything, here's what I think are the remaining and relevant tooling to do with the printk that we haven't had a chance to cover until now. Do check it out, it will probably turn out to be useful one day!

### Printing before console init – the early printk

You understand that the printk output can be sent to the console device of course (we covered this in the section *Understanding where the printk output goes* (see *Table 5.1*). By default, on most systems, it's configured such that all printk message of log level 3 and below (<4) are auto-routed to the console device as well (in effect, all kernel printks emitted at log levels `emerg/alert/crit/err` will find their way to the console device).

### What exactly is the console device

Before going any further, it's useful to understand what exactly the console device is... Traditionally, the console device is a pure kernel feature, the initial Terminal window that the superuser logs into (`/dev/console`) in a non-graphical environment. Interestingly, on Linux, we can define several consoles – a **teletype terminal (tty)** window (such as `/dev/console`), a text-mode VGA, a framebuffer, or even a serial port served over USB (this being common on embedded systems during development).

For example, when we connect a Raspberry Pi to an x86\_64 laptop via a USB-to-RS232 TTL UART (USB-to-serial) cable (see the *Further reading* section of this chapter for a blog article on this very useful accessory and how to set it up on the Raspberry Pi!) and then use `minicom(1)` (or `screen(1)`) to get a serial console, this is what shows up as the `tty` device – it's the serial port:

```
rpi # ttv
```

```
/dev/ttys0
```

Now, what's the problem? Let's find out!

## Early init – the issue and a solution

Okay, but think on this: very early in the boot process when the kernel is initializing itself, the console device isn't ready, it's not initialized and thus can't be used. Obviously, for any `printk`'s emitted at this early boot time, their output can't be seen on the *screen* – the console (even though it may be logged within the kernel log buffer; but we don't have a shell to look it up).

Pretty often (especially during things like embedded board bring-up), hardware quirks or failures can cause the boot to hang, endlessly probe for some non-existent or faulty device, or even crash! The frustrating thing is that these issues become hard to debug (to say the least!) in the absence of console – `printk` – output, which, if visible, can instrument the kernel's boot process and pretty clearly show where the issue(s) are occurring (recall, the kernel command line parameters `debug` and `initcall_debug` can be really useful at times like this; look back at the section *Kernel boot-time parameters* if you need to).

Well, as we know, necessity is the mother of invention: the kernel community came up with a possible solution to this issue – the so-called **early printk**. With it configured, kernel `printk`'s are still able to be sent to the console device. How? Well, it's pretty arch and device specific, but the broad and typical idea is that bare minimal console initialization is performed (this console device is called the `early_console`) and the string to be displayed on it is literally *bit-banged* out over a serial line one character at a time within a loop (with typical bitrates ranging between 9600 to 115,200 bps).

To make use of the facility involves doing three things:

- Configure and build the kernel to support the early `printk` (set `CONFIG_EARLY_PRINTK=y`); one time only
- Boot the target kernel with the appropriate kernel command line parameter – `earlyprintk=<value>`
- The API to use to emit the early `printk` is called `early_printk()` ; the syntax is identical to that of the `printf()`

Let's check out each of the above points briefly; first, configuring the kernel for early printk.

The kernel config for this feature tends to be arch-dependent. On an x86, you'll have to configure the kernel with `CONFIG_EARLY_PRINTK=y` (it's under the Kernel Hacking | x86 Debugging | Early printk menu; optionally, you can enable early printk via a USB debug port). The file that forms the UI - the menu system - for the kernel config (via the usual `make menuconfig`), for the kernel debug options is the file `arch/x86/Kconfig.debug`; we show a snippet of it – the section where the early printk menu option is:

```
config EARLY_PRINTK
 bool "Early printk" if EXPERT
 default y
 help
 Write kernel log output directly into the VGA buffer or to a serial
 port.
```

This is useful for kernel debugging when your machine crashes very early before the console code is initialized. For normal operation it is not recommended because it looks ugly and doesn't cooperate with klogd/syslogd or the X server. You should normally say N here, unless you want to debug such a crash.

*Figure 5.7 – Screenshot showing the early printk portion of the Kconfig.debug file*

Reading the `help` screen shown here is indeed helpful! As it says, this option isn't recommended by default as the output isn't well formatted and can interfere with normal logging; you're typically only to use it to debug an early init issue. (If interested, you'll find the details on the kernel's *Kconfig* grammar and usage in my earlier *Linux Kernel Programming* book).

On the other hand, on an ARM (Aarch32) system, the kernel config option is under the Kernel Hacking | Kernel low-level debugging functions (read `help!`) with the config option being called `CONFIG_DEBUG_LL`. As the kernel insists, let's read the help screen:

The screenshot shows a terminal window with the title ".config - Linux/arm 5.4.70 Kernel Configuration". The menu path is > Kernel hacking. A sub-menu titled "Kernel low-level debugging functions (read help!)" is open. Inside this sub-menu, the option "CONFIG\_DEBUG\_LL:" is selected. The description for this option states: "Say Y here to include definitions of printascii, printch, printhex in the kernel. This is helpful if you are debugging code that executes before the console is initialized." It also notes: "Note that selecting this option will limit the kernel to a single UART definition, as specified below. Attempting to boot the kernel image on a different platform \*will not work\*, so this option should not be enabled for kernels that are intended to be portable." Below the description, the symbol information is provided: "Symbol: DEBUG\_LL [=y]", "Type : bool", "Prompt: Kernel low-level debugging functions (read help!)". The location is listed as "Location: -> Kernel hacking" and "Defined at arch/arm/Kconfig.debug:103". It also depends on "DEBUG\_KERNEL [=y]". At the bottom right of the terminal window, there is a progress bar showing "( 99% )". At the bottom center, there is a button labeled "< Exit >".

Figure 5.8 - Screenshot showing the make menuconfig UI menu for early printk on an ARM-32

Do take note of what it says! Further, the sub-menu following it allows you to configure the low-level debug port (it's set to the EmbeddedICE DCC channel by default; you can change it to a serial UART if you have one available).

Okay, that's as far as kernel config goes, a one-time thing. Next, enable it by passing the appropriate kernel command line parameter – `earlyprintk=<value>`. The official kernel documentation shows all possible ways to pass it (here: <https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html>):

```
earlyprintk= [X86, SH, ARM, M68k, S390]
 earlyprintk=vga
 earlyprintk=sclp
 earlyprintk=xen
 earlyprintk=serial[,ttySn[,baudrate]]
 earlyprintk=serial[,0x...[,baudrate]]
 earlyprintk=ttySn[,baudrate]
```

```
earlyprintk=dbgp[debugController#]
earlyprintk=xdbc[xhciController#]
```

(The remaining paragraphs of the kernel doc are useful to read as well).

The optional, `keep` parameter implies that `printk` messages sent via the early `printk` facility, even after the VGA subsystem (or whatever the real console is) begins to post messages, are kept as well.

Once the `earlyprintk=` parameter is passed, the kernel is primed to use it (essentially redirecting `printk`'s onto a serial or VGA or whatever console you specified via this parameter). To emit a print, simply invoke the `early_printk()`; here's an example within the kernel codebase:

```
// kernel/events/core.c
if (!irq_work_queue(&perf_duration_work)) {
 early_printk("perf: interrupt took too long (%lld > %lld),
 "kernel.perf_event_max_sample_rate to %d\n",
 __report_avg, __report_allowed,
 sysctl_perf_event_sample_rate);
}
```

What we've described above is mostly the arch-independent stuff. As an example, (only) on the x86, you can leverage a USB debug port (provided your system has one), as follows. Pass the kernel command line parameter `earlyprintk=dbgp`. Note that it requires a USB debug port on the (x86) host system and a NetChip USB2 debug port key/cable (to connect to the client or target system). The kernel documentation details this facility here:

<https://www.kernel.org/doc/html/latest/x86/earlyprintk.html#early-printk>.

## Designating the `printk` to some known presets

The kernel provides macros that give ones the ability to prefix – and thus designate – a `printk` as a firmware bug or warning, a hardware error, a message regarding a deprecated feature, and so on. This is specified via some kernel-defined macros; the value of the macro – a string (for example, `"[Firmware Bug]: "`) – is what will be prefixed to the message you're emitting:

```
// include/linux/printk.h
#define FW_BUG "[Firmware Bug]: "
#define FW_WARN "[Firmware Warn]: "
```

```

#define FW_INFO "[Firmware Info]: "
[...]
/*
 * HW_ERR
 * Add this to a message for hardware errors, so that user can repro
 */
#define HW_ERR "[Hardware Error]: "
/*
 * DEPRECATED
 * Add this to a message whenever you want to warn user space about
 */
#define DEPRECATED "[Deprecated]: "

```

Be sure to read the useful comments atop each of these.

An example or two of their usage follows:

```

// drivers/acpi/thermal.c
static int acpi_thermal_trips_update(struct acpi_thermal *tz, int f:
{
[...]
/*
 * Treat freezing temperatures as invalid as well; some
 * BIOSes return really low values and cause reboots at startup. Be
[...] */
} else if (tmp <= 2732) {
 pr_warn(FW_BUG "Invalid critical threshold (%llu)\n", tmp);
[...]

```

Here's another example of a printk issuing a *deprecated* warning (notice the use of rate-limiting as well!):

```

// net/batman-adv/debugfs.c
pr_warn_ratelimited(DEPRECATED "%s (pid %d) Use of debugfs file \\"%s

```

Let's move along to the next point...

## Printing exactly once

To emit a printk exactly once, use the macro `printk_once()`; it guarantees it will emit the message exactly once. No matter how many times you call it (thus making it similar to macros like `WARN_[ON]_ONCE()`).

As with the *usual* `pr_*` macros, their equivalents are defined for printing a

message exactly once: `pr_*_once()`! The wildcard `*` is replaced by the usual log levels (`emerg`, `alert`, `crit`, `err`, `warn`, `notice`, `info`, `debug`).

The IPv4 TCP code has an example of using the `pr_err_once()`:

```
// net/ipv4/tcp.c
[...]
if (unlikely(TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN))
 offset--;
```

Driver authors: you'll realize that the equivalent macros exist in `include/linux/dev_printk.h`: you're expected to use the `dev_*_once()` (in place of the `pr_*_once()`) macros. As one example, this i.MX53 **Real Time Clock (RTC)** chip driver uses it:

```
// drivers/rtc/rtc-mxc_v2.c
if (!--timeout) {
 dev_err_once(dev, "SRTC_LPSCLR stuck! Check your hw.\n");
 return;
}
[...]
```

Then – again, dear driver authors, do take note - there's the `dev_WARN()` and `dev_WARN_ONCE()` macros; the kernel comment explains it clearly:

```
// include/linux/dev_printk.h
/*
 * dev_WARN*() acts like dev_printk(), but with the key difference (
 */
```

Do think before using these `[pr|dev]_*_once()` macros; they're to be used when you want to emit a message exactly once.

## Exercise

How many instances of using the `dev_WARN_ONCE()` can you find within the 5.10.60 kernel codebase? (Tip: use cscope!).

## Emitting a printk from userspace

Testing is a critical part of the SDLC; while testing, it's often the case that you will run an automated batch (or suite) of test cases via a script. Now, say that ~~your test (batch) script has initiated a test on your driver hv involving something~~

your test (bash) script has initiated a test on your driver by invoking something like:

```
echo "test data 123<...>" > /dev/mydevnode
```

That's fine, but you'd like to see the point at which the script initiated some action within our kernel module, by printing out a certain distinct (signature) message. As a concrete example, say we want the log to look something like this:

```
my_test_script:----- start testcase 1
my_driver_module:
msg1, ..., msgn, msgn+1, ..., msgn+m
my_test_script:----- end testcase 1
[...]
```

You can have your userspace test script write a message into the kernel log buffer, just like a kernel `printk` would, by writing the given message into the character device file `/dev/kmsg`:

```
sudo bash -c "echo \"my_test_script: start testcase 1\" > /dev/kmsg"
```

(Note how we code it to run with root access).

The message written to the kernel log via the special `/dev/kmsg` device file will be printed at the current default log level, typically, 4: `KERN_WARNING`. We can override this by actually prefixing the message with the required log level (as a number in string format). For example, to write from the userspace into the kernel log at log level 6:`KERN_INFO`, use this:

```
sudo bash -c "echo "<6>my_test_script: start testcase 1" > /dev/kmsg
$ sudo dmesg --decode |tail -n1
user :info : [33561.862960] my_test_script: start testcase 1
```

Notice how I used the `--decode` option to `dmesg` to get more human-readable output. Also, you can see that our latter message is emitted at log level 6 as specified within the echo statement.

There is really no way to distinguish between a user-generated kernel message and a kernel `printk()`-generated one; they look identical. So, of course, it could be as simple as prefixing some special signature byte or string within the message, such as `@myprj@`, in order to help you distinguish these user-generated

prints from the kernel ones.

## Easily dumping buffer content

Once, when working on a network driver, I wrote C code to quite painstakingly dump the content of the Ethernet (link) header, IP header, and so on, in order to analyze and understand exactly how things are working... My code did the typical thing: within a loop, dump each byte of the header structure, printing it in hexadecimal (and, if you're feeling adventurous, when printable, in ASCII as well on the right side). Sure, we can do these things, but don't waste time – the kernel provides!

The macro, `print_hex_dump_bytes()` is there for precisely this kind of work; it's a wrapper over a similar macro. The comments within its code clearly show you the meaning of each of its four parameters, and thus how to use it to efficiently dump memory buffer content:

```
// include/linux/printk.h
/**
 * print_hex_dump_bytes - shorthand form of print_hex_dump() with de-
 * @prefix_str: string to prefix each line with; caller supplies tra-
 * @prefix_type: controls whether prefix of an offset, address, or i-
 * @buf: data blob to dump
 * @len: number of bytes in the @buf
 * Calls print_hex_dump(), with log level of KERN_DEBUG, rowsize of
 */
#define print_hex_dump_bytes(prefix_str, prefix_type, buf, len) \
 print_hex_dump_debug(prefix_str, prefix_type, 16, 1, buf, len, 1)
```

Great; but why does the macro invoke the debug version of the same? Ah, it's tied into the kernel's dynamic debug circuitry! Thus (as we already mentioned in the section *Using the kernel's powerful dynamic debug feature*), every `print_hex_dump_debug()` and `print_hex_dump_bytes()` callsite is able to be dynamically toggled via the dynamic debug control file. Useful!

Here's an example of this macro in action (within the Qualcomm Atheros ath6kl wireless network driver):

```
// drivers/net/wireless/ath/ath6kl/debug.c
void ath6kl_dbg_dump(enum ATH6K_DEBUG_MASK mask,
 const char *msg, const char *prefix,
 const void *buf, size_t len)
{
```

```

{
 if (debug_mask & mask) {
 if (msg)
 ath6kl_dbg(mask, "%s\n", msg);
 print_hex_dump_bytes(prefix,
 DUMP_PREFIX_OFFSET, buf, len);
 }
}
[...]

```

There, all done! Well, no, it's never actually all done, is it...

## Remaining points – bootloader log peeking, LED flashing and more

A common problem when debugging kernel crashes is that once the kernel has crashed or panicked, the system is unusable (typically hung). Reading the kernel log will almost certainly help in debugging the (root) cause... But – I'm sure you see this – how can we see the kernel log if the system is hung!? Moreover, there's no iron-clad guarantee that the log data has been flushed from RAM into non-volatile log files before the system went into a tailspin...

For these reasons, more exotic debug techniques are required at times. One of them is this: after the system hangs (or panics), ‘warm’ boot or reset back into the bootloader prompt (this is assuming that there is a way to do so; let’s assume there is).

### **Warm reset – how?**

A warm reset or reboot is one where the board reboots but RAM content is preserved. I once worked on a prototyping project on a TI PandaBoard; it had a soft reset button; pressing it led to the board performing a warm reboot.

The PC’s *Ctrl + Alt + Delete* (the famous “*three finger salute*”; the temptation to put in a smiley here is high!) is the equivalent... But that’s typically not configured on Linux; you can use the kernel’s *Magic SysRq* facility (again, assuming it’s so configured) to do so (fear not, we shall cover this in a later chapter).

Once at the bootloader prompt, use it’s intelligence to dump the kernel log buffer memory region and you will see the kernel printk’s! (For example, many

embedded systems use the powerful and elegant **Das U-Boot** as their bootloader; the command to dump a memory region is memory display (`md`). Hang on though, a key point: even if you know the kernel log buffer address (that's easy: just do `sudo grep __log_buf /proc/kallsyms` to get it), it's not a physical address, it's a kernel virtual address. You will first have to figure out how to translate it to its physical counterpart – as that's all the bootloader sees. This is typically done by referring to the **Technical Reference Manual (TRM)** for the board or platform you're working on. Once you have the physical address, simply issue the `md` (or equivalent; GRUB has the `dump` command) command to dump the memory content; you will, in effect, see the kernel log!

I refer you to a few actual examples in this (older but excellent) information here:

[https://elinux.org/Debugging\\_by\\_printing#Debugging\\_early\\_boot\\_problems](https://elinux.org/Debugging_by_printing#Debugging_early_boot_problems)).

## Flashing LEDs to debug

Sometimes, especially during the very early stages of board bring-up and likewise, all we really need to know is that some line of code got executed. You can do this by toggling an LED on/off or flashing it as lines of code are hit! Developers at times go to the extent of rigging up the system's GPIO lines (or equivalent) to do so and insert custom code in the kernel to trigger the LED. This is really nothing new – it's the "*poor man's printf*".

(Interestingly, the Raspberry Pi does precisely this when it can't boot – it flashes an on-board LED a given number of times, short and long flashes... here's the documentation that explains how to interpret the LED flashes and thus understand what's causing the boot issue:

<https://www.raspberrypi.com/documentation/computers/configuration.html#led-warning-flash-codes>).

Even better, you could perhaps even rig up a device like the OLED display mentioned earlier to display debug messages; of course, this will require that I2C initialization is performed earlier.

Another thing: you may have heard of the kernel's **netconsole** facility; isn't that something to delve into? Certainly it is – netconsole is a powerful thing, a means to send kernel printk's across a network to a target system which will store it for later perusal. We shall cover it, though in a later chapter; keep a keen eye out!

# Summary

Good going! You've just completed the first of many techniques on debugging the kernel. Instrumentation, though deceptively simple, almost always proves to be a useful and powerful debugging technique.

In this chapter, you began by learning the basics regarding the ubiquitous kernel `printk()`, `pr_*`( ), and `dev_*`( ) routines. We then went into more detail on specific use of these routines to help in debug situations, tips and tricks that will prove useful in debugging your (driver) modules... This included leveraging the kernel's ability to rate-limit printks, often a necessity on high volume code paths.

The kernel's elegant and powerful dynamic debug framework was the highlight of this chapter; here, you learned about it, and how to leverage it: to be able to toggle your (and indeed the kernel's) debug prints even on production systems, with hardly any performance degradation.

We finished this chapter with a few remaining uses of `printk` macros that are sure to prove useful at some point in your kernel/driver journeys.

With these tools in hand, we now move onto another powerful technology in the coming chapter: the kernel's Kprobes framework where we'll of course focus on using it to aid us debug things, primarily via the instrumentation approach. See you there!

## Further reading

- Code browsers tutorials:
  - Ctags Tutorial:  
<https://courses.cs.washington.edu/courses/cse451/10au/tutorials/tutorial>
  - The Vim/Cscope tutorial:  
[http://cscope.sourceforge.net/cscope\\_vim\\_tutorial.html](http://cscope.sourceforge.net/cscope_vim_tutorial.html)
- The `printk` and `/dev/kmsg`:  
<https://www.kernel.org/doc/Documentation/ABI/testing/dev-kmsg>
- *Debugging by printing*: [https://elinux.org/Debugging\\_by\\_printing](https://elinux.org/Debugging_by_printing); covers useful info on debugging with the early `printk` facility, even debugging by dumping the kernel log from the bootloader as well!

- Signing kernel modules; official kernel documentation: *Kernel module signing facility*: <https://www.kernel.org/doc/html/v5.0/admin-guide/module-signing.html#kernel-module-signing-facility>

## RedHat Developer series on GDB

- *The GDB developer's GNU Debugger tutorial, Part 1: Getting started with the debugger*, Seitz, RedHat Developer, Apr 2021: <https://developers.redhat.com/blog/2021/04/30/the-gdb-developers-gnu-debugger-tutorial-part-1-getting-started-with-the-debugger#>
- *The GDB developer's GNU Debugger tutorial, Part 2: All about debuginfo*, Seitz, RedHat Developer, Jan 2022: <https://developers.redhat.com/articles/2022/01/10/gdb-developers-gnu-debugger-tutorial-part-2-all-about-debuginfo>
- *Printf-style debugging using GDB, Part 3*, Buettner, RedHat Developer, Dec 2021: <https://developers.redhat.com/articles/2021/12/09/printf-style-debugging-using-gdb-part-3#>
- Dynamic debug:
  - Official kernel doc: *Dynamic debug*: <https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html#dynamic-debug>
  - *The dynamic debugging interface*, Jon Corbet, LWN, March 2011: <https://lwn.net/Articles/434833/>

# 4 Debug via Instrumentation – Kprobes

A **kernel probe (kprobe)** is one of the powerful weapons in our debug/performance/observability armory! Here, you'll learn what exactly it can do for you and how to leverage it, with the emphasis being on debug scenarios. You will find that there's a so-called static and a dynamic probing approach to using them... We'll also cover using a way to figure the return value of any function via a **kernel return probe (kretprobe)**!

Along the way, you'll learn what the **Application Binary Interface (ABI)** is and why it's important to know at least the basics of the processor ABI.

Don't miss delving into the section on dynamic kprobes or kprobe-based event tracing, as well as employing the `perf-tools` and (especially) the modern eBPF BCC frontends; it makes it all so much easier!

In this chapter, we're going to cover the following main topics:

- Understanding kprobes basics
- Using static kprobes – traditional approaches to probing
- Understanding the basics of the Application Binary Interface (ABI)
- Using static kprobes – demo 3 and demo 4
- Getting started with kretprobes
- Kprobes - limitations and downsides
- The easier way – dynamic kprobes or kprobe-based event tracing
- Trapping into the `execve()` – via perf and eBPF tooling

## Understanding kprobes basics

A kernel probe (**Kprobe** or **kprobe**, or simply, **probe**) is a way to hook or trap into (almost) any function in the kernel proper, or within a kernel module, including interrupt handlers. You can think of kprobes as a dynamic analysis / instrumentation toolset that can even be used on production systems to collect (and later analyze) debugging and/or performance-related telemetry.

To use it, kprobes have to be enabled in the kernel; the kernel configs `CONFIG_KPROBES` must be set to `y` (you'll typically find it under the General architecture-dependent options menu). Selecting it automatically selects `CONFIG_KALLSYMS=y` as well. With kprobes, you can set up three – all optional – types of traps or hooks; to illustrate, let's say you're trapping into the kernel function `do_sys_open()` (which is the kernel function invoked when a userspace process or thread issues the `open(2)` system call):

- **A pre-handler routine:** Invoked just before the call to `do_sys_open()`
- **A post-handler routine:** Invoked just after the call to `do_sys_open()`
- **A fault-handler routine:** Invoked if, during the execution of the pre or post handler a processor fault (exception) is generated (or kprobes is single-stepping instructions; often, a page fault can occur)

They're optional – it's up to you to setup one or more of them. Further, there are two broad types of kprobes you can register (and subsequently unregister):

- **The regular kprobe:** via the `[un]register_kprobe[s]()` kernel APIs
- **A return probe or kretprobe:** via the `[un]register_kretprobe[s]()` kernel APIs, providing access to the probed function's return value.

Let's first work with the regular kprobe and come to the kretprobe a bit later... To trap into a kernel or module function, issue the kernel API:

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *p);
```

The parameter, the `kprobe` structure contains the details; the key members we need to be concerned with are:

- `const char *symbol_name` : The name of the kernel or module function to trap into (internally, the framework employs the `kallsyms_lookup()` API – or a variation of it – to resolve the symbol name into a **kernel virtual address (kva)**, and store it into a member named `addr`); there are a few limitations on which functions you can and cannot trap into (we cover this in the section *Kprobes – limitations and downsides*)
- `kprobe_pre_handler_t pre_handler` : The pre-handler routine function pointer, called just before `addr` is executed
- `kprobe_post_handler_t post_handler` : The post-handler routine function pointer, called just after `addr` is executed

- `kprobe_fault_handler_t fault_handler` : Function pointer to a fault handling routine, which is invoked if executing `addr` causes a fault of any kind; you must return the value `0` to inform the kernel that it must actually handle the fault, `1` if you handled it (uncommon)

Without going into the gory details, it's interesting to realize that you can even set up a probe to a specified *offset* within a function! This is achieved by setting the `offset` member of the kprobe structure to the desired value (watch out though: the offset should be used with care especially on **Complex Instruction Set Computing (CISC)** machines).

Once done, you're expected to release the trap or probe (often on module exit), via the `unregister_kprobe` routine:

```
void unregister_kprobe(struct kprobe *p);
```

Failing to do so will cause a kernel bug(s) and freeze when that kprobe is next hit; it's a resource leak failure of a sort.

## How do kprobes work under the hood?

Unfortunately, this topic lies beyond the scope of this book. Interested readers can certainly refer the excellent kernel documentation which explains the fundamentals of how kprobes actually work: *Concepts: Kprobes and Return Probes*:

<https://www.kernel.org/doc/html/latest/trace/kprobes.html#concepts-kprobes-and-return-probes>

## What we intend to do

This kind of methodology to setup a probe – where, if any change in the function to be probed or in the output format is required – requires a recompile of the module code, is called a **static kprobe**. Is there any other way? Indeed there is: modern Linux kernels have the infrastructure – mostly via the deep Ftrace and tracepoints framework - called **dynamic probing** or kprobe-based event tracing. No C code to write and deal with, no recompile necessary!

In the following sections, we'll show you different ways of setting things up, going from the traditional 'manual' static kprobes interfaces approaches to the more recent and advanced dynamic kernel probes/tracepoints approach. To make

this interesting, here's how we'll go about writing a few demos, most of which will trap into the kernel file open code path:

- Traditional and manual approach, simplest case: attaching a static kprobe, hard-coding it to trap into the open system call; code:  
`ch6/kprobes/1_kprobe`
- Traditional and manual approach: attaching a static kprobe, slightly better, soft-coding it via a module parameter (to the open system call); code:  
`ch6/kprobes/2_kprobe`
- Traditional and manual approach: attaching a static kprobe via a module parameter (to the open system call), plus retrieving the pathname to the file being opened (useful!); code: `ch6/kprobes/3_kprobe`
- Traditional, semi-automated approach: a helper script *generates a template* for both the kernel module C code and the Makefile, enabling attaching a static kprobe to a given function via module parameter; code:  
`ch6/kprobes/4_kprobe_helper`
- A quick look at what a return probe is – the kretprobe – and how to use it (static)
- Modern, easier, dynamic event tracing approach: attaching a dynamic kprobe (as well as kretprobe) to both the `open` and the `execve` system calls, retrieving the pathname to the file being opened / executed
- Modern, easier and powerful, eBPF approach: tracing the file open and the `execve`; just a mention on it.

Great, we'll begin with the traditional static kprobes approach; let's get going!

## Using static kprobes – traditional approaches to probing

In this section, we'll cover writing kernel modules that can probe a kernel or module function in the traditional manner – statically. Any changes will require a recompile of the source.

### Demo 1 – static kprobe – trapping into the file open the traditional static kprobes way – simplest case

Right, let's see how we can trap into (or intercept) the `do_sys_open()` kernel routine by *planting* a kprobe (this code snippet will typically be within the `init`

function of a kernel module; you'll find the code for this demo here:  
ch6/kprobes/1\_kprobe ):

```
// ch6/kprobes/1_kprobe/1_kprobe.c
#include "<...>/convenient.h"
#include "linux/kprobes.h"
[...]
static struct kprobe kpb;
[...]
/* Register the kprobe handler */
kpb.pre_handler = handler_pre;
kpb.post_handler = handler_post;
kpb.fault_handler = handler_fault;
kpb.symbol_name = "do_sys_open";
if (register_kprobe(&kpb)) {
 pr_alert("register_kprobe on do_sys_open() failed!\n");
 return -EINVAL;
}
pr_info("registering kernel probe @ 'do_sys_open()'\n");
```

An interesting use of kprobes is to figure out (approximately) how long a kernel/module function takes to execute. To figure this out (come on, you don't need me to tell you!):

- Take a timestamp in the pre-handler routine (call it `tm_start`); we can use the `ktime_get_real_ns()` routine to do so
- First thing in the post-handler routine, take another timestamp (call it `tm_end`)
- $(tm\_end - tm\_start)$  is the time taken; (do peek at our `convenient.h:SHOW_DELTA()` macro to see how exactly to correctly perform the calculation).

The pre- and post-handler routines follow; let's begin with the pre-handler routine:

```
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
 PRINT_CTX(); // uses pr_debug()
 spin_lock(&lock);
 tm_start = ktime_get_real_ns();
 spin_unlock(&lock);
 return 0;
}
```

Here's our post-handler:

```
static void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long ip)
{
 spin_lock(&lock);
 tm_end = ktime_get_real_ns();
 PRINT_CTX(); // uses pr_debug()
 SHOW_DELTA(tm_end, tm_start);
 spin_unlock(&lock);
}
```

It's pretty straightforward, yes? We grab the timestamps; the `SHOW_DELTA()` macro calculates the difference; where is it? In our *convenience* header (named - surprise surprise! - `convenient.h`). Similarly, the `PRINT_CTX()` macro defined there, gives us a nice one-line summary of the state of the process/interrupt context in the kernel that executed the macro (details on interpreting this follows). The spinlock is used, of course, for concurrency control – as we're operating on shared writable data items.

As the comment next to the `PRINT_CTX()` macro says, it internally uses the `pr_debug()` to emit output to the kernel log; hence, it will only appear if either:

- The symbol `DEBUG` is defined, or
- More usefully, `DEBUG` is deliberately left undefined (as is typical in production) and you make use of the kernel's dynamic debug facility to turn on/off these prints (as discussed in detail in the section *Using the kernel's powerful dynamic debug feature*)

A sample fault handler too is defined; we don't do anything much here, merely emit a `printk` specifying which fault occurred, leaving the actual fault handling – a complex task – to the core kernel (here, we simply copy the fault handler code from the kernel tree: `samples/kprobes/kprobe_example.c`):

```
static int handler_fault(struct kprobe *p, struct pt_regs *regs, int type)
{
 pr_info("fault_handler: p->addr = 0x%p, trap #%dn",
 p->addr, trapnr);
 /* Return 0 because we don't handle the fault. */
 return 0;
}
NOKPROBE_SYMBOL(handler_fault);
```

Notice a couple of things here:

- The third parameter to the fault handler callback, `trapnr`, is the numerical value of the trap that occurred; it's very arch-specific. For example, on x86, 14 implies it's a page fault (similarly, you can always lookup the manual for other processor families to see their values and the meaning)
- The `NOKPROBE_SYMBOL(foo)` macro is used to specify that the function `foo` cannot be probed; here it's specified so that recursive or double faults are prevented.

Now that we've seen the code, let's give it a spin!

### Trying it out

The `test.sh` and `run` bash scripts are simple wrappers (`run` is a wrapper over the wrapper `test.sh`!) to ease testing these demo kernel modules; I'll leave it to you to check out how they work:

```
$ cd <lkd-src-tree>/ch6/kprobes/1_kprobe ; ls
1_kprobe.c Makefile run test.sh
$ cat run
KMOD=1_kprobe
echo "sudo dmesg -C && make && ./test.sh && sleep 5 && sudo rmmod \$.
sudo dmesg -C && make && ./test.sh && sleep 5 && sudo rmmod \${{KMOD}}
$
```

The `run` wrapper script invokes the `test.sh` wrapper script (which performs the `insmod` and sets up the dynamic debug control file to enable our debug `printk`'s). We allow the probe to remain active for 5 seconds; plenty of file open system calls – resulting in the invocation of the `do_sys_open()` and our resulting pre- and post- handlers running – can happen in that time span.

Let's give our first demo a spin on our x86\_64 Ubuntu VM running our custom *production* kernel:

```

$./run
sudo dmesg -C && make && ./test.sh && sleep 5 && sudo rmmod 1_kprobe 2>/dev/null ; sudo dmesg
--- Building : KDIR=/lib/modules/5.10.60-prod01/build ARCH= CROSS_COMPILE= EXTRA_CFLAGS=-DDYNAMIC_DEBUG_MODULE ---
make -C /lib/modules/5.10.60-prod01/build M=/home/letsdebug/Linux-Kernel-Debugging/ch5/kprobes/1_kprobe modules
make[1]: Entering directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
make[1]: Leaving directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
Module 1_kprobe: function to probe: do_sys_open()

-- Module 1_kprobe now inserted, turn on any dynamic debug prints now --
Wrt module 1_kprobe, one or more dynamic debug prints are On
/home/letsdebug/Linux-Kernel-Debugging/ch5/kprobes/1_kprobe/1_kprobe.c:68 [1_kprobe]handler_post =p "\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/kprobes/1_kprobe/1_kprobe.c:65 [1_kprobe]handler_post =p "%03d) %c%c%c:%d | %c%c%
%c%c/* %s() */\012"
/home/letsdebug/Linux-Kernel-Debugging/ch5/kprobes/1_kprobe/1_kprobe.c:49 [1_kprobe]handler_pre =p "%03d) %c%c%c:%d | %c%c%
%c%c/* %s() */\012"
-- All set, look up kernel log with, f.e., journalctl -k -f --

```

*Figure 6.1 – Kprobes demo 1 – invoking the run script*

You can see that the `run` script (invoking the `test.sh` script), sets things up... About five seconds later, here's a snippet of the output seen via `sudo dmesg`:

```

[81970.137707] 1_kprobe:handler_post(): 002) rmmod :8183 | ...1 /* handler_post() */
[81970.138152] 1_kprobe:handler_pre(): 003) systemd-journal :395 | ...1 /* handler_pre() */
[81970.138589] 1_kprobe:handler_post(): delta: 195 ns (~ 0 us ~ 0 ms)
[81970.139587] 1_kprobe:handler_post():
[81970.139588] 1_kprobe:handler_post(): 003) systemd-journal :395 | ...1 /* handler_post() */
[81970.139589] 1_kprobe:handler_post(): delta: 142 ns (~ 0 us ~ 0 ms)
[81970.141131] 1_kprobe:handler_post():
[81970.141752] 1_kprobe:handler_pre(): 003) systemd-journal :395 | ...1 /* handler_pre() */
[81970.142245] 1_kprobe:handler_post(): 003) systemd-journal :395 | ...1 /* handler_post() */
[81970.143010] 1_kprobe:handler_post(): delta: 100 ns (~ 0 us ~ 0 ms)
[81970.143545] 1_kprobe:handler_post():
[81970.175571] 1_kprobe:kprobe_lkm_exit(): bye, unregistering kernel probe @ 'do_sys_open()'
$

```

*Figure 6.2 – Kprobes demo 1 – partial dmesg output*

Great! Our static kprobe, being hit both before and after entering the `do_sys_call()` kernel function, executes the pre and post handlers in our module and produces the prints you're seeing in the preceding screenshot. We need to interpret it.

### Interpreting the PRINT\_CTX() macro's output

In *Figure 6.2*, notice the useful output we obtain from our `PRINT_CTX()` macro (defined within our `convenient.h` header); I reproduce three of the relevant lines below, color-coding them, to help you clearly understand it:

```
[81970.141752] 1_kprobe:handler_pre(): 003) systemd-journal :395 | ...1 /*
handler_pre() */
[81970.142245] 1_kprobe:handler_post(): 003) systemd-journal :395
| ...1 /* handler_post() */
[81970.143010] 1_kprobe:handler_post(): delta: 100 ns (~ 0 us ~ 0 ms)
```

*Figure 6.3 – Kprobes demo 1 – pre- and post- handler sample kernel printk output*

Let's get into what these three lines of output (as seen in the preceding screenshot) are:

- **First line:** Output from the pre-handler routine's `PRINT_CTX()` macro
- **Second line:** Output from the post-handler routine's `PRINT_CTX()` macro
- **Third line:** The delta – the (approximate, usually pretty accurate) time it took for the `do_sys_open()` to run – is seen in the post-handler as well; it's fast, isn't it!
- **Also notice:** The `dmesg` timestamp (the time from boot in seconds.microseconds – don't completely trust it's absolute value though!) and – due to our enabling the debug `printk`'s and the fact that it employs the `pr_fmt()` overriding macro – the module name:function\_name() is also prefixed (for example, here, the latter two lines are prefixed with: `1_kprobe:handler_post()`:

Further, the following screenshot shows you how to fully interpret the output from our useful `PRINT_CTX()` macro:

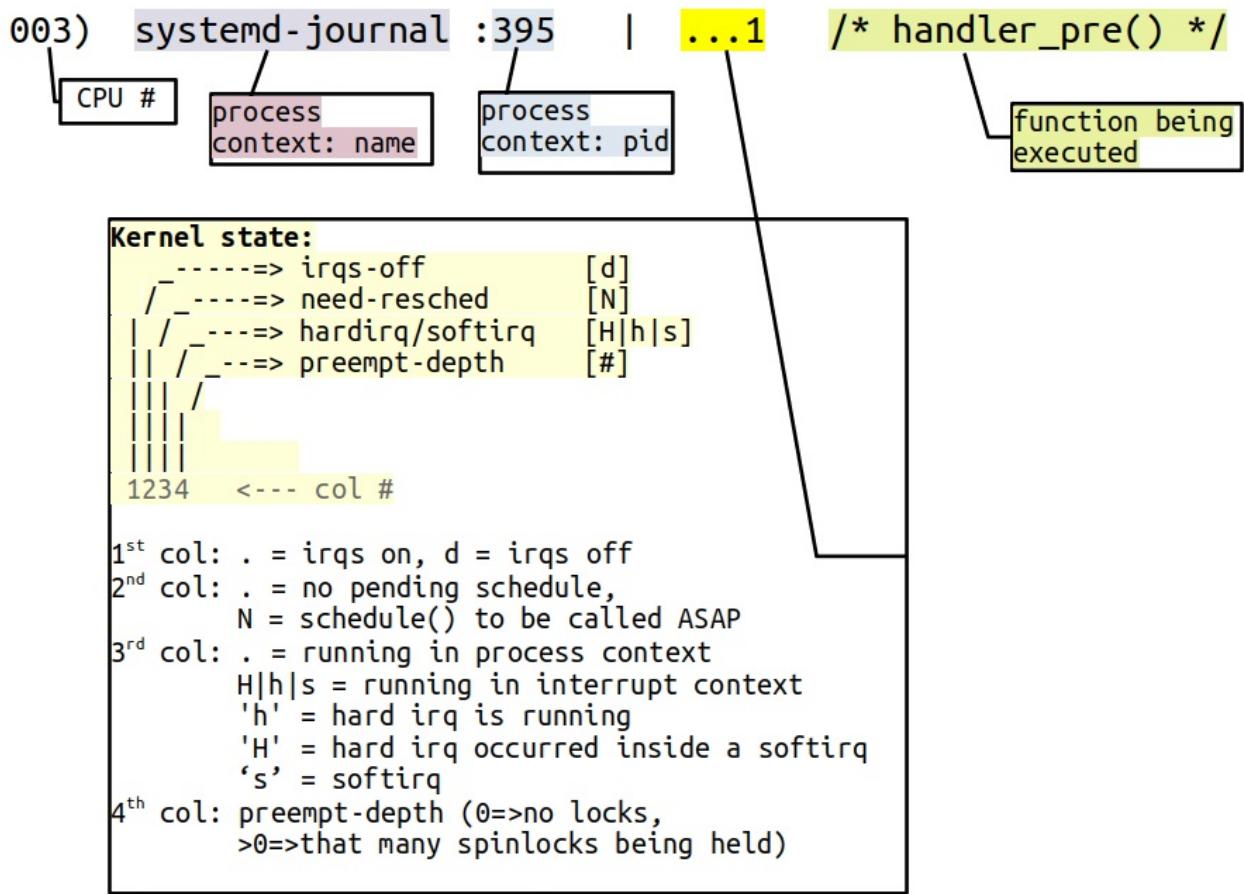


Figure 6.4 – Interpreting the PRINT\_CTX() macro output

Do ensure you carefully study and understand this, it can be very useful in deep debug situations. (In fact, I've mostly mimicked this well-known format from the kernel's Ftrace infrastructure's latency format display; it's explained in the Ftrace documentation here as well:

<https://www.kernel.org/doc/Documentation/trace/ftrace.txt>, under the heading *Output format*:. Worry not, we'll check out *Ftrace* in a later chapter).

Interestingly, ask yourself: what if the PRINT\_CTX() macro runs in an interrupt context? What will be the values for process context name and **Process Identifier (PID)**? The short answer – it will be that of whichever process (or thread) happened to *caught in* (preempted / interrupted by) the interrupt!

Importantly, actually try out this and the following demo modules out yourself! It will go a long way in helping you experiment and learn to sue kprobes effectively.

## The kprobes blacklist - you can't trap this!

A few kernel functions cannot be trapped into via the kprobes interfaces (mainly because they're used internally within the kprobe implementation). You can quickly check which these are – they're available in the (pseudo) blacklist file here: <debugfs-mount>/kprobes/blacklist (the **debug filesystem (debugfs)** is typically mounted under /sys/kernel/debug of course). The kernel documentation discusses this and other kprobe limitations here:

<https://www.kernel.org/doc/html/latest/trace/kprobes.html#kprobes-features-and-limitations>. Do check it out.

You can even do cool (read dangerous) things like modify parameters in the pre-handler! Careful though, it can result in freezes or even outright kernel panic if done incorrectly. (This could be a useful thing – a way to, for example, inject deliberate faults, for testing... while on this, FYI, the kernel has a sophisticated *fault injection* framework).

## Demo 2 – Static kprobe – specifying the function to probe via a module parameter

Our second kprobes demo is quite similar to the first; it differs as follows.

One, we now add and make use of two module parameters – a string one for the name of the function to probe, and an integer determining verbosity:

```
// ch6/kprobes/2_kprobe/2_kprobe.c
#define MAX_FUNCNAME_LEN 64
static char kprobe_func[MAX_FUNCNAME_LEN];
module_param_string(kprobe_func, kprobe_func, sizeof(kprobe_func), MODULE_PARM_DESC(kprobe_func, "function name to attach a kprobe to"));
static int verbose;
module_param(verbose, int, 0644);
MODULE_PARM_DESC(verbose, "Set to 1 to get verbose printk's (default 0)
```

The `kprobe_func` module parameter is useful! It allows us to pass any (valid) function as the probe target, avoiding hard-coding the same. Of course, we now set the `symbol_name` member of the `kprobe` structure to the parameter:

```
kpb.symbol_name = kprobe_func;
```

In addition, the init module code checks that the `kprobe_func` string is non-NULL.

The `verbose` parameter, if set, has the post-handler routine invoke the `PRINT_CTX()` macro. In this demo, we have our `test.sh` wrapper set the module parameters as follows:

```
// ch6/kprobes/2_kprobe/test.sh
FUNC_TO_KPROBE=do_sys_open
VERBOSE=1
[...]
sudo insmod ./${KMOD}.ko kprobe_func=${FUNC_TO_KPROBE} verbose=${VERBOSE}
```

One issue you'll quickly notice with kprobes instrumentation (indeed, it's quite common to many kinds of instrumentation and tracing), is the sheer volume of `printk`'s that can get generated! With a view to limiting it (thus trying to mitigate the overflow of the kernel ring buffer), we introduce a macro named `SKIP_IF_NOT_VI`; if defined, we only log information in the pre and post handlers when the process context is the `vi` process:

```
#ifdef SKIP_IF_NOT_VI
/* For the purpose of this demo, we only log information when the process
 context is the vi process */
if (strncmp(current->comm, "vi", 2))
 return 0;
#endif
```

That's pretty much it; I leave it to you to try it out.

## Exercise

As a small exercise:

- a) Try passing other functions to probe via the module parameter `kprobe_func`
- b) Convert the hard-coded `SKIP_IF_NOT_VI` macro into a module parameter.

Well, great, you now know how to write (simple) kernel modules that leverage the kernel's kprobes framework. To be able to go deeper within a typical debugging context, it becomes necessary to understand deeper things... Things like, how the processor **General Purpose Registers (GPRs)** are typically used,

how the processor interprets a stack frame, passes function parameters and so on... this is the domain of the **Application Binary Interface**, the **ABI**! The following section helps you gain an introduction to it, sufficient information that can prove extremely valuable during a deep debug session. Read on!

## Understanding the basics of the Application Binary Interface (ABI)

In order to gain access to the parameters of a function, you have to first understand at least the basics of how the compiler arranges for the parameters to be passed, at the level of the assembly (assuming the programming language is C, you'll realize that it's really the compiler that generates the required assembly that actually implements the function call, parameter passing, local variable instantiation and return!).

But how does the compiler manage to do this? Compiler authors need to understand how the machine works... Obviously, all of this is very arch (CPU) specific; the precise specification as to how exactly *function calling conventions, return value placement, stack and register usage*, and so on, are done is provided by the microprocessor documentation and it's called the ABI document.

Briefly, the ABI covers the underlying details at the level of the machine concerning:

- CPU register usage
- Function procedure calling and return conventions
- Precise stack frame layout in memory
- The details regarding data representation, linkage, object file formats, and so on.

For example, the x86-32 processors always use the stack to store the parameters to a function before issuing the `CALL` machine instruction; on the other hand, the ARM-32 processors use both CPU **GPRs** as well as the stack (details follow).

Here, we shall only focus on one primary aspect of the ABI – the *function calling conventions (and related register usage)* on a few key architectures:

**Arch (CPU) family**

**How parameters to a**

**Other useful info**

|                  | <b>function are passed</b>                                                                                                                                                                                                                                            | <b>(typical usage)</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IA-32 (x86-32)   | <p>All parameters passed via the stack.</p> <p>Access is via offsets to SP and BP (Base Pointer) registers</p>                                                                                                                                                        | <ul style="list-style-type: none"> <li>• <b>Stack frame layout:</b> <ul style="list-style-type: none"> <li>◦ Parameters &lt;-- <i>higher address</i></li> <li>◦ RET address</li> <li>◦ [SFP] : Optional, if <b>Frame Pointer (FP)</b> enabled</li> <li>◦ Local variables &lt;-- <i>lower address (top of stack)</i></li> </ul> </li> <li>• <b>Return value:</b> Typically in accumulator (EAX)</li> </ul>                                                                                 |
| ARM-32 (Aarch32) | <p>Follows the <b>ARM Procedure Call standard (APCS)</b>. First four parameters to the function are passed in these 32-bit CPU registers: r0 , r1 , r2 , r3 .</p> <p>Any remaining parameters are passed via the stack (stack frame layout is as with the x86-32)</p> | <ul style="list-style-type: none"> <li>• <b>r4 to r9:</b> Scratch registers (often used for local variables)</li> <li>• <b>r7:</b> If system call being issued, holds <b>system call (syscall)</b> #</li> <li>• <b>r11:</b> if enabled, the <b>FP</b></li> <li>• <b>r13: stack pointer (SP)</b></li> <li>• <b>r14: link register (LR);</b> holds the return address, used to return at function exit</li> <li>• <b>r15: program counter (PC)</b> Return value: typically in r0</li> </ul> |

## X86\_64

First six parameters to the function are passed in these 64-bit CPU registers: RDI , RSI , RDX , RCX , R8 , R9 .

Any remaining parameters are passed via the stack (stack frame layout is as with the x86-32, except that 64-bit alignments are used and RBP acts as the base pointer register)

- **RAX:** accumulator; holds syscall # when syscall being issued
- **RBP:** base pointer, start of stack
- **RSP:** stack pointer, current location
- **Return value:** typically in accumulator ( RAX )
- **PSR: Processor Status Register**

## Aarch64 (ARMv8)

APCS: first eight parameters to the function are passed in these 64-bit CPU registers: x0 to x7

Any remaining parameters are passed via the stack

- **X8:** (indirect) Return value address
- **X9 to X15:** Local variables, caller saved
- **X29 (FP):** Frame pointer
- **X30 (LR):** Link register, used to return at function exit
- **X31 (SP):** Stack pointer or a zero register, depending on context
- **Return value:** Typically in x0

Table 6.1 – Summary of function call and register usage ABI information for a few processor families

A few additional points:

- Pretty much all modern processors have a *downward-growing stack* – the stack *grows* from higher virtual addresses to lower virtual addresses. If interested (and I recommend you should be!), do lookup more details in the blog article mentioned just after these points. Things are not always simple: in the presence of compiler optimization, the details seen in *Table 6.1* might not hold (for example, `gcc` and Windows FASTCALL piggyback the first two function parameters into registers `ECX` and `EDX`, on the x86-32). So, do check, **Your Mileage May Vary (YMMV)**...
- The ABI details mentioned here apply to how the C compiler (`gcc / clang`) typically works, thus for the C language, using integer or pointer parameters (not floating point arguments or returns). Also, we don't go into deeper detail here (callee/caller-saved registers, the so-called red zone optimization, exception cases, and so on; refer to the *Further reading* section for links to more).

Links to the ABI documentation for various processor families and its basic details can be found in this (my) blog article: *APPLICATION BINARY INTERFACE (ABI) DOCS AND THEIR MEANING:*

<https://kaiwantech.wordpress.com/2018/05/07/application-binary-interface-abi-docs-and-their-meaning/>.

Now that we have at least basic knowledge of the processor ABI and how the compiler (`gcc / clang`) uses it on Linux, let's put our new found know-how to use; in the following section, we'll learn how to do something pretty useful – determine that pathname of the file being opened via our kprobe-based open system call trap. More generically, we'll learn in effect how to retrieve the parameters to the trapped (probed) function.

## Using static kprobes – demo 3 and demo 4

Continuing to work via the traditional static kprobes approach (recall: the word static implies any change will require a code recompile), let's learn to do more with kprobes – useful and practical stuff that really helps when debugging. Retrieving the parameters to the probed function certainly qualifies as being a very useful skill!

The two demo programs that follow (demos 3 and 4), will show you how to do

precisely this, with demo 4 using an interesting approach – we *generate* our kprobe C code (and `Makefile` file) via a bash script. Let's work on and understand these demos!

## Demo 3 – static kprobe – probing the file open and retrieving the filename parameter

You'll agree, I think, that the second demo is better than the first – it allows the passing of any function to be probed (as a module parameter). Now, continuing with our example of probing the `do_sys_open()`, you've seen (from the first two demos) that we can indeed probe it. In a typical debugging / troubleshooting scenario, though, this isn't nearly enough: being able to **retrieve the parameters to the probed function** can be really important, and prove to be the difference between figuring out the root cause of the issue or not.

### Tip

One often finds that the underlying cause of many bugs is the incorrect passing of parameters (often an invalid pointer). Take care, check and recheck your assumptions!

In line with our demos, this is the signature of the to-be-probed routine, the `do_sys_open()`:

```
long do_sys_open(int dfd, const char __user *filename, int flags, ur
```

Gaining access to its parameters within the pre-handler can be tremendously helpful! The previous section on the basics of the ABI, focused on how to precisely do this; we'll end up demonstrating that we can gain access to and print the pathname to the file being opened – the second parameter `filename`.

### Jumper probes (jprobes)

There's a set of kernel interfaces that enable direct access to any probed function's parameters – it's called a **jumper probe (jprobe)**. However, the jprobe interfaces were deprecated in the 4.15 kernel, the rationale being that one can gain access to a probed (or traced) function's parameters in other, simpler, ways.

We do cover the basics of using the kernel tracing infrastructure to do

various useful things at different points in this book. Here, look out for capturing parameters the *manual* way in the material that just follows, and a much simpler automated way in the section on kernel event tracing within this chapter. (It's worth mentioning that, if your project or product uses a kernel version below 4.15, leveraging the jprobes interfaces can be a useful thing indeed!). Kernel doc on this:

[https://www.kernel.org/doc/html/latest/trace/kprobes.html?  
highlight=kretprobes#deprecated-features](https://www.kernel.org/doc/html/latest/trace/kprobes.html?highlight=kretprobes#deprecated-features).

So, let's leverage our knowledge of the processor ABI, and now, in this our third kprobes demo, gain access to the probed function's second parameter, the file being opened. Interesting stuff, yes? Read on!

## Retrieving the filename

Here's a few snippets from the code of our third demo (obviously, I won't show everything here due to space constraints, please install and look it up from the book's GitHub repo). Let's take the key portion of the code, the kernel module's pre-handler code path:

```
// ch6/kprobes/3_kprobe/3_kprobe.c
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
 char *param_fname_reg;
```

Notice the parameters to the pre-handler:

- First, a pointer to the kprobe structure
- Second, a pointer to a structure named `pt_regs`

Now, this `struct pt_regs` is of interest to us: it encapsulates – obviously in an arch-specific manner – the CPU registers; its definition is thus within an arch-specific header file. Let's consider, as an example, that you're going to run this kernel module on an ARM-32 (Aarch32) based system (for example, a Raspberry Pi 0W). The `pt_regs` structure for ARM-32 is defined here:

`arch/arm/include/asm/ptrace.h` (and/or in  
`arch/arm/include/uapi/asm/ptrace.h`). For ARM (Aarch32), the processor's CPU registers are held in the array member named `uregs`. The `ptrace.h` header has a macro:

```
#define ARM_R1 uregs[1]
```

From the ABI for the ARM-32 (refer *Table 6.1*), we know that the first four parameters (arguments) to a function are passed in CPU GPRs `r0` to `r3`; the second parameter is thus piggy-backed into the register `r1`; hence, our code to gain access to it is:

```
#ifdef CONFIG_ARM
/* ARM-32 ABI:
 * First four parameters to a function are in the foll GPRs: r0, r1,
 * See the kernel's pt_regs structure - rendition of the CPU registers
 * https://elixir.bootlin.com/linux/v5.10.60/source/arch/arm/include/asm
 */
param_fname_reg = (char __user *)regs->ARM_r1;
#endif
```

In a completely analogous fashion, we use conditional compilation based on the CPU architecture, to retrieve the value for the *second parameter* into our local variable `param_fname_reg` as follows:

```
#ifdef CONFIG_X86
param_fname_reg = (char __user *)regs->si;
#endif
[...]
#ifdef CONFIG_ARM64
/* Aarch64 ABI:
 * First eight parameters to a function (and return val) are in the r0-r7
 * See the kernel's pt_regs structure - rendition of the CPU registers
 * https://elixir.bootlin.com/linux/v5.10.60/source/arch/arm64/include
 */
param_fname_reg = (char __user *)regs->regs[1];
#endif
```

Clearly (as *Table 6.1* reveals), on the `x86_64`, the second parameter is held in the `[R]SI` register, and in the register `X1` on the `ARM64` (`Aarch64`); our code retrieves it as per the ABI!

Now it's simply a matter of emitting a `printk` to reveal the file being opened. But hang on... the intricacies of programming in the kernel imply that you cannot simply retrieve the memory at the pointer referred to by our local variable `param_fname_reg`; why not? Careful, it's a pointer to *userspace* memory (and we're running in kernel-space); hence, we employ the `strncpy_from_user()` kernel API to bring it (copy it) into kernel memory space into our already-

allocated kernel buffer `fname` (which we allocate in the module's init code path via `kmalloc()`):

```
if (!strncpy_from_user(fname, param_fname_reg, PATH_MAX + 1))
 return -EFAULT;
pr_info("FILE being opened: reg:0x%px fname:%s\n",
(void *)param_fname_reg, fname);
```

As an interesting aside, only when we test this kernel module on our *debug kernel*, the `strncpy_from_user()` throws a warning `printk`:

```
BUG: sleeping function called from invalid context at lib/strncpy_fi
```

The line of code at this point (`lib/strncpy_from_user.c:117` in the 5.10.60 kernel, as seen) is the function `might_fault()`. A bit simplistically, this function checks: if the kernel config `CONFIG_PROVE_LOCKING` or `CONFIG_DEBUG_ATOMIC_SLEEP` is enabled, it calls the routine `might_sleep()`; the comments for this routine (`include/linux/kernel.h`) clearly tell the story – *it's a debug aid, checking that a sleep does not occur in any kind of atomic context*:

```
/**
 * might_sleep - annotation for functions that can sleep
 * this macro will print a stack trace if it is executed in an atom:
 * context (spinlock, irq-handler, ...). Additional sections where I
 * not allowed can be annotated with non_block_start() and non_bloc
 * pairs.
 * This is a useful debugging help to be able to catch problems ear:
 */
```

I've highlighted the key part of the comment. We find that both `CONFIG_PROVE_LOCKING` and `CONFIG_DEBUG_ATOMIC_SLEEP` are enabled in our debug kernel; that's why this warning is emitted. Well, here and now, we can't do much about it; we simply leave it at that – a warning to be acknowledged, a *Todo* on our list.

So there, it's done; the remainder of the module code is mostly identical to that of our `2_kprobe` module, so we'll skip showing it here. Let's perform a sample run by executing our wrapper `run` script; as before (in the `2_kprobe` demo), to cut down on the volume, we only emit `printk`'s when the process context is `vi`. The final `sudo dmesg` from our wrapper script reveals the kernel log buffer content; the screenshot here (Figure 6.5) showing the trailing portion of the

output:

```
[138698.587054] 3_kprobe:handler_pre(): 003) vi :20612 | ...1 /* handler_pre() */
[138698.588181] 3_kprobe:handler_pre(): FILE being opened: reg:0x000061bfeaedd10 fname:/etc/vim/after/syntax/sh/
[138698.590315] 3_kprobe:handler_post(): delta: 190 ns (~ 0 us ~ 0 ms)
[138698.591400] 3_kprobe:handler_pre(): 003) vi :20612 | ...1 /* handler_pre() */
[138698.592480] 3_kprobe:handler_pre(): FILE being opened: reg:0x000061bfeaedd10 fname:/var/lib/vim/addons/after/syntax/sh/
[138698.594687] 3_kprobe:handler_post(): delta: 190 ns (~ 0 us ~ 0 ms)
[138698.595773] 3_kprobe:handler_pre(): 003) vi :20612 | ...1 /* handler_pre() */
[138698.596914] 3_kprobe:handler_pre(): FILE being opened: reg:0x000061bfeaefbc80 fname:/home/letsdebug/.vim/after/syntax/sh/
[138698.599127] 3_kprobe:handler_post(): delta: 176 ns (~ 0 us ~ 0 ms)
[138700.289318] 3_kprobe:handler_pre(): 003) vi :20612 | ...1 /* handler_pre() */
[138700.292977] 3_kprobe:handler_pre(): FILE being opened: reg:0x000061bfeaed7980 fname:/home/letsdebug/.viminfo
[138700.300213] 3_kprobe:handler_post(): delta: 855 ns (~ 0 us ~ 0 ms)
[138700.303410] 3_kprobe:handler_pre(): 003) vi :20612 | ...1 /* handler_pre() */
[138700.306711] 3_kprobe:handler_pre(): FILE being opened: reg:0x000061bfeaf06640 fname:/home/letsdebug/.viminfo.tmp
[138700.313252] 3_kprobe:handler_post(): delta: 552 ns (~ 0 us ~ 0 ms)
[138700.374248] 3_kprobe:kprobe_lkm_exit(): bye, unregistering kernel probe @ 'do_sys_open'
```

Figure 6.5 – Trailing portion of the dmesg kernel log buffer output from the 3\_kprobe demo on an x86\_64 VM (filtered to show only vi process context)

Look at the preceding screenshot; the pathname of the file being opened – the second parameter to the probed function `do_sys_open()` – is clearly displayed!

## Trying it out on a Raspberry Pi 4 (Aarch64)

For a bit of variety and fun, I also ran this kernel module on a Raspberry Pi 4 running a 64-bit Ubuntu system (thus fully configured to exploit its Aarch64 – arm64 – architecture). We build the module and then `insmod` it:

```
rpi4 # sudo dmesg -C; insmod ./3_kprobe.ko kprobe_func=do_sys_open
[3893.514219] 3_kprobe:kprobe_lkm_init(): FYI, skip_if_not_vi is on
[3893.525200] 3_kprobe:kprobe_lkm_init(): registering kernel probe
```

The printk's clearly show that the (new) module parameter `skip_if_not_vi` is on by default, implying that *only* the `vi` process context – when it opens files – will be captured by our module. Okay, let's do an experiment, let's change it by modifying the parameter on-the-fly, a useful thing. First, though, don't forget to dynamically turn on all our debug prints:

```
rpi4 # echo -n "module 3_kprobe +p" > /sys/kernel/debug/dynamic_debug/control
rpi4 # grep 3_kprobe /sys/kernel/debug/dynamic_debug/control
<...>/3_kprobe.c:98 [3_kprobe]handler_pre =p "%03d) %c%s%c:%d" | ;
<...>/3_kprobe.c:158 [3_kprobe]handler_post =p "%03d) %c%s%c:%d" | ;
rpi4 #
```

Now we query and then modify the module parameter `skip_if_not_vi` to the value `0`:

```
rpi4 # cat /sys/module/3_kprobe/parameters/skip_if_not_vi
1
rpi4 # echo -n 0 > /sys/module/3_kprobe/parameters/skip_if_not_vi
```

Now, *all* file open system calls are trapped via our module; the following screenshot reveals this (you can clearly see both the `dmesg` and the `systemd-journal` process's opening various files):

```
[4410.773412] 3_kprobe:handler_pre(): 001) dmesg :10746 | d..1 /* handler_pre() */
[4410.779891] systemd-journald[890]: /dev/kmsg buffer overrun, some messages lost.
[4410.787758] 3_kprobe:handler_pre(): FILE being opened: reg:0x0000aaaac84c6be8 fname:/etc/terminal-color.s.d
[4410.787762] 3_kprobe:handler_post(): delta: 1888 ns (~ 1 us ~ 0 ms)
[4410.787859] 3_kprobe:handler_pre(): 001) dmesg :10746 | d..1 /* handler_pre() */
[4410.795365] 3_kprobe:handler_pre(): 003) systemd-journal :890 | d..1 /* handler_pre() */
[4410.805236] 3_kprobe:handler_pre(): FILE being opened: reg:0x0000aaaac84c5e60 fname:/dev/kmsg
[4410.811591] 3_kprobe:handler_pre(): FILE being opened: reg:0x0000aaab01cb3b10 fname:/run/log/journal/beef23d9925c4395a56932e79c3b6d4d/system.journal
[4410.819616] 3_kprobe:handler_post(): delta: 2407 ns (~ 2 us ~ 0 ms)
[4410.857187] 3_kprobe:handler_post(): delta: 2018 ns (~ 2 us ~ 0 ms)
[4410.863792] 3_kprobe:handler_pre(): 003) systemd-journal :890 | d..1 /* handler_pre() */
[4410.872539] 3_kprobe:handler_pre(): FILE being opened: reg:0x0000aaab01cb3b10 fname:/run/log/journal/beef23d9925c4395a56932e79c3b6d4d/system.journal
[4410.886218] 3_kprobe:handler_post(): delta: 5260 ns (~ 5 us ~ 0 ms)
[4410.892820] systemd-journald[890]: /dev/kmsg buffer overrun, some messages lost.
[4410.900428] 3_kprobe:handler_pre(): 003) systemd-journal :890 | d..1 /* handler_pre() */
rpi4 #
```

*Figure 6.6 – Partial screenshot showing our 3\_kprobe running on a Raspberry Pi 4 (AArch64), displaying all files being opened*

So, good, it runs flawlessly here as well – thanks to our taking the Aarch64 arch into account in our module code (recall the `#ifdef CONFIG_ARM64 ...` lines within the `3_kprobe.c` module code)!

Voilà! We have the names of all files being opened. Make sure you try this out yourself (at least on your x86\_64 Linux VM).

## Demo 4 – Semi-automated static kprobe via our helper script

This time we make it more interesting! A shell (bash) script (`kp_load.sh`), takes parameters – including the name of the function we'd like to probe and, optionally, the kernel module that contains it (if within a kernel module). It then generates a template for both the kernel module C code and the Makefile, enabling attaching a kprobe to a given function via module parameter.

Due to a scarcity of space, I won't attempt to show the code of the script and the kernel module (`helper_kp.c`) here, just its usage. Of course, I'd expect you to

browse through the code ( ch6/kprobes/4\_kprobe\_helper ) and try it out.

The helper script will first perform a few sanity checks - it first verifies that kprobes is indeed supported on the current kernel. Running it (as root) without parameters has it display its usage or help screen:

```
$ cd ch6/kprobes/4_kprobe_helper
$ sudo ./kp_load.sh
[sudo] password for letsdebug: xxxxxxxxxxxx
[+] Performing basic sanity checks for kprobes support... OK
kp_load.sh: minimally, a function to be kprobe'd has to be specified
Usage: kp_load.sh [-verbose] [--help] [--mod=module-pathname] --probe
 --probe=probe-this-function : if module-pathname
 is not passed, then we assume the function to be
 kprobed is in the kernel itself.
 [--mod=module-pathname] : pathname of kernel
 module that has the function-to-probe
 [-verbose] : run in verbose mode;
 shows PRINT_CTX() o/p, etc
 [--showstack] : display kernel-mode
 stack, see how we got here!
 [--help] : show this help
 screen
$
```

Let's do something interesting – *probe the system's network adapter's hardware interrupt handler*. The steps that follow perform this, using our kp\_load.sh helper script to actually get things done. (**Platform:** Ubuntu 20.04 LTS running our custom production kernel (5.10.60-prod01) on an x86\_64 guest VM.):

1. Identify the network driver on the device (on interface enp0s8 ). The ethtool utility can interrogate a lot of low level details on the network adapter; here we use it to query the driver that's driving the **Network Interface Card (NIC)** or adapter:

```
ethtool -i enp0s8 |grep -w driver
driver: e1000
```

The -i parameter to ethtool specifies the network interface. Further, lsmod verifies the device driver is indeed present in kernel memory:

```
lsmod |grep -w e1000
e1000 135168 0
```

- Find the `e1000` driver's code location within the kernel and identify the hardware interrupt handler function. Most (if not all) Ethernet **Original Equipment Manufacturers (OEM)** NIC's device driver code is within the `drivers/net/ethernet` folder. The `e1000` network driver resides here as well: `drivers/net/ethernet/intel/e1000/`.

Okay, here's the code that sets up the network adapter's hardware interrupt:

```
// drivers/net/ethernet/intel/e1000/e1000_main.c
static int e1000_request_irq(struct e1000_adapter *adapter)
{
 struct net_device *netdev = adapter->netdev;
 irq_handler_t handler = e1000_intr;
 [...]
 err = request_irq(adapter->pdev->irq, handler,
 irq_flags, netdev->name, netdev);
 [...]
```

(FYI, here's the convenient link to the code online:

<https://elixir.bootlin.com/linux/v5.10.60/source/drivers/net/ethernet/intel/e1> (Bootlin's online kernel code browser tooling can be a life saver!).

We can see that the **hardware interrupt (hardirq)** handler routine is named `e1000_intr()`; this is its signature:

```
static irqreturn_t e1000_intr(int irq, void *data);
```

Its code itself is here:

<https://elixir.bootlin.com/linux/v5.10.60/source/drivers/net/ethernet/intel/e1> Cool; let's probe it via our *helper* script!

- Probe it via our helper script:

```
./kp_load.sh --mod=/lib/modules/5.10.60-prod01/kernel/drivers.
```

Do carefully check and note the parameters we've passed to our `kp_load.sh` helper script. It runs... in the following screenshot you can see how our helper script performs its sanity checks, validates the function to probe (and even shows its kernel virtual address via it's `/proc/kallsyms` entry). It then creates a temporary folder (`tmp/`), copies in the C LKM template file (`helper_kp.c`) renaming it appropriately within there, generates the `Makefile` file (using a shell scripting technique called a **HERE document**), switches to the `tmp/`

folder, builds the kernel module and then `insmod`'s it into kernel memory. Whew:

```
$ ls
Readme.txt common.sh* err_common.sh* helper_kw.c kp_load.sh*
$ sudo ./kp_load.sh --mod=/lib/modules/5.10.60-prod01/kernel/drivers/net/ethernet/intel/e1000/e1000.ko --probe=e1000_intr --verbose --showstack
[+] Performing basic sanity checks for kprobes support... OK

FUNCTION=e1000_intr PROBE_KERNEL=0 TARGET_MODULE=/lib/modules/5.10.60-prod01/kernel/drivers/net/ethernet/intel/e1000/e1000.ko ; VERBOSE=1 SHOWSTACK=1
Verbose mode is on

[Validate the to-be-kprobed function e1000_intr]

fffffc00a7b20 t e1000_intr [e1000]
Target kernel Module: /lib/modules/5.10.60-prod01/kernel/drivers/net/ethernet/intel/e1000/e1000.ko

KPMOD=helper_kw-e1000_intr-110ct21
--- Generating tmp/Makefile ---
--- make ---
make -C /lib/modules/5.10.60-prod01/build M=/home/letsdebug/Linux-Kernel-Debugging/ch6/kprobes/4_kprobe_helper/tmp modules
make[1]: Entering directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
--- Dynamic Makefile for helper_kprobes util ---
Building with KERNELRELEASE =
CC [M] /home/letsdebug/Linux-Kernel-Debugging/ch6/kprobes/4_kprobe_helper/tmp/helper_kw-e1000_intr-110ct21.o
/home/letsdebug/Linux-Kernel-Debugging/ch6/kprobes/4_kprobe_helper/tmp/helper_kw-e1000_intr-110ct21.c:61:12: warning: 'running_avg' defined but not used [-Wunused-variable]
 61 | static int running_avg=0;
 | ^~~~~~
--- Dynamic Makefile for helper_kprobes util ---
Building with KERNELRELEASE =
MODPOST /home/letsdebug/Linux-Kernel-Debugging/ch6/kprobes/4_kprobe_helper/tmp/Module.symvers
CC [M] /home/letsdebug/Linux-Kernel-Debugging/ch6/kprobes/4_kprobe_helper/tmp/helper_kw-e1000_intr-110ct21.mod.o
LD [M] /home/letsdebug/Linux-Kernel-Debugging/ch6/kprobes/4_kprobe_helper/tmp/helper_kw-e1000_intr-110ct21.ko
make[1]: Leaving directory '/home/letsdebug/lkd_kernels/productionk/linux-5.10.60'
-rw-r--r-- 1 root root 14640 Oct 11 10:32 helper_kw-e1000_intr-110ct21.ko

kernel module helper_kw-e1000_intr-110ct21 is already inserted... proceeding...
/sbin/insmod ./helper_kw-e1000_intr-110ct21.ko funcname=e1000_intr verbose=1 show_stack=1
$
$ journalctl -k > myklog
$ sudo rmmod helper_kw-e1000_intr-110ct21
$
```

*Figure 6.7 – Screenshot showing the `kp_load.sh` helper script executing and loading up the custom kprobe LKM*

1. I save the kernel log to a file (`journalctl -k > myklog`), remove the LKM from kernel memory and open the log file it in the vi editor; the output is pretty large. Here's a partial screenshot (*Figure 6.8*), capturing our custom kprobe's pre-handler routine's `printk`, the output from our `PRINT_CTX()` macro, and mostly, the output from the `dump_stack()`! The last two lines of output are from the kprobe's post handler routine:

```

24872 Oct 11 06:52:03 dbg-LKD kernel: delta: 44593120 ns (~ 44593 us ~ 44 ms)
24873 Oct 11 06:52:03 dbg-LKD kernel: helper_kw_e1000_intr_110ct21:handler_pre():Pre 'e1000_intr'.
24874 Oct 11 06:52:03 dbg-LKD kernel: 003) [kworker/3:3]:2086 | d.h1 /* handler_pre() */
24875 Oct 11 06:52:03 dbg-LKD kernel: CPU: 3 PID: 2086 Comm: kworker/3:3 Tainted: G OE 5.10.60-prod01 #4
24876 Oct 11 06:52:03 dbg-LKD kernel: Hardware name: innotech GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
24877 Oct 11 06:52:03 dbg-LKD kernel: Workqueue: events e1000_watchdog [e1000]
24878 Oct 11 06:52:03 dbg-LKD kernel: Call Trace:
24879 Oct 11 06:52:03 dbg-LKD kernel: <IRQ>
24880 Oct 11 06:52:03 dbg-LKD kernel: dump_stack+0x76/0x94
24881 Oct 11 06:52:03 dbg-LKD kernel: ? e1000_intr+0x1/0x110 [e1000]
24882 Oct 11 06:52:03 dbg-LKD kernel: handler_pre.cold+0x5/0xc4a [helper_kw_e1000_intr_110ct21]
24883 Oct 11 06:52:03 dbg-LKD kernel: kprobe_ftrace_handler+0xf2/0x160
24884 Oct 11 06:52:03 dbg-LKD kernel: ? __handle_irq_event_percpu+0x45/0x1c0
24885 Oct 11 06:52:03 dbg-LKD kernel: ftrace_ops_assist_func+0x98/0x140
24886 Oct 11 06:52:03 dbg-LKD kernel: 0xfffffffffc0500e0e3
24887 Oct 11 06:52:03 dbg-LKD kernel: RIP: 0010:e1000_intr+0x1/0x110 [e1000]
24888 Oct 11 06:52:03 dbg-LKD kernel: Code: c2 77 bf 48 8b 87 80 03 00 00 f0 80 a0 90 00 00 00 fe 45 31 c0 83 87 48 0b 00 00 01 e
b a4 66 66 2e 0f 1f 84 00 00 00 e8 <d8> 64 46 00 48 8b 80 0d 00 00 8b 80 c0 00 00 85 c0 0f 84 b2
24889 Oct 11 06:52:03 dbg-LKD kernel: RSP: 0018:fffffb63b80148f28 EFLAGS: 00000046 ORIG_RAX: 0000000000000000
24890 Oct 11 06:52:03 dbg-LKD kernel: RAX: ffffffff00a7b20 RBX: fffff9b25e526e000 RCX: 0000000000000000
24891 Oct 11 06:52:03 dbg-LKD kernel: RDX: 00000000000000010001 RSI: fffff9b25c5092000 RDI: 000000000000000010
24892 Oct 11 06:52:03 dbg-LKD kernel: RBP: ffffffb63b80148f60 R08: fffff9b25f78da400 R09: 0000000000000000
24893 Oct 11 06:52:03 dbg-LKD kernel: R10: 0000000000000000 R11: 0000000000000000 R12: 0000000000000000
24894 Oct 11 06:52:03 dbg-LKD kernel: R13: ffffffb63b80148f74 R14: 0000000000000010 R15: 0000000000000000
24895 Oct 11 06:52:03 dbg-LKD kernel: ? e1000_maybe_stop_tx+0x90/0x90 [e1000]
24896 Oct 11 06:52:03 dbg-LKD kernel: ? e1000_intr+0x5/0x110 [e1000]
24897 Oct 11 06:52:03 dbg-LKD kernel: ? __handle_irq_event_percpu+0x45/0x1c0
24898 Oct 11 06:52:03 dbg-LKD kernel: ? e1000_intr+0x5/0x110 [e1000]
24899 Oct 11 06:52:03 dbg-LKD kernel: ? __handle_irq_event_percpu+0x45/0x1c0
24900 Oct 11 06:52:03 dbg-LKD kernel: handle_irq_event_percpu+0x33/0x90
24901 Oct 11 06:52:03 dbg-LKD kernel: handle_irq_event+0x39/0x60
24902 Oct 11 06:52:03 dbg-LKD kernel: handle_fasteo1_irq+0xc5/0x1a0
24903 Oct 11 06:52:03 dbg-LKD kernel: ? handle_nested_irq+0x110/0x110
24904 Oct 11 06:52:03 dbg-LKD kernel: asm_call_irq_on_stack+0x12/0x20
24905 Oct 11 06:52:03 dbg-LKD kernel: </IRO>
24906 Oct 11 06:52:03 dbg-LKD kernel: common_interrupt+0x136/0x1d0
24907 Oct 11 06:52:03 dbg-LKD kernel: asm_common_interrupt+0x1e/0x40
24908 Oct 11 06:52:03 dbg-LKD kernel: RIP: 0010:e1000_watchdog+0x19d/0x590 [e1000]
24909 Oct 11 06:52:03 dbg-LKD kernel: Code: 39 f0 0f 82 fa 01 00 00 41 83 bc 24 f8 fb ff ff 04 0f 87 51 01 00 00 49 8b 94 24 e0 f
b ff ff b8 10 00 00 00 89 82 c8 00 00 <d8> c6 84 24 f1 f9 ff 01 49 8b 44 24 c8 a8 04 0f 84 e6 01 00 00
24910 Oct 11 06:52:03 dbg-LKD kernel: RSP: 0018:fffffb63b81b67e20 EFLAGS: 00000297
24911 Oct 11 06:52:03 dbg-LKD kernel: RAX: 0000000000000010 RBX: fffff9b25c5092000 RCX: 0000000000000010
24912 Oct 11 06:52:03 dbg-LKD kernel: RDX: fffffb63b82160000 RSI: 00000000000000100 RDI: fffff9b25c5092d80
24913 Oct 11 06:52:03 dbg-LKD kernel: RBP: ffffffb63b81b67e58 R08: fffff9b25c50931a8 R09: fffff9b263ddab9e0
24914 Oct 11 06:52:03 dbg-LKD kernel: R10: fffff9b25e45c366c R11: 00000000000000018 R12: fffff9b25c50931a0
24915 Oct 11 06:52:03 dbg-LKD kernel: R13: ffffffb662ad00 R14: fffff9b25c5092d80 R15: fffff9b25c5092900
24916 Oct 11 06:52:03 dbg-LKD kernel: process_one_work+0x1b8/0x3b0
24917 Oct 11 06:52:03 dbg-LKD kernel: worker_thread+0x50/0x3a0
24918 Oct 11 06:52:03 dbg-LKD kernel: ? process_one_work+0x3b0/0x3b0
24919 Oct 11 06:52:03 dbg-LKD kernel: kthread+0x154/0x180
24920 Oct 11 06:52:03 dbg-LKD kernel: ? kthread_unpark+0x80/0x80
24921 Oct 11 06:52:03 dbg-LKD kernel: ret_from_fork+0x22/0x30
24922 Oct 11 06:52:03 dbg-LKD kernel: helper_kw_e1000_intr_110ct21:handler_post():kworker/3:3:2086. Post 'e1000_intr'.
24923 Oct 11 06:52:03 dbg-LKD kernel: delta: 25862601 ns (~ 25862 us ~ 25 ms)

```

24922,1      20%

*Figure 6.8 – (Partial) screenshot showing output from the kernel log as emitted by our ‘helper’ script’s custom kprobe within the pre-handler routine; last two lines from the post handler*

Interesting! Our custom auto-generated kprobe has achieved this!

Don’t fret regarding how exactly to interpret the kernel-mode stack right now; we shall cover all this in detail in coming chapters. For now, I’ll point out the following key things with regard to *Figure 6.8* (ignoring the line number and left five columns):

- Line 24873 : output from our custom generated kprobe: as the verbose flag is set, a debug printk showing the call site –  
`helper_kw_e1000_intr_110ct21:handler_pre():Pre 'e1000_intr'.`

- Line 24874 : output from our `PRINT_CTX()` macro –  
`003) [kworker/3:3]:2086 | d.h1 /* handler_pre() */.` The four-character `d.h1` sequence is interpreted as per our *Figure 6.8*: hardware interrupts are disabled (off), we’re currently running in hardirq context (of course we are, the probe is on the NIC’s interrupt handler) and a (spin)lock is currently held.
- Line 24875 to line 24921 : output from the `dump_stack()` ; useful information indeed! For now, just read it bottom-up ignoring all lines that begin with a `?`. Well, one key point: in this particular case, do you notice that there are actually *two* kernel-mode stacks on display here?
  - One: the upper portion, is within the `<IRQ>` and `</IRQ>` tokens; this tells us it’s the IRQ stack – a special stack region used to hold stack frames when a hardware interrupt is being processed (this is an arch-specific feature known as **interrupt** (or **IRQ**) stacks; most modern processors use it)
  - The lower portion of the stack, after the `</IRQ>` is the regular kernel-mode stack; it’s in fact the (kernel) stack of the process context that happened to get rudely interrupted by the hardware interrupt (here, it happens to be a kernel thread named `kworker/3:3`)

## Interpreting kthread names

By the way, how does one interpret kernel thread names (like the `kworker/3:3` seen here)? They’re essentially cast in this format:  
`kworker/%u:%d%s` (`cpu, id, priority`) Refer this link for more details:  
<https://www.kernel.org/doc/Documentation/kernel-per-CPU-kthreads.txt>

Nice; using the helper script does make things easier. There’s a price to pay of course, there’s always a trade-off (as with life): our `helper_kp.c` LKM’s C code template remains hard-coded for any and every probe we set up using it.

Now you know how to code static kprobes; more so, how you can leverage this technology to help you carefully instrument – and thus debug – kernel / module code, even on production systems! The other side of the coin is the kretprobe; let’s jump into learning how to use it.

## Getting started with kretprobes

At the outset of this chapter, you learned how to use the basic kprobes APIs to

setup a static kprobe (or two). Let's now cover an interesting counterpart to the kprobe – the **kretprobe**, *allowing us to gain access to any (well, most) kernel or module function's return value!* This – being able to dynamically look up a given function's return value - can be a game changer in a debug scenario.

## Pro tip

**Don't assume:** If a function returns a value, always check for the failure case. One day it could fail – yes, even the `malloc()` or the `kmalloc()`! Fail to catch the possible failure and you'll be flailing to figure out what happened!

The relevant kretprobe APIs are straight-forward:

```
#include <linux/kprobes.h>
int register_kretprobe(struct kretprobe *rp);
void unregister_kretprobe(struct kretprobe *rp);
```

The `register_kretprobe()` returns `0` on success and, in the usual kernel style (the `0/-E` convention), a negative `errno` value on failure.

## Tip – errno values and their meaning

As you'll know, `errno` is an integer found in every process's uninitialized data segment (more recently, it's constructed to be *thread-safe* by employing the powerful **Thread Local Storage (TLS)** Pthreads feature, implemented via the compiler and the usage of the `__thread` keyword in the variable declaration). When a system call fails (typically returning `-1`), the programmer can query the error diagnostic by looking up `errno`; the kernel (or underlying driver) will return the appropriate negative `errno` integer; glibc glue code will set it by multiplying it by `-1` (thus making it positive). It serves as an index into a 2d array of English error messages which can be conveniently looked up via the `[p]error(3)` or `strerror(3)` glibc APIs.

I often find it useful to be able to quickly lookup a given `errno` value; use the userspace headers `/usr/include/asm-generic/errno-base.h` (covers `errno` values 1 to 34) and `/usr/include/asm-generic/errno.h` (covers `errno` values 35 to 133), as of this writing.

For example, if you see the (kernel function) return value -101 in a log file: `#define ENETUNREACH 101 /* Network is unreachable */`

The kretprobe structure internally contains the kprobe structure, allowing one to setup the probe point (the function to *return* probe) via it; in effect, the probe point will be `rp->kp.addr` (where `rp` is the pointer to the kretprobe structure, with the address being typically figured out via `rp->kp.symbol_name` – set to the name of the function to be probed). The `rp->handler` is the kretprobe handler function; its signature is:

```
int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs
```

Just as with kprobes, you will receive all CPU registers within the handler function via the second parameter, the `pt_regs` structure. The first parameter, the `kretprobe_instance` structure, holds (among other housekeeping fields), the following:

- `ri->ret_addr` : The return address
- `ri->task` : The pointer to the process context's task structure (which encapsulates all attributes of the task)
- `ri->data` : A means to gain access to a per-instance private data item

But what about the main feature, the return value from the probed function? Ah, recall our discussion on the processor ABI (section *Understanding the basics of the Application Binary Interface (ABI)*); the return value is again placed into a processor register, the particular register being of course very arch (CPU) specific. *Table 6.1* shows you the relevant details. But hang on, you don't have to manually look it up – there's an elegant, simpler way: a macro:

```
regs_return_value(regs);
```

This macro is a hardware-agnostic abstraction, separately defined for each processor family, that provides the return value from the appropriate register (the registers being passed via `struct pt_regs *regs` of course)! For example, the essential implementation of `regs_return_value()` on the:

- ARM (Aarch32) is: `return regs->ARM_r0;`
- A64 (Aarch64) is: `regs->regs[0]`
- x86 is: `return regs->ax;`

It just works

The kernel community has been providing sample source code for some select kernel features; this includes the kprobe and kretprobe. Here's some relevant snippets from the sample code for the kretprobe (found here within the kernel code base: `samples/kprobes/ kretprobe_example.c`) via the following bullet points:

- The module parameter `func` enables us to pass any function to probe (ultimately, for its return value):

```
static char func_name[NAME_MAX] = "kernel_clone";
module_param_string(func, func_name, NAME_MAX, S_IRUGO);
MODULE_PARM_DESC(func, "Function to kretprobe; this module will
```

- The kretprobe structure definition:

```
static struct kretprobe my_kretprobe = {
 .handler = ret_handler,
 .entry_handler = entry_handler,
 .data_size = sizeof(struct my_data),
 /* Probe up to 20 instances concurrently. */
 .maxactive = 20,
};
```

Let's now delve into this `kretprobe` structure:

- `handler` : member that specifies the function to run when the function we're probing completes, enabling us to fetch the return value; it's the *ret handler*
- `entry_handler` : member that specifies the function to run when the function we're probing is entered; it gives you a chance to determine whether the return will be collected:
  - if you return `0` (implying success), the return function – `handler` – will be called upon return of the probed function
  - if you return non-zero, the k[ret]probe does not even happen; in effect it gets disabled for this particular function instance (the kernel doc here gives you the deeper details on the entry handler and private data fields:  
[https://www.kernel.org/doc/html/latest/trace/kprobes.html?  
highlight=kretprobes#kretprobe-entry-handler](https://www.kernel.org/doc/html/latest/trace/kprobes.html?highlight=kretprobes#kretprobe-entry-handler))
- `maxactive` : used to specify how many instances of the probed

function can be simultaneously probed; the default is the number of CPU cores on the box ( `NR_CPUS` ); if the `nmissed` field of the kretprobe structure is positive, it implies you missed that many instances (you can then increase `maxactive` ). Again, the kernel doc here gives you the deeper details:

[https://www.kernel.org/doc/html/latest/trace/kprobes.html?  
highlight=kretprobes#how-does-a-return-probe-work](https://www.kernel.org/doc/html/latest/trace/kprobes.html?highlight=kretprobes#how-does-a-return-probe-work)

- In the module initialization code path, plant the return probe:

```
my_kretprobe.kp.symbol_name = func_name;
ret = register_kretprobe(&my_kretprobe);
```

- The actual return handler code (skipping details):

```
static int ret_handler(struct kretprobe_instance *ri, struct pt
{
 unsigned long retval = regs_return_value(regs);
 struct my_data *data = (struct my_data *)ri->data;
 [...]
 delta = ktime_to_ns(ktime_sub(now, data->entry_stamp));
 pr_info("%s returned %lu and took %lld ns to execute\n", fu
 return 0;
}
```

I've highlighted (in bold) the key lines – where the return address is obtained and printed

- In the module cleanup code path, the kretprobe is unregistered (and the missed instances count displayed):

```
unregister_kretprobe(&my_kretprobe);
pr_info("kretprobe at %p unregistered\n", my_kretprobe.kp.addr)
/* nmissed > 0 suggests that maxactive was set too low. */
pr_info("Missed probing %d instances of %s\n", my_kretprobe.nmi
```

Do try it out...

## Kprobes miscellany

A couple of remaining things to mention while on this topic of k[ret]probes:

- One, you can even set up multiple kprobes or kretprobes with a single API call, as follows:

```
#include <linux/kprobes.h>
int register_kprobes(struct kprobe **kps, int num);
int register_kretprobes(struct kretprobe **rps, int num);
```

As you might expect, these are convenience wrappers calling the underlying registration routine in a loop. The `unregister_k[ret]probes()` routine counterparts are used to unregister the probes; we don't delve further into these here.

- Two, a kprobe or kretprobe, can be temporarily disabled; this can be done via

```
int disable_kprobe(struct kprobe *kp);
int disable_kretprobe(struct kretprobe *rp);
```

And later re-enabled via the corresponding and analogous `enable_k[ret]probe()` APIs. This can be useful: a way to throttle the amount of debug telemetry being logged.

### **Inner workings**

If you'd like to delve into the inner working, into how kprobes and kretprobes are internally implemented, the official kernel documentation covers it here: <https://www.kernel.org/doc/Documentation/kprobes.txt>. (Check out the sections *Concepts: Kprobes, and Return Probes*).

Now that you know how to use both kprobes and kretprobes, it's also important to understand that they have some inherent limitations, even downsides. The following section covers just this.

## Kprobes – limitations and downsides

We do realize that no single feature can do anything and everything – in the words of Frederick J Brooks (in his incomparable book *The Mythical Man Month*): “*there is no silver bullet*”.

As we've seen, certain kernel / module functions cannot be probed; these include:

- functions marked with the `__kprobes` or `nokprobe_inline` annotation
- Functions marked via the `NOKPROBE_SYMBOL()` macro
- The pseudo-file `/sys/kernel/debug/kprobes/blacklist` holds the names of functions that can't be kprobe'd. As well, some inline functions might not be able to be probed.

There's more to note on the point of using k[ret]probes on production systems due to the possibility of stability issues; the next section throws some light on this.

## Interface stability

We know that kernel APIs can change at any point (this is in any case a given with kernel development and maintenance); so one can imagine a situation where your kernel module sets up kprobes for some functions, say, `x()` and `y()`. In a later kernel release though, there's no telling what happens – these functions might be deprecated, or their signatures (and thus parameters and return type) might change, leading to your k[ret]probe kernel module requiring constant maintenance. (Well, honestly, this is pretty much a given).

A bit more on this, the last point. An important word of caution: it can be dangerous – from a stability and security viewpoint – to include third party kernel modules on production systems, especially (and obviously), on mission-critical ones. Their presence can also void the warranty given by the OS vendors (like Red Hat, **SUSE Linux Enterprise Server (SLES)**, Canonical, and so on). (Dev) Ops people are in general extremely wary of letting in untested code onto production systems, let alone kernel modules; they won't exactly be thrilled when you insert them.

Also, kprobes can cause kernel instability when attached to high volume code paths (like scheduling, interrupt / timer or networking code); avoid them, if possible. If not, at least mitigate the risk by reducing `printk` usage and using `printk` rate limiting APIs (we covered rate limiting the `printk` in the previous chapter).

How will I know if a certain kernel (or module) function runs very often? The **funccount** utility – via either the `perf-tools[-unstable]` package or the more recent eBPF tools packages – can profile and show you high volume code paths

within the kernel. (The utility script is typically named `funccount-perf` or `funccount-bpfcc`, depending on what you have installed).

A modern, cleaner and far more efficient approach to the *static* kprobe (or kretprobe) is to employ tracing mechanisms that are already built into the kernel fabric, and are thus tested and production-ready. These include using dynamic kprobes or kprobe-based event tracing (frontends like `kprobe-perf` take advantage of these), kernel trace points (provided via Ftrace), perf and eBPF frontends, and so on. They're also simply a lot easier to use; they don't require C coding and deep knowledge of kernel internals, they're DevOps / sysad friendly as well! Let's get started exploring them!

## The easier way – dynamic kprobes or kprobe-based event tracing

Similar, but much superior, to how I built a small script in our demo 4 to make it easier for us to hook into any kernel function via kprobes, there is a package called `perf-tools` (or `perf-tools-unstable`); the creator and lead author is Brendan Gregg. Within the useful tools you'll find in this package, a bash script named `kprobe` (or `kprobe-perf`) is a fantastic wrapper, easily letting us setup kprobes (and kretprobes)!

Assuming you've installed the package (we specified it back in *Chapter 1, A General Introduction to Debugging Software*), let's go ahead and run the script (by the way, on my x86\_64 Ubuntu 20.04 LTS system, the package name is `perf-tools-unstable` and the script is called `kprobe-perf`):

```
dpkg -l|grep perf-tools
ii perf-tools-unstable 1.0.1~20200130+git491
file $(which kprobe-perf)
/usr/sbin/kprobe-perf: Bourne-Again shell script, ASCII text execut:
#
```

Great; let's run it and see it's help screen (you will need to eventually run it as root, so I just do so here):

```

kprobe-perf
USAGE: kprobe [-FhHsv] [-d secs] [-p PID] [-L TID] kprobe_definition [filter]
 -F # force. trace despite warnings.
 -d seconds # trace duration, and use buffers
 -p PID # PID to match on events
 -L TID # thread id to match on events
 -v # view format file (don't trace)
 -H # include column headers
 -s # show kernel stack traces
 -h # this usage message

```

Note that these examples may need modification to match your kernel version's function names and platform's register usage.

eg,

```

kprobe p:do_sys_open
 # trace open() entry
kprobe r:do_sys_open
 # trace open() return
kprobe 'r:do_sys_open $retval'
 # trace open() return value
kprobe 'r:myopen do_sys_open $retval'
 # use a custom probe name
kprobe 'p:myopen do_sys_open mode=%cx:u16'
 # trace open() file mode
kprobe 'p:myopen do_sys_open filename=+0(%si):string'
 # trace open() with filename
kprobe -s 'p:myprobe tcp_retransmit_skb'
 # show kernel stacks
kprobe 'p:do_sys_open file=+0(%si):string' 'file ~ "*stat"'
 # opened files ending in "stat"

```

See the man page and example file for more info.

#

*Figure 6.9 – Screenshot showing the help screen of the kprobe-perf script*

The help screen does a great job of summing up how you can use this useful utility; do refer to its man page (as well as the online page here:

[https://github.com/brendangregg/perf-tools/blob/master/examples/kprobe\\_example.txt](https://github.com/brendangregg/perf-tools/blob/master/examples/kprobe_example.txt)).

I recommend you first try out a few examples similar to those shown (in *Figure 6.9*).

Next, and without further ado, let's leverage this powerful script to very easily do what we so painstakingly progressed to in the earlier four demos: setup a

kprobe on the `do_sys_open()` and print the pathname of the file being opened (as our earlier example in *Figure 6.5* shows!):

```
kprobe-perf 'p:do_sys_open file=+0(%si):string'
Tracing kprobe do_sys_open. Ctrl-C to end.
 kprobe-perf-8171 [002] ...1 9159.540104: do_sys_open: (do_
 kprobe-perf-8171 [002] ...1 9159.540259: do_sys_open: (do_
 kprobe-perf-8171 [002] ...1 9159.542030: do_sys_open: (do_
 kprobe-perf-8171 [002] ...1 9159.542818: do_sys_open: (do_
 irqbalance-676 [000] ...1 9162.010699: do_sys_open: (do_
 irqbalance-676 [000] ...1 9162.011642: do_sys_open: (do_
[...]^C
```

Notice the syntax:

- The ‘`p:do_sys_open`’ sets up a kprobe on the `do_sys_open()` kernel function
- on the x86\_64, the ABI tells us that the `[R]SI` register holds the second parameter to the function (recall *Table 6.1*) – in this case, it’s the pathname of the file being opened; the script employs the syntax `+0(%si):string` to display its content as a string (prefixed with the `file=`)

As easy as that! To minimally test, I ran `ps` in another terminal window; immediately the `kprobe-perf` script dumped lines like this:

```
ps-8172 [000] ...1 9164.231685: do_sys_open: (do_sys_open+0x0/0)
ps-8172 [000] ...1 9164.232582: do_sys_open: (do_sys_open+0x0/0)
ps-8172 [000] ...1 9164.233758: do_sys_open: (do_sys_open+0x0/0)
ps-8172 [000] ...1 9164.234776: do_sys_open: (do_sys_open+0x0/0)
[...]
ps-8172 [000] ...1 9164.248680: do_sys_open: (do_sys_open+0x0/0)
ps-8172 [000] ...1 9164.249511: do_sys_open: (do_sys_open+0x0/0)
ps-8172 [000] ...1 9164.260290: do_sys_open: (do_sys_open+0x0/0)
ps-8172 [000] ...1 9164.260854: do_sys_open: (do_sys_open+0x0/0)
[...]
```

...and plenty more... You can literally gain insight into how `ps` works by doing this! (In fact, the wonderful `strace` utility – *it traces all system calls issued by a process* – can approach this level of detail as well! Don’t ignore it).

The point here of course is to simply show you how much easier it is to get the same valuable information – internally leveraging the kernel’s kprobes framework and knowledge of the processor ABI - using this tool.

Further, the output format that the `kprobe-perf` script uses is as follows:

```
-----=> irqs-off
/ -----=> need-resched
| / -----=> hardirq/softirq
|| / -----=> preempt-depth
||| / delay
TASK-PID CPU# |||| TIMESTAMP FUNCTION
| | | |||| | | |
ps-8172 [000] ...1 9164.260854: do_sys_open: (do_sy:
```

It's familiar, with good reason: it's again that of Ftrace, and very similar to what we did with our `PRINT_CTX()` macro (recall *Table 6.4*).

As you probably guessed, the `kprobe-perf` script, to get the job done, somehow sets up a kprobe; this is indeed easily verified by looking up the `kprobes/list` pseudo-file under your debugfs mount point. While the above command was running, I ran this in another terminal window:

```
cat /sys/kernel/debug/kprobes/list
ffffffff965d1a60 k do_sys_open+0x0 [FTRACE]
```

Clearly, a kprobe was set up on the `do_sys_open()` kernel function.

## Kprobe-based event tracing – minimal internal details

So, how does the `kprobe-perf` script setup a kprobe? Ah, here's the really interesting thing: it does so by leveraging the kernel's *Ftrace* infrastructure which internally tracks key events within the kernel; this is known as the kernel's **event tracing** framework, and within it, the **kprobes events** framework. It can be considered to be a subset of the larger Ftrace kernel system; that's why you see the `[FTRACE]` to the right of the `kprobes/list` line!

The kprobe events code was introduced into the kernel back in 2009 by Masami Hiramatsu. Essentially, via it, the kernel can toggle the tracing of select (with a few limitations) kernel functions.

Internally speaking, here's the bare minimal information on how a kprobe is set up: within the `debug/tracing/` folder (typically here:

`/sys/kernel/debug/tracing/`) there will exist a directory named `events` (this is assuming the kernel config `CONFIG_KPROBE_EVENTS=y`; it typically is, even on

distro and many production kernels). Under it, there are folders representing various subsystems and/or well known event classes that the kernel's event tracing infrastructure tracks.

## Using the event tracing framework to trace built-in functions

The kernel's **event tracing** infrastructure also mirrors these tracepoints at this location: `/sys/kernel/tracing`. This can be particularly useful when, on a production system, debugfs is kept invisible (as a security measure).

Let's peek into it:



A screenshot of a terminal window titled "root@dbg-LKD: /home/letsdebug". The terminal shows the output of several commands related to kernel event tracing:

```
pwd
/sys/kernel/tracing
ls events/
alarmtimer/ enable huge_memory/ irq_vectors/ neigh/ random/ signal/ timer/
avc/ exceptions/ hwmon/ jbd2/ net/ ras/ skb/ tlb/
block/ ext4/ i2c/ kmem/ libata/ raw_syscalls/ smbus/ udp/
bpf_test_run/ fib/ initcall/ mce/ mmi/ rCU/ sock/ vmscan/
bpf_trace/ fib6/ intel_iommu/ interconnect/ libata/ page_isolation/ regulator/ spi/ vsyscall/
cgroup/ filelock/ iomap/ mddio/ migrate/ page_pool/ resctrl/ swiotlb/
clk/ filemap/ iommu/ mmc/ power/ printk/ rpm/ sync_trace/
compaction/ fs_dax/ iomap/ io_uring/ module/ page_pool/ rseq/ task/
cpuhp/ ftrace/ iommu/ mmc/ power/ printk/ rtc/ tcp/
devfreq/ gpio/ io_uring/ module/ msr/ pwm/ sched/ thermal/
dma_fence/ header_event irq/ msr/ napi/ qdisc/ sscsi/ thermal_power_allocator/
drm/ header_page irq_matrix/ napi/ qdisc/ sscsi/ thermal_power_allocator/ xhci-hcd/
#
ls events/kmem/
enable kalloc/ kmem_cache_alloc_node/ mm_page_alloc_extfrag/ mm_page_free_batched/
filter kalloc_node/ kmem_cache_free/ mm_page_alloc_zone_locked/ mm_page_pcpu_drain/
	kfree/ kmem_cache_alloc/ mm_page_alloc/ mm_page_free/ rss_stat/
#
ls events/kmem/kalloc
enable filter format hist id inject trigger
```

Figure 6.9 – Screenshot showing the kernel's event tracing - (pseudo) files and folders under `/sys/kernel/tracing/events`

All right; a huge number of event classes and subsystems can be easily and readily traced!

Take, for example, the common kernel memory allocator routine, the really popular `kmalloc()` slab API. In Figure 6.9, you can see the pseudo-files corresponding to tracing the `kmalloc()` at the bottom (within `events/kmem/kmalloc`).

The `format` pseudo-file really has the details on what gets reported (and how it's internally looked up); it essentially represents a structure that the kernel

maintains and has the ability to lookup. (Running `kprobe-perf` with the `-v` option switch will show you this format file and won't perform tracing).

Writing `1` to the `enable` pseudo-file enables tracing and it runs under the hood. You can see the resulting output by reading the pseudo-file named `/sys/kernel/[debug]/tracing/trace` (or `trace-pipe`; reading from `trace_pipe` keeps a *watch* on the file, similar to doing a `tail -f` on a file; useful indeed).

Let's try this out; we give it a quick spin (here, on a Raspberry Pi 0W):

```
rpi # pwd
/sys/kernel/debug/tracing
rpi # cat events/kmem/kmalloc/enable
0
rpi # echo 1 > events/kmem/kmalloc/enable
rpi #
rpi # cat trace_pipe
 sshd-680 [000] 700.723280: kmalloc: call_site=__alloc_skb+0x70/0x164 ptr=236acd8 byte
s_req=576 bytes_alloc=1024 gfp_flags=GFP_KERNEL|__GFP_NOWARN|__GFP_NOMEMALLOC
 sshd-680 [000] 700.723391: kmalloc: call_site=pskb_expand_head+0x70/0x33c ptr=f5e025aa
bytes_req=1024 bytes_alloc=1024 gfp_flags=GFP_ATOMIC|__GFP_NOWARN|__GFP_NOMEMALLOC
 kworker/u2:1-56 [000] 700.723674: kmalloc: call_site=__alloc_skb+0x70/0x164 ptr=a25030bf byte
s_req=352 bytes_alloc=512 gfp_flags=GFP_ATOMIC|__GFP_NOWARN|__GFP_NOMEMALLOC
 kworker/u2:1-56 [000] 700.725507: kmalloc: call_site=__alloc_skb+0x70/0x164 ptr=a25030bf byte
s_req=352 bytes_alloc=512 gfp_flags=GFP_ATOMIC|__GFP_NOWARN|__GFP_NOMEMALLOC
 kworker/u2:1-56 [000] 700.725607: kmalloc: call_site=__alloc_skb+0x70/0x164 ptr=a25030bf byte
s_req=384 bytes_alloc=512 gfp_flags=GFP_ATOMIC|__GFP_NOWARN|__GFP_NOMEMALLOC
```

Figure 6.10 – Truncated screenshot showing an example of easily tracing the `kmalloc()`

Voila; and so easily achieved! Every single `kmalloc()` invocation – invoked by either the kernel or a module – is traced. The precise `printk` format specifier that details the `kmalloc` information (the content you see from `kmalloc: call site=... onwards`) is specified by the `events/kmem/kmalloc/format` pseudo-file.

Once done, turn the probe off with:

```
rpi # echo 0 > events/kmem/kmalloc/enable
And empty the kernel trace buffer with:
rpi # echo > trace
```

(By the way, event tracing via the `enable` pseudo-file is just one way to use the kernel's powerful event tracing framework; do refer the kernel documentation for more here: *Event Tracing*:

<https://www.kernel.org/doc/html/latest/trace/events.html#event-tracing>).

So, think on this: *Figure 6.9* shows us the automatically available tracepoints that the kernel makes available – in effect, the built-in kernel tracepoints. But what if you need to trace a function that isn't within there (in effect, that's not present under `/sys/kernel/[debug]/tracing/events`)? Well, there's always a way – the coverage of the next section!

## Setting up a dynamic kprobe (via kprobe events) on any function

To setup a kprobe dynamically on *any* given kernel (or module) function (with a few exceptions as mentioned in the section *Kprobes - limitations and downsides*), let's learn how to employ the kernel's dynamic event tracing framework and what's christened the *function based kprobes* feature.

You should realize, though, that the kprobe can only be set up if the function to be probed is:

- Present in the kernel global symbol table (`/proc/kallsyms`)
- Or, is present within the Ftrace framework's available function list (here: `<debugs_mount>/tracing/available_filter_functions`)

What if the function to be probed is within a kernel module? That's no problem: think, once the module is loaded into kernel memory, the internal machinery will ensure that all symbols are by now part of the kernel's symbol table (and will thus be visible within `/proc/kallsyms`; view it as root of course. In fact, the section that follows shows precisely this).

To set up a *dynamic kprobe*, do as follows:

1. Initialize the dynamic probe point

```
cd /sys/kernel/debug/tracing
```

If, for whatever reason, this doesn't work – typically debugfs being made invisible on a production kernel (or Ftrace being disabled) – then change it to:

```
cd /sys/kernel/tracing
```

Then:

```
echo "p:<kprobe-name> <function-to-kprobe> [...]" >> kprobe_events
```

The p: specifies you're setting up a (dynamic) kprobe; the name following the : character is any name you wish to give this probe (it will default to the function name if you don't pass anything). Then, put a space and the actual function to probe. Optional arguments can be used to specify more stuff – querying the probed function's parameter values being typical! We'll learn more as we go along...

### Tip

On production systems that are configured with the CONFIG\_DEBUG\_FS\_DISALLOW\_MOUNT=y – rendering the debugfs filesystem effectively invisible – the debugfs filesystem won't even have a mount point. In cases like this, make use of the /sys/kernel/tracing location (as shown earlier) and perform the dynamic kprobe work from therein.

Let's set this up with our usual example; setup a simple kprobe (with no additional info, like open file pathname, generated) on the function do\_sys\_open():

```
echo "p:my_sys_open do_sys_open" >> kprobe_events
```

Now that it's setup, under the /sys/kernel/[debug]/tracing/events folder you will now find a (pseudo) folder named kprobes ; it – /sys/kernel/[debug]/tracing/events/kprobes/ – *will contain any and all dynamic kprobes that have been defined*; so, here:

```
ls -lR events/kprobes/
events/kprobes/:
total 0
drwxr-xr-x 2 root root 0 Oct 9 18:58 my_sys_open/
-rw-r--r-- 1 root root 0 Oct 9 18:58 enable
-rw-r--r-- 1 root root 0 Oct 9 18:58 filter
events/kprobes/my_sys_open:
total 0
-rw-r--r-- 1 root root 0 Oct 9 18:59 enable
-rw-r--r-- 1 root root 0 Oct 9 18:58 filter
-r--r--r-- 1 root root 0 Oct 9 18:58 format
r 1
```

[...]

2. The probe is disabled by default; enable it (as root) by:

```
echo 1 > events/kprobes/my_sys_open/enable
```

Now it's enabled and running... you can look up the trace data by simply doing:

```
cat trace
[...]
 cat-192796 [001] 392192.698410: my_sys_open:
 cat-192796 [001] 392192.698650: my_sys_open:
 gnome-shell-7441 [005] 392192.777608: my_sys_open:
[...]
```

Doing a `cat trace_pipe` allows you to *watch* the file, feeding data as it becomes available – a very useful thing while using dynamic kprobe events interactively. Or, you can perhaps do something like this to save it to a file:

```
cp /sys/kernel/tracing/trace /tmp/mytrc.txt
```

3. To finish, first write `0` to the `enable` file to disable the kprobe and then do this to destroy it:

```
echo 0 > events/kprobes/my_sys_open/enable
echo "-: <kprobe-name>" >> kprobe_events
```

Alternatively, doing the following:

```
echo > /sys/kernel/tracing/kprobe_events
```

... clears *all* probe points.

So here:

```
echo 0 > events/kprobes/do_sys_open/enable
echo "-:my_sys_open" >> kprobe_events
```

Once all dynamic probe points (kprobes) are destroyed, the `/sys/kernel/[debug]/tracing/events/kprobe_events` pseudo-file itself disappears.

Doing `echo > trace` empties the kernel trace buffer of all its trace data.

Even deeper details on how to use this powerful dynamic kprobes-based event tracing are beyond the scope of this book; I refer you to the excellent kernel documentation here: *Kprobe-based Event Tracing*:

<https://www.kernel.org/doc/html/latest/trace/kprobetrace.html#kprobe-based-event-tracing>.

It's also very educative to read through the source of the `kprobe-perf` script itself: <https://github.com/brendangregg/perf-tools/blob/master/kernel/kprobe.c>.

## Taking care not to overflow or overwhelm

Do keep this in mind though! Just as mentioned with regard to our manual usage of kprobes, the `kprobe-perf` script has a similar warning within it:

```
WARNING: This uses dynamic tracing of kernel functions, and could
```

Try and mitigate this by only tracing precisely what's required for as small a time window as is feasible. The `kprobe-perf` script's `-d` option – duration specifier – is useful in this regard. It has the kernel internally buffer the output into a per-CPU buffer; the size is fixed via

`/sys/kernel/[debug]/tracing/buffer_size_kb`. If you still get overflows, try increasing its size.

## Trying this on an ARM system

### Doing the

`echo "p:my_sys_open do_sys_open" > /sys/kernel/debug/tracing/kprobe` (as root) on an ARM system will work of course... But what if we'd like to display the filename parameter to the open as well? Well, we know how to, so let's try it out:

```
echo 'p:my_sys_open do_sys_open file=+0(%si):string' > /sys/kernel/debug/tracing/kprobe
```

Whoops; why did it fail?

It should be quite obvious; the register holding the second argument – the file being opened pathname – is named `[R]SI` on the x86[\_64], but not on the ARM

processor! On ARM-32 the first four parameters to a function are piggy-backed on CPU registers `r0`, `r1`, `r2`, and `r3` (again, do refer to *Table 6.1*). So, taking this arch-dependence into account:

```
echo 'p:my_sys_open do_sys_open file=+0(%r1):string' > /sys/kernel/c
```

Now it will work!

We can go further, printing out all arguments to the open call:

```
echo 'p:my_sys_open do_sys_open dfd=%r0 file=+0(%r1):string flags=%l'
```

(Don't forget to enable the probe).

It becomes even simpler to do with the wrapper `kprobe[-perf]` script (you need the `perf-tools[-unstable]` package installed):

```
rpi # kprobe-perf 'p:my_sys_open do_sys_open dfd=%r0 file=+0(%r1):st
Tracing kprobe my_sys_open. Ctrl-C to end.
 cat-1866 [000] d... 8803.206194: my_sys_open: (do_
 cat-1866 [000] d... 8803.206548: my_sys_open: (do_
 cat-1866 [000] d... 8803.207085: my_sys_open: (do_
 cat-1866 [000] d... 8803.207235: my_sys_open: (do_
 cat-1866 [000] d... 8803.209703: my_sys_open: (do_
 cat-1866 [000] d... 8803.210395: my_sys_open: (do_
^C
Ending tracing...
rpi #
```

Interesting, yes? Do try it out yourself.

### Exercise

Setup a kprobe to trigger whenever an interrupt handler's tasklet (bottom half) routine is scheduled to execute. Also display the kernel mode stack leading up to this point.

### One solution

With traditional IRQ handling (top/bottom halves, rather than the modern thread-based IRQ handling), the top half runs with all interrupts disabled across all CPUs (guaranteeing it runs atomically) while the bottom half – the tasklet –

runs with all interrupts enabled on all processors. In this context, this is what typically occurs. A driver author, within the hardware interrupt handler (the so-called top half) typically requests the kernel to schedule it's tasklet by invoking the kernel API `schedule_tasklet()`; let's look up its underlying kernel implementation:

```
grep tasklet_schedule /sys/kernel/debug/tracing/available_filter_1
__tasklet_schedule_common
__tasklet_schedule
```

Okay; this tells us that we should setup a dynamic kprobe on the function named `__tasklet_schedule()`. Further, we pass the `-s` option switch to `kprobe-perf`, asking it to also provide a (kernel-mode) *stack trace* – in effect telling us how exactly each instance of this function invoked! This can be really useful when debugging:

```
kprobe-perf -s 'p:my __tasklet_schedule'
Tracing kprobe my. Ctrl-C to end.
 kworker/0:0-1855 [000] d.h. 9909.886809: my: (__tasklet_scl
 kworker/0:0-1855 [000] d.h. 9909.886829: <stack trace>
=> __tasklet_schedule
=> bcm2835_mmc_irq
=> __handle_irq_event_percpu
=> handle_irq_event_percpu
=> handle_irq_event
=> handle_level_irq
=> generic_handle_irq
=> __handle_domain_irq
=> bcm2835_handle_irq
=> __irq_svc
=> bcm2835_mmc_request
=> __mmc_start_request
=> mmc_start_request
=> mmc_wait_for_req
=> mmc_wait_for_cmd
=> mmc_io_rw_direct_host
=> mmc_io_rw_direct
=> process_sdio_pending_irqs
=> sdio_irq_work
=> process_one_work
=> worker_thread
=> kthread
=> ret_from_fork
[...]
```

*Figure 6.4* helps us interpret the output of the `kworker... lines`: we can see from the `d.h.` four-character sequence that interrupts are currently disabled (off) and a hardirq (a hardware interrupt handler) is running.

The remaining output – the kernel mode stack content at the time, *in effect the IRQ stack* – shows us how this particular interrupt came up and how it ended up running a tasklet (which itself internally becomes a softirq of type `TASKLET_SOFTIRQ`). Further, the stack (*always read it bottom-up*) shows that this interrupt is likely generated by I/O being performed on the **Secure Digital MultiMedia Card (SD MMC)** card.

(Again folks, FYI, the deep details regarding interrupts and their handling are covered in my earlier (freely available e-book!) book *Linux Kernel Programming – Part 2*).

## Using dynamic kprobe event tracing on a kernel module

Note that we're trying this out on our custom production kernel, to mimic a production environment.

1. First, load up our test driver, the `miscdrv_rdwr` kernel module from *Chapter 5, Debug via Instrumentation - printk and friends*:

```
$ cd <lkd-src-tree>/ch5/miscdrv_rdwr
$../../lkm
Usage: lkm name-of-kernel-module-file (without the .c)
$../../lkm miscdrv_rdwr
Version info:
Distro: Ubuntu 20.04.3 LTS
Kernel: 5.10.60-prod01
[...]
sudo dmesg

[1987.178246] miscdrv_rdwr:misscdrv_rdwr_init(): LLKD misc driv
$
```

2. A quick grep shows that its symbols are now part of the kernel global symbol table (as expected, even on our production kernel):

```
$ sudo grep miscdrv /proc/kallsyms
fffffffffc0562000 t write_miscdrv_rdwr [miscdrv_rdwr]
fffffffffc0562982 t write_miscdrv_rdwr.cold [miscdrv_rdwr]
fffffffffc00000000 + open miscdrv_rdwr [miscdrv_rdwr]
```

```
|||||||00002290 t open_misdrv_v_i_uwi [misdrv_v_i_uwi]
fffffc0562480 t close_misdrv_rdwr [misdrv_rdwr]
fffffc0562650 t read_misdrv_rdwr [misdrv_rdwr]
fffffc05629b5 t read_misdrv_rdwr.cold [misdrv_rdwr]
[...]
```

## The `.cold` compiler attribute

By the way, why are some functions suffixed with `.cold`? The short answer is that it's a compiler attribute specifying that the function is unlikely to be executed. The cold functions are typically placed in a separate linker section to improve code locality of the required-to-be-fast non-cold sections! *It's all about optimization.* Also notice that, above, there's both the normal version and the cold version of some functions (the read and write IO routines of our driver).

3. In another terminal window, let's setup a dynamic kprobe on our `write_misdrv_rdwr()` module function; as root:

```
cd /sys/kernel/tracing
echo "p:mymisdrv_wr write_misdrv_rdwr" >> kprobe_events
cat kprobe_events
p:kprobes/mymisdrv_wr write_misdrv_rdwr
#
```

We give the probed function our own name `mymisdrv_wr`. Now enable it:

```
echo 1 > events/kprobes/mymisdrv_wr/enable
```

4. Test:

- A. In one terminal window, within the tracing folder (`/sys/kernel/tracing`) run:

```
cat trace_pipe
```

- B. In another window, we run our userspace program to write to our misc class driver's device file; this will have our probe point (the `write_misdrv_rdwr()` module function) get invoked:

```
$./rdwr_test_secret w /dev/11kd_misdrv_rdwr "dyn kprobes"
```

The userspace process executes, writing into our device driver; the following screenshot shows both the execution of this userspace

THE FOLLOWING SCREENSHOT SHOWS BOTH THE EXECUTION OF THIS USERSPACE process as well as the dynamic kprobe being set up and traced:

The screenshot shows two terminal windows. The top window is titled 'root@dbg-LKD: /home/letsdebug' and contains the following command output:

```
$ lsmod |grep miscdrv_rdwr
miscdrv_rdwr 20480 0
$./rdwr_test_secret r /dev/llkd_miscdrv_rdwr
Device file /dev/llkd_miscdrv_rdwr opened (in read-only mode): fd=3
./rdwr_test_secret: read 7 bytes from /dev/llkd_miscdrv_rdwr
The 'secret' is:
"initmsg"
$./rdwr_test_secret w /dev/llkd_miscdrv_rdwr "dyn kprobes event tracing is awesome"
Device file /dev/llkd_miscdrv_rdwr opened (in write-only mode): fd=3
./rdwr_test_secret: wrote 37 bytes to /dev/llkd_miscdrv_rdwr
$./rdwr_test_secret w /dev/llkd_miscdrv_rdwr "dyn kprobes event tracing is awesome"
Device file /dev/llkd_miscdrv_rdwr opened (in write-only mode): fd=3
./rdwr_test_secret: wrote 37 bytes to /dev/llkd_miscdrv_rdwr
$
```

The bottom window is also titled 'root@dbg-LKD: /home/letsdebug' and contains the following command output:

```
cd /sys/kernel/tracing/
cat kprobe_events
grep write_misdrv_rdwr /proc/kallsyms
fffffc04f1000 t write_misdrv_rdwr [misdrv_rdwr]
fffffc04f1982 t write_misdrv_rdwr.cold [misdrv_rdwr]
echo "p:mymisdrv_wr write_misdrv_rdwr" >> kprobe_events
#
cat kprobe_events
p:kprobes/mymisdrv_wr write_misdrv_rdwr
#
echo 1 > events/kprobes/mymisdrv_wr/enable
#
cat trace_pipe
rdwr_test_secre-1557 [003] ...1 235.317228: mymisdrv_wr: (write_misdrv_rdwr+0x0/0x290 [misdrv_rdwr])
rdwr_test_secre-1558 [003] ...1 239.407952: mymisdrv_wr: (write_misdrv_rdwr+0x0/0x290 [misdrv_rdwr])
```

Figure 6.11 – Screenshot showing our testing a dynamic kprobe via the kprobe events framework on a driver module function

Study the preceding screenshot carefully; the terminal window at the bottom is where we set up our dynamic probe (corresponding to steps 3 and 4A above); the terminal window on top is where we test by invoking the write functionality of our driver (twice; in effect, this corresponds to step 4b). You can see how, in the lower terminal, the dynamic kprobe is setup and enabled. Then it *watches* for trace data by doing a `cat` on the `trace_pipe` file; when data becomes available, we see it...

Disable the probe point and destroy it with:

```
echo 0 > events/kprobes/mymisdrv_wr/enable
echo "-mymisdrv_wr" >> kprobe_events
```

```
echo -n mymills@v-wi -- kprobe_events
cat kprobe_events
echo > trace
```

The last command empties out the kernel trace buffer.

In effect, you should by now realize that what we've done in this section is pretty much automated by the `kprobe-perf` bash script! (It even has other interesting options to try). This makes it a powerful weapon in our debug / observability armory!

Before concluding this, it's good to know that even userspace application processes can be traced via the kernel's dynamic event tracing framework – this feature is called **Uprobes** (as opposed to kprobes). I refer you to the official kernel documentation on it here: *Uprobe-tracer: Uprobe-based Event Tracing*: <https://www.kernel.org/doc/html/latest/trace/uprobedtracer.html#uprobe-tracer-uprobe-based-event-tracing>.

## Setting up a return probe (kretprobe) with kprobe-perf

With the `kprobe-perf` wrapper script, you can setup a return probe – a kretprobe – as well! Using it is simplicity itself; here's our usual example, fetching the return value of the `do_sys_open()` kernel function:

```
rpi # kprobe-perf 'r:do_sys_open ret=$retval'
Tracing kprobe do_sys_open. Ctrl-C to end.
 kprobe-perf-2287 [000] d... 13013.021003: do_sys_open: (sys_<...>-2289 [000] d... 13013.027167: do_sys_open: (sys_<...>-2289 [000] d... 13013.027504: do_sys_open: (sys_
^C
Ending tracing...
rpi #
```

The key point here is the return value being fetched; it shows up as the

`ret=0x3`

This makes sense; the return to the open API is the file descriptor assigned within the process context's open file table; here, it happens to be the value 3 (with 0, 1 and 2 typically being taken up by `stdin`, `stdout`, and `stderr`).

Next, the notation

```
do_sys_open: (sys_openat+0x1c/0x20 <- do_sys_open)
```

implies that our probed function `do_sys_open()` has been called by and is returning to the function `sys_openat()`. Further, the notation `<kfunc>+0x1c/0x20` following the function name, or generically, the `+x/y`, is interpreted as:

- `x` : the offset within the function `<kfunc>` where the code returns, in effect, the return address
- `y` : what the kernel feels is the overall length of the function `<kfunc>` (it's an approximation, usually correct).

The remainder of the output is in the usual Ftrace format notation that you should be familiar with by now...

Before concluding this section, we'll mention that even more can be achieved by leveraging this powerful function-based dynamic kprobes framework within the kernel. Steven Rostedt's slides show how one can burrow ever deeper and extract pretty much any arguments to a function being probed and indeed, delve into relevant kernel structures (via offsets) to reveal their runtime values ([https://events19.linuxfoundation.org/wp-content/uploads/2017/12/oss-eu-2018-fun-with-dynamic-trace-events\\_steven-rostedt.pdf](https://events19.linuxfoundation.org/wp-content/uploads/2017/12/oss-eu-2018-fun-with-dynamic-trace-events_steven-rostedt.pdf)). Do check it out.

Well, well, we're almost done! Let's complete this chapter with one more section where you'll learn something quite practical – briefly understanding and tracing the execution of processes on the system. This can help as an audit-like facility, allowing you to log whatever userspace processes executed.

## Trapping into the `execve()` – via perf and eBPF tooling

On Linux (and UNIX), user mode applications – processes – are launched or executed via a family of so-called exec C library (glibc) APIs: `exec1()`, `execlp()`, `execv()`, `execvp()`, `execle()`, `execvpe()`, and the `execve()`.

A quick couple of things to know among these seven APIs, the first six are merely glibc wrappers that ultimately transform their arguments and invoke the `execve()` API – *it is the actual system call*, the one that causes the process context to switch to kernel mode and run the kernel code corresponding to the

system call. Also, FYI, the `execvpe()` is a GNU extension (and thus practically only seen on Linux).

The point here is simply this: ultimately, pretty much all processes' (and thus apps) are executed via the kernel code of the `execve()`! Within the kernel, the `execve()` becomes the function `sys_execve()` (in a bit of an indirect fashion, via the `SYSCALL_DEFINE3()` macro), which invokes the actual worker routine, the `do_execve()`.

## System calls and where they land in the kernel

This, in fact, is typical of many (but not all) system calls: the user-issued system call `foo()` becomes `sys_foo()`, which, if short enough performs the work itself, else invokes the actual worker routine, the `do_foo()`.

For example, the `execve(2)` system call becomes `fs/exec.c :sys_execve()` in the kernel (technically via the `SYSCALL_DEFINE3()` macro, 3 being the number of parameters passed via the syscall) which in turn invokes the worker function `fs/exec.c:do_execve()`.

Caution though, this isn't always the case... For example, the `open(2)` system call's code path within the kernel's a bit different; the following screenshot sums this up:

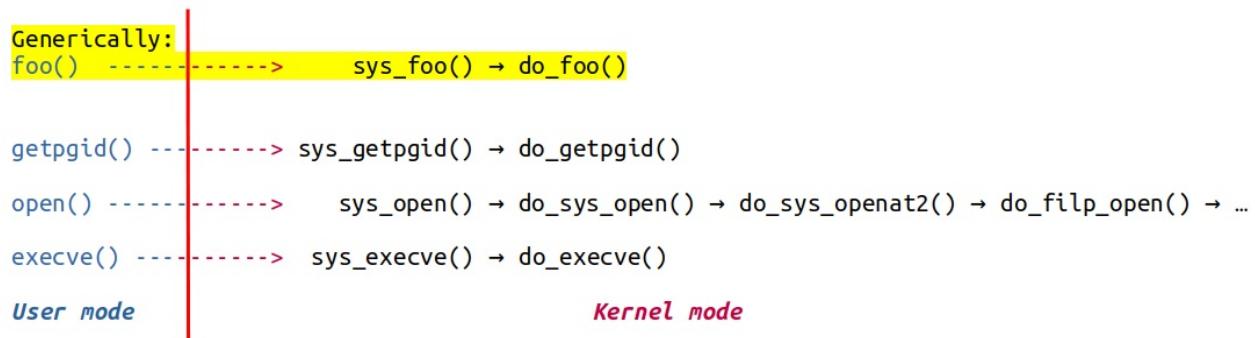


Figure 6.12 – How user mode system calls map within the kernel

An aside, but a useful one: how does a non-privileged user mode task (a process or thread) actually manage to cross the boundary from user mode into privileged kernel mode (depicted by the vertical red line in *Figure 6.12*)? The short answer is that every processor supports one or more machine instructions that allow this

to happen – these are often referred to as **call gates** or **traps** (we say that the process *traps* from user mode to kernel mode).

For example, the x86 traditionally used software interrupt `int 0x80` to perform the trap; modern versions use the `syscall` machine instruction; ARM-32 uses the `SWI` (software interrupt) machine instruction, Aarch64 (ARM64) the `SVC` (supervisor) instruction to do so. See the man page on `syscall(2)` for more detail).

(Again, FYI, there is an alternate almost equivalent system call to the `execve()` – the `execveat()`; the difference being that the first parameter to it is a directory relative to which the program – the second parameter – is executed).

Let's return to the main point: now that we know that processes are executed via the `execve()`, won't it be cool to trap into it – perhaps via injecting a kprobe into the `sys_execve()` or `do_execve()` kernel APIs? Yes, but... (there's always a *but isn't it*): on modern kernels *it simply doesn't work* (trying it via the static kprobe approach, the `register_kprobe()` fails. Please try it yourself; remember, always, the empirical approach!).

In fact, on my x86\_64 Ubuntu 20.04 LTS VM, even the `execsnoop-perf(8)` wrapper tool, built for precisely this purpose, (which internally uses the kernel's Ftrace `kprobe_events` pseudo file), fails:

```
$ sudo execsnoop-perf
Tracing exec()s. Ctrl-C to end.
ERROR: adding a kprobe for execve. Exiting.
```

The more recent eBPF tooling solves this once and for all; install and employ (as root) the `execsnoop-bpfcc(8)` and it just works! The following section has us peeking into the exec via an eBPF frontend.

## Observability with eBPF tools – an introduction

An extension of the well-known **Berkeley Packet Filter** or BPF, **eBPF** is the **extended BPF**. Very briefly, BPF used to provide the supporting infrastructure within the kernel to effectively trace network packets. eBPF is a relatively very recent kernel innovation – available only from the Linux 4.1 kernel onward. It extends the BPF notion, allowing you to trace much more than just the network stack. Also, it works for tracing both kernel space as well as userspace apps. In

effect, *eBPF and its frontends are the modern approach to tracing and performance analysis on a Linux system.*

To use BPF, you will need a system with the following:

- Linux kernel 4.1 or newer
- Kernel support for BPF (do see  
<https://github.com/iovisor/bcc/blob/master/INSTALL.md#kernel-configuration>)

Using the eBPF kernel feature directly is considered to be very hard, so there are several easier *frontends* to use. Among them, the **BPF Compiler Collection (BCC)**, **bpftrace**, and **libbpf+BPF CO-RE (Compile Once - Run Everywhere)** are regarded as being very useful. It's simplest to install the `bcc` binary packages for these front-ends; you'll find instructions regarding this here: <https://github.com/iovisor/bcc/blob/master/INSTALL.md#packages>.

Check out the following link to a picture that opens your eyes to just how many powerful BCC/BPF tools are available to help trace different Linux subsystems and hardware:

[https://www.brendangregg.com/BPF/bcc\\_tracing\\_tools\\_early2019.png](https://www.brendangregg.com/BPF/bcc_tracing_tools_early2019.png).

Here, we don't intend to delve into details; instead, we'll give you a quick flavor of using a BCC frontend utility to track processes being executed. To try this, I assume you've installed the BCC front-end package (we did this back in *Chapter 1, A General Introduction to Debugging Software*; *quick tip*: on Ubuntu, do: `sudo apt install bpfcc-tools`; but do see the following callout).

## eBPF BCC installation

You can install the BCC tools package for your regular host Linux distro by reading the installation instructions here:

<https://github.com/iovisor/bcc/blob/master/INSTALL.md>. Sometimes, though (especially on older distros, like Ubuntu 18.04), this approach of installing the `bpfcc-tools` package will typically work only on a pre-built Linux distro (like Ubuntu/Debian/RedHat/...) installation *but may not on a Linux that has a custom kernel*. The reason: the installation of the BCC toolset includes (and depends upon) the installation of the `linux-headers-$(uname -r)` package; this `linux-headers` package exists only for distro kernels (and not for our custom 5.10 kernel that we

shall often be running on the guest). With Ubuntu 20.04 LTS, it *does* seem to work, even when running a custom kernel.

Once the `bpfcc-tools` package is installed, you can get a feel for all the frontend utilities by doing:

```
dpkg -L bpfcc-tools |grep "/usr/sbin.*bpfcc$"
```

On my x86\_64 Ubuntu 20.04 LTS guest VM (running our custom 5.10.60-prod01 kernel), I find there are 112 `*-bpfcc` utilities installed (they're actually Python scripts).

In the section just prior to this one, we saw that the `execve()` (or `execveat()`) system call is the one that actually executes processes; we attempted to trace its execution via the `perf-tools` utilities (`execsnoop-perf`), but it just failed. Now, with the eBPF BCC front-ends installed, let's retry:

```
$ uname -r
5.10.60-prod01
$ sudo execsnoop-bpfcc 2>/dev/null
[...]
PCOMM PID PPID RET ARGS
id 7147 7053 0 /usr/bin/id -u
id 7148 7053 0 /usr/bin/id -u
git 7149 7053 0 /usr/bin/git config --global crea
cut 7151 7053 0 /usr/bin/cut -d= -f2
grep 7150 7053 0 /usr/bin/grep --color=auto ^PRET
cat 7152 7053 0 /usr/bin/cat /proc/version
ip 7157 7053 0 /usr/bin/ip a
sudo 7159 7053 0 /usr/bin/sudo route -n
route 7160 7159 0 /usr/sbin/route -n
[...]
```

It just works as processes get executed, the `execsnoop-bpfcc` script displays a line of output showing a few details regarding the one that just executed. Notice how all parameters to the command being executed are displayed as well. The help screen is definitely worth looking up (just run with the `-h` option switch); the man pages should be installed as well. Both have one-liner example usage; do check it out.

As with the `perf-tools` utilities, all the `*-bpfcc` scripts need to be run as root. A fair amount of noise can be generated initially; we defeat it by redirecting

`stderr` to the null device.

Our good old example – trapping into and tracing the `do_sys_open()` – right from the beginning of this chapter, can, once again, be very easily achieved with BCC:

```
$ sudo opensnoop-bpfcc 2>/dev/null
PID COMM FD ERR PATH
1431 upowerd 9 0 /sys/devices/LNXSYSTM:00/LNXSYBUS:0
1431 upowerd 9 0 /sys/devices/LNXSYSTM:00/LNXSYBUS:0
1431 upowerd -1 2 /sys/devices/LNXSYSTM:00/LNXSYBUS:0
[...]
431 systemd-udevd 14 0 /sys/fs/cgroup/unified/system.slice
431 systemd-udevd 14 0 /sys/fs/cgroup/unified/system.slice
[...] ^C
```

Again, Brendan Gregg's page on eBPF tracing tools (<https://www.brendangregg.com/ebpf.html>) will help you see the depth of tools available and how to begin making use of them.

## Summary

In this chapter you learned what kprobes and kretprobes are, how to exploit them to add useful telemetry (instrumentation) into your project or product in a dynamic fashion. We saw that one can even use them on production systems (though one should be careful to not overload the system).

We first covered the traditional static approach to using k[ret]probes, one where any change will require a recompile of the code; we even provided a semi-automated script to generate a kprobe as required. We then covered the better, efficient dynamic kprobe tracing facilities that are built-in to modern Linux kernels; using these techniques is not only a lot easier but has other advantages – they're pretty much always built-in to the kernel, no new code is required at the last minute on production systems and running them is more efficient under the hood. As a bonus, you learned how to leverage the kernel's Ftrace-based event tracepoints – a large number of kernel subsystems and their APIs can be very easily traced.

We finished this large-ish chapter by delving a bit into a practical consideration – how to trace the execution of a process (as an example). You found that the

tracing or tracking process execution, the opening of files (and in a similar fashion, most other things), can be very easily done via the modern eBPF tooling (`bpfcc-tools` BCC frontends), and, to some extent, via the `perf-tools` front-end.

The next chapter is bound to be very useful; we delve into kernel memory issues, how to find and debug them! I highly recommend you first take the time to practice (do the suggested exercises mentioned during the course of this chapter), get comfortable with the content of this and earlier chapters and then, after a quick break, jump into the next one!

## Further reading

- Official kernel documentation: *Kernel Probes (Kprobes)*:  
<https://www.kernel.org/doc/html/latest/trace/kprobes.html#kernel-probes-kprobes>
- [Kernel] *Kprobe*, Brian Pan, Nov 2020: <https://ppan-brian.medium.com/kernel-kprobe-5036d7a8455f>
- Kprobes via modern Ftrace tracing, kprobe events:
  - *Taming Tracepoints in the Linux Kernel*, Keenan, Mar 2020:  
<https://blogs.oracle.com/linux/post/taming-tracepoints-in-the-linux-kernel>
  - *Fun with Dynamic Kernel Tracing Events, The things you just shouldn't be able to do!* Steven Rostedt, Oct 2018:  
[https://events19.linuxfoundation.org/wp-content/uploads/2017/12/oss-eu-2018-fun-with-dynamic-trace-events\\_steven-rostedt.pdf](https://events19.linuxfoundation.org/wp-content/uploads/2017/12/oss-eu-2018-fun-with-dynamic-trace-events_steven-rostedt.pdf)
  - *Dynamic tracing in Linux user and kernel space*, Pratyush Anand, July 2017: <https://opensource.com/article/17/7/dynamic-tracing-linux-user-and-kernel-space> (includes coverage on user space probing with uprobe as well)
- Brendan Gregg's perf-tools page: <https://github.com/brendangregg/perf-tools>
- Specific to kprobes: kprobes-perf examples:  
[https://github.com/brendangregg/perf-tools/blob/master/examples/kprobe\\_example.txt](https://github.com/brendangregg/perf-tools/blob/master/examples/kprobe_example.txt)
- Specific to kprobes: kprobes-perf and related tooling code:  
<https://github.com/brendangregg/perf-tools/tree/master/kernel>
- *Traps, Handlers* (x86 specific):

<https://www.cse.iitd.ernet.in/~sbansal/os/lec/l8.html>

- CPU ABI, function calling and register usage conventions:
  - APPLICATION BINARY INTERFACE (ABI) DOCS AND THEIR MEANING:  
<https://kaiwantech.wordpress.com/2018/05/07/application-binary-interface-abi-docs-and-their-meaning/>
  - X86\_64
    - x64 Cheat Sheet:  
[https://cs.brown.edu/courses/cs033/docs/guides/x64\\_cheatsheet.pdf](https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf)
    - X86 64 Register and Instruction Quick Start:  
[https://wiki.cdot.senecacollege.ca/wiki/X86\\_64\\_Register\\_and\\_Instruction\\_Quick\\_Start](https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start)
    - ARM32 / Aarch32: Overview of ARM32 ABI Conventions, Microsoft, July 2018: <https://docs.microsoft.com/en-us/cpp/build/overview-of-arm-abi-conventions?view=msvc-160>
  - ARM64 / Aarch64:
    - ARMv8-A64-bit Android on ARM, Campus London, September 2015, Architecture Overview presentation:  
[https://armkeil.blob.core.windows.net/developer/Files/pdf/graphic\\_and-multimedia/ARMv8\\_Overview.pdf](https://armkeil.blob.core.windows.net/developer/Files/pdf/graphic_and-multimedia/ARMv8_Overview.pdf) [do check out the ARMv8 terminology reference on page 32]
    - Overview of ARM64 ABI conventions, Microsoft, Mar 2019:  
<https://docs.microsoft.com/en-us/cpp/build/arm64-windows-abi-conventions?view=msvc-160>
    - ARM Cortex-A Series Programmer's Guide for ARMv8-A / Fundamentals-of-ARMv8:  
<https://developer.arm.com/documentation/den0024/a/Fundamentals-of-ARMv8>
    - ARMv8 Registers:  
<https://developer.arm.com/documentation/den0024/a/ARMv8-Registers>
    - ARM64 Reversing and Exploitation Part 1 - ARM Instruction Set + Simple Heap Overflow, Sept 2020:  
<http://highaltitudehacks.com/2020/09/05/arm64-reversing-and-exploitation-part-1-arm-instruction-set-heap-overflow/>
- How Linux kprobes works, Dec 2016: <https://vjordan.info/log/fpga/how-linux-kprobes-works.html>
- eBPF:
  - Installing eBPF:  
<https://github.com/iovisor/bcc/blob/master/INSTALL.md>

- BCC tutorial:  
<https://github.com/iovisor/bcc/blob/master/docs/tutorial.md>
- *Linux Extended BPF (eBPF) Tracing Tools*, Brendan Gregg (see the pics as well!): <https://www.brendangregg.com/ebpf.html>
- *How eBPF Turns Linux into a Programmable Kernel*, Jackson, October 2020: <https://thenewstack.io/how-ebpf-turns-linux-into-a-programmable-kernel/>
- *A Gentle Introduction to eBPF*, InfoQ, May 2021:  
<https://www.infoq.com/articles/gentle-linux-ebpf-introduction/>
- (Kernel-level) *A thorough introduction to eBPF*, Matt Fleming, LWN, December 2017: <https://lwn.net/Articles/740157/>
- *How io\_uring and eBPF Will Revolutionize Programming in Linux*, Glauber Costa, April 2020: [https://thenewstack.io/how-io\\_uring-and-ebpf-will-revolutionize-programming-in-linux/](https://thenewstack.io/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/)
- Miscellaneous:
  - Old but interesting, mostly on using SystemTap: *Locating System Problems Using Dynamic Instrumentation*, Prasad, Cohen, et al, 2005: <https://sourceware.org/systemtap/systemtap-ols.pdf>
  - *Different Approaches to Linux Host Monitoring*, Kelly Shortridge, capsule8: <https://capsule8.com/blog/different-approaches-to-linux-monitoring/>

# 5 Debugging Kernel Memory Issues

## – Part 1

There's no doubt about it: C (and C++) are really powerful programming languages, one that allows the developer to straddle both high-level layered abstractions (after all, object-oriented languages like Java and Python are written in C) as well as work upon the bare metal, as it were. This is fantastic. Of course, there's a price to pay: the compiler will do only so much; you want to overflow a memory buffer? Go ahead, it doesn't care. Peek or poke an unmapped memory region? No problem.

Well, no problem for the compiler, but big problems for us! This is nothing new really. We mentioned just this in *Chapter 4, Approaches to Kernel Debugging*; C being a procedural and non-managed programming language (in memory terms), it's ultimately the programmer's responsibility to ensure that runtime memory usage is correct and well behaved.

The Linux kernel is almost entirely written in C (over 98% of the code is in C, as of the time of this writing); you see the potential for problems, yes? (In fact, there's a slowly growing effort to begin porting the kernel, or portions of it, to a more memory-safe language like **Rust**; see the *Further reading* section for links on this. In a similar vein, compilers are getting smarter; the **Clang/Low Level Virtual Machine (LLVM)** compiler – with which you can certainly build the kernel and modules – seems superior to the well-known **GNU Compiler Collection** or **GCC** compiler in terms of intelligent code generation, avoiding **Out Of Bounds (OOB)** accesses, and more. We cover some introductory material on using Clang as well here though the focus is on the most commonly used GCC compiler). Here, we'll attempt to tackle this all-too-common and stubborn bug source – memory issues! The goal, after all, is to make your code memory safe.

Due to the vast scope of material to be covered on kernel memory debugging, we split the discussion into two chapters, this one and the next.

In this chapter, we shall focus upon and cover the following main topics (look out for the detailed coverage on the kernel's SLUB debug framework and

catching memory leakage in the next one):

- What's the problem with memory anyway?
- Using KASAN and UBSAN to find memory bugs
- Building your kernel and modules with Clang
- Catching memory defects in the kernel – comparison and notes (Part 1).

## Technical requirements

The technical requirements and workspace remain identical to what's described in *Chapter 1, A General Introduction to Debugging Software*. The code examples can be found within the book's GitHub repository here:

<https://github.com/PacktPublishing/Linux-Kernel-Debugging>. The only thing new in terms of software installation is the usage of the powerful Clang compiler; we cover the details in the section *Building your kernel and modules with Clang*.

## What's the problem with memory anyway?

The introductory paragraphs at the start of this chapter informed you on the annoying fact that though programming in C is like having a superpower (at least for your typical OS/driver/embedded domains), it's a double-edged sword: we humans inadvertently create defects and bugs. Memory bugs, especially, are simply all too common.

In fact, in *Chapter 4, Approaches to Kernel Debugging*, under the *Types of bugs – the memory view* section, we mentioned that among the different ways of classifying bug types is the *memory view*; for easy recall – and to stress its importance here – I reproduce the short list of *common memory corruption bug types*:

- Incorrect memory accesses:
  - Using variables uninitialized; aka **Uninitialized Memory Read (UMR)** bugs
  - **Out-Of-Bounds (OOB)** memory accesses (read/write and underflow/overflow bugs)
  - **Use-After-Free (UAF)** and **Use-After-Return (UAR)** (aka out-of-scope) bugs

- Double-free bugs
- Memory leakage
- Data races
- (Internal) Fragmentation

These (except the last) are among the well understood **Undefined Behaviour (UB)** issues, that a process, or even the OS, can blunder into. In this chapter, you'll learn about these issues – with the emphasis being within the kernel/driver code – and, more importantly, how to use various tools and approaches to catch them.

More precisely, within this chapter we shall focus on the first two: **incorrect memory accesses** – which includes all kinds of common memory bugs: UMR, OOB, UAF/UAR, double-free. In the following chapter, we'll focus on catching memory defects in slab memory via the SLUB debug framework as well as with detecting **memory leaks**. We'll cover data races and their complexities in *Chapter 8, Lock Debugging*, (as it most commonly is caused due to incorrectly working with locks); (internal) fragmentation, or wastage, will be mentioned in the next chapter in the *Learning to use the slabinfo and related utilities* section.

### **It's not only about bugs, but also about security**

Human error and C (and C++), create an unfortunate mix at times – **bugs!** But – and here's a key point – *security issues very often tend to be bugs or defects at heart*. This is why getting it right in the first place, and/or later hunting down and fixing bugs is even more critical to today's modern production systems and, indeed the cloud (a huge portion of which is powered via the Linux kernel and its built-in hypervisor component – **Kernel Virtual Machine (KVM)**). Hackers currently have a pretty wide choice of OS-level exploits to choose from; this is especially true for older kernels. (To see what I mean, take a peek here: <https://github.com/xairy/linux-kernel-exploitation>).

If nothing else, remember: unless you're running the latest stable kernel (which will have the latest bugfix and security patches), and have configured it with security in mind as well, you're asking for trouble! (Again, see (much) more on Linux kernel security via a link in the *Further reading* section).

The goal is to have your project or product achieve **memory safety**.

## Tools to catch kernel memory issues – a quick summary

Let's get to the important thing: what tools and/or approaches are available to you when debugging kernel memory issues? Several exist; among them are:

- Directly with dynamic (runtime) analysis, specifically, memory checker, tooling:
  - **Kernel Address Sanitizer (KASAN)**
  - **Undefined Behavior Sanitizer (UBSAN)**
  - SLUB debug techniques
  - **Kernel memory leak detector (Kmemleak)**
- Indirectly with:
  - Static analysis tools: **checkpatch.pl**, **sparse**, **smatch**, **Coccinelle**, **cppcheck**, **flawfinder**, **GCC**.
  - Tracing techniques
  - K[ret]probes instrumentation
  - Post-mortem analysis tooling (logs, Oops analysis, kdump/crash, [K]GDB)

The first bullet point above – the one using which you can more or less *directly catch kernel memory defects* – is of course what we shall primarily focus on here. Subsequent chapters in this book will cover the *indirect* techniques mentioned in the second bullet point; so, patience, you'll get there. (Also, as implied by the *indirect* wording, these may or may not help you catch memory bugs).

Okay; I'll attempt to summarize this information with specifics on the tools you can use in the following table. More detailed tables will be presented later in this chapter.

| Type of memory bug or defect                             | Tool(s) / techniques to detect it    |
|----------------------------------------------------------|--------------------------------------|
| <b>Uninitialized Memory Reads ( UMR )</b>                | Compiler (warnings), static analysis |
| <b>Out-of-bounds ( OOB )</b> memory accesses: read/write | KASAN [2],                           |

|                                                                                     |                                       |
|-------------------------------------------------------------------------------------|---------------------------------------|
| underflow/overflow defects on compile-time and dynamic memory (including the stack) | SLUB debug                            |
| <b>Use-After-Free ( UAF ) or dangling pointer defects</b>                           | KASAN, SLUB debug                     |
| <b>Use-After-Return ( UAR ) defects</b>                                             | Compiler (warnings), static analysis  |
| Double-free                                                                         | Vanilla kernel, SLUB debug, KASAN [3] |
| Memory leakage                                                                      | Kmemleak                              |

Table 5.1 – A summary of tools (and techniques) you can use to detect kernel memory issues

A few notes to match the numbers in square brackets in the second column:

- [1]: Modern `gcc` / `clang` compilers definitely emit a warning for UMR, with recent ones even being able to auto-initialize local variables (if so configured)
- [2]: KASAN catches (almost!) all of them; wonderful. The SLUB debug approach can catch a couple of these, not all. Vanilla kernels don't seem to catch any
- [3]: By vanilla kernel I mean that this defect was caught on a regular distro kernel (with no special config set for memory checking)

All right! Now you know – in theory – how to catch memory bugs in the kernel or your driver, but in practice? Well, that requires you to learn to use the tool(s) mentioned above and practice! As mentioned already, understanding, configuring and learning to leverage KASAN and UBSAN (along with using Clang) is the focus of this chapter (SLUB debug and Kmemleak will be that of the next one). So, let's get on with it then.

## Using KASAN and UBSAN to find memory bugs

The **Kernel Address Sanitizer (KASAN)** is a port of the **Address Sanitizer (ASAN)** tooling to the Linux kernel. The ASAN project proved to be so useful in detecting memory-related defects that having similar abilities within the kernel was a no-brainer (ASAN is one of the few tools that could detect the

buffer overread defect that was at the root of the (in)famous so-called **Heartbleed** exploit! See the *Further reading* section for a very interesting XKCD comic link that superbly illustrates the bug at the heart of Heartbleed).

## Understanding KASAN – the basics

A few points on KASAN will help you understand more:

- KASAN is a dynamic – runtime – analysis tool; it works while the code runs. This should have you realize that unless the code actually runs (executes), KASAN will not catch any bugs; *this underlines the importance of writing really good test cases (both positive and negative), and the use of fuzzing tools to catch rarely-run code paths!* (More on this in later chapters, but it's such a key point that am stressing it here as well)
- The technology behind KASAN is called **Compile-Time Instrumentation (CTI)** (aka **static instrumentation**); here, we don't intend to go into the internals of how it works; please see the *Further reading* section for more on this. Very briefly, when the kernel is built with the GCC or Clang `-fsanitize=kernel-address` option switch, the compiler inserts assembly-level instructions to validate every memory access; further, every byte of memory is *shadowed* (tracked) using 1 byte of shadow memory to track 8 bytes of actual memory
- Overhead is relatively low (a factor of around  $2x$  to  $4x$ ; this is low, especially when compared with dynamic instrumentation approaches like Valgrind's, where the overhead can easily be  $20x$  to  $50x$ ).

Well, in terms of overhead from KASAN, it's really the RAM (more than CPU) overheads that can hurt. It does all depend on where you're coming from. For an enterprise class server system, using several megabytes of RAM as overhead for KASAN can be considered tolerable; this is likely not the case for a resource constrained embedded system (your typical Android smartphone, TV, wearable devices, low-end routers, and similar products, being good examples). For this very key reason, the modern Linux kernel supports three types, or modes, of KASAN implementations:

- **Generic KASAN** (the one we're referring to and using here, unless mentioned otherwise): high overhead and debug-only
- **Software tag-based KASAN**: Medium-to-low overhead on actual workloads. Currently ARM64 only

- **Hardware tag-based KASAN:** Low overhead and production capable. Currently ARM64 only.

The first is the default and the one to use when actively debugging (or bug hunting). It has the largest relative overhead among the three, but is very effective at bug catching! The software tag-based approach has significantly lower overhead; it's appropriate for testing actual workloads. The third hardware tag-based version has the lowest overhead and is even suitable for production use!

### Memory checking on user-mode apps?

The ASAN tooling was in fact first implemented (by Google engineers) as a GCC (and soon, Clang) patch for userspace applications. The suite includes **ASAN**, **Leak Sanitizer (LSAN)**, **Memory Sanitizer (MSAN)**, **Thread Sanitizer (TSAN)**, and **Undefined Behaviour Sanitizer (UBSAN)**. They – especially ASAN – are really powerful and is simply a *must-use for user-space app memory checking!* My earlier book *Hands-On System Programming with Linux* does cover using ASAN (and Valgrind) in some detail.

In the discussion that follows, I assume that the generic KASAN mode is being employed, primarily for the purpose of (memory) debugging. (Actually, as you'll see in the following section, this is a bit of a moot point as the other tag-based modes are currently only supported on the arm64).

## Requirements to use KASAN

Firstly, as KASAN (as well as UBSAN) are compiler-based technologies, so which compiler should you use? Both GCC and Clang are supported. You will require a relatively recent version of the compiler to be able to leverage KASAN; as of this writing, you'll need:

- **GCC version:** 8.3.0 or later
- **Clang version:** Any. For detecting OOB accesses on global variables, Clang version 11 or later is required.

The table below neatly summarizes some key information on KASAN:

| KASAN Mode               | GCC           | Clang                                   | Internal working                                                                                  | Platforms supported                                                       | Suitable for                                                                               |
|--------------------------|---------------|-----------------------------------------|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <b>Generic KASAN</b>     | = 8.3.0       | Any (>= 11 for OOB on global variables) | CTI                                                                                               | x86_64, arm, arm64, xtensa, s390, riscv                                   | Development / Debug only; global variables also instrumented; SLUB and SLAB implementation |
| Software tag-based KASAN | Not supported | CTI                                     | Currently only on arm64 (hardware-based: requires ARMv8.5 or later with Memory Tagging Extension) | Dev/Debug and Production; hardware tag-based requires SLUB implementation |                                                                                            |
| Hardware tag-based KASAN | >= 10+        | >= 11+                                  | Hardware-based                                                                                    |                                                                           |                                                                                            |

Table 5.2 – Types of KASAN and compiler/hardware support requirements

## The kernel and compilers

Traditionally, the Linux kernel has been very tightly coupled to the GCC compiler; that's slowly changing. Clang is now almost fully supported, and Rust is making an entry. In fact, FYI, Clang is typically used to compile **Android Open Source Project (AOSP)** kernels. We cover using Clang in the section *Building your kernel and modules with Clang*.

Next, hardware-wise, KASAN traditionally requires a 64-bit processor; why? Recall that it uses a shadow memory region whose size is one-eighth of the kernel virtual address space. On an x86\_64, the kernel VAS region is 128 TB (as is the user-mode **Virtual Address Space (VAS)** region); an eighth of this is significant, it's 16 terabytes. So, what platform's does KASAN actually work on? Quoting directly from the official kernel doc: *Currently, generic KASAN is*

*supported for the x86\_64, arm, arm64, xtensa, s390, and riscv architectures, and tag-based KASAN modes are supported only for arm64.*

Did you notice? Even the **arm** – implying the ARM 32-bit processor – is supported! This is a recent thing, as of the 5.11 kernel. Not only that, as of this writing at least, the lower overhead tag-based KASAN type is supported only for the arm64. (Did you pause to wonder, why arm64? Clearly, it's due to the incredible popularity of Android; many, if not most, Android devices are powered via an arm64 core within a **System on Chip (SoC)**). Detecting memory defects on Android – both in user-space and within the kernel – are critical in today's information economy; thus, tag-based KASAN modes are working on this key platform!).

In *Table 5.2*, I highlight **Generic KASAN** in bold as it's the one we're going to work with here.

## Configuring the kernel for generic KASAN mode

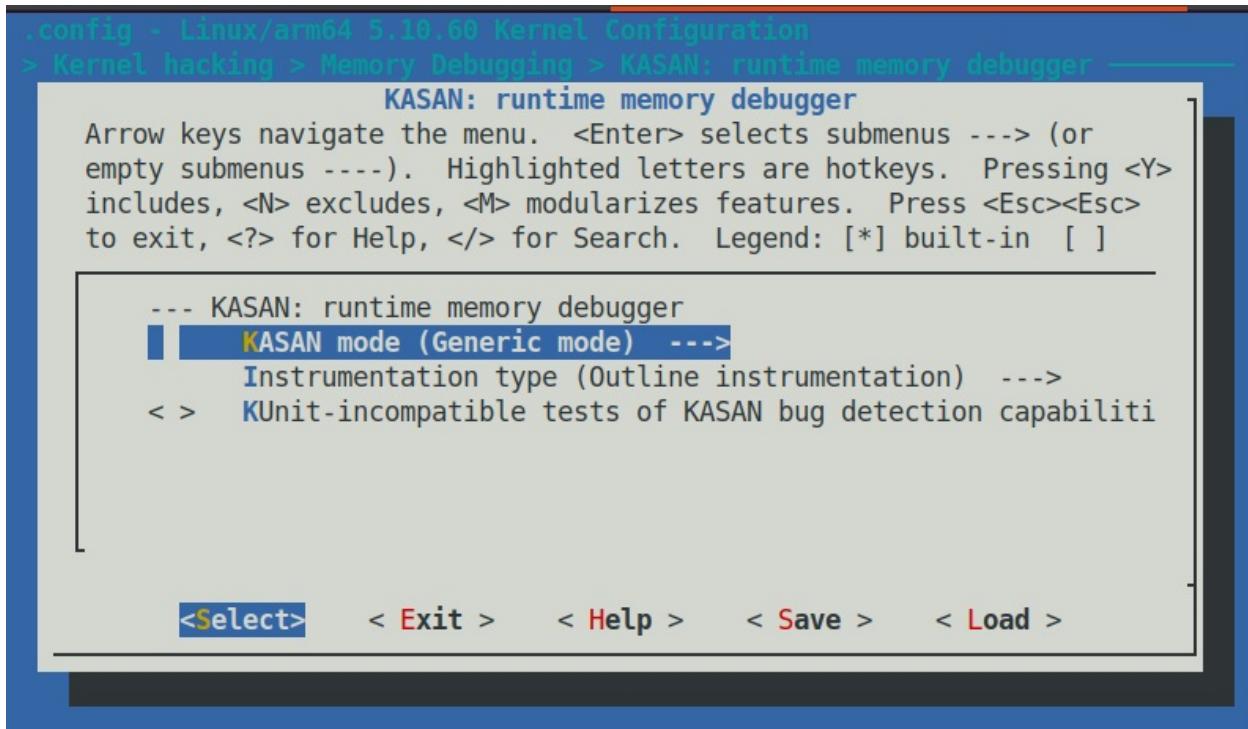
Of course, you need to configure your kernel to support generic KASAN mode. It's straightforward: enable it by setting `CONFIG_KASAN=y`. When performing the kernel config (via the usual method, the `make menuconfig`), you'll find the menu option here:

```
Kernel hacking | Memory Debugging | KASAN: runtime memory debugger
```

To make it a bit more interesting, let's configure the kernel for the arm64 by doing:

```
make ARCH=arm64 menuconfig
```

The screenshot shows you how it looks (here, we've navigated to the KASAN sub-menu):



*Figure 5.1 – Screenshot of kernel config enabling KASAN*

Keep the mode as Generic mode ; the < Help > button will show you that this corresponds to the kernel config `CONFIG_KASAN_GENERIC=y` . In fact, this Help display reveals some interesting information:

This mode consumes about 1/8th of available memory at kernel start :

Also, here, only because it's an arm64 does the kernel config option `CONFIG_HAVE_ARCH_KASAN_SW_TAGS` get initialized to y :

```
$ grep KASAN .config
CONFIG_KASAN_SHADOW_OFFSET=0xdfffffd0000000000
CONFIG_HAVE_ARCH_KASAN=y
CONFIG_HAVE_ARCH_KASAN_SW_TAGS=y
CONFIG_CC_HAS_KASAN_GENERIC=y
CONFIG_KASAN=y
CONFIG_KASAN_GENERIC=y
[...]
```

In addition, you can see how the kernel configures the shadow memory region start offset (it's a kernel virtual address of course), and other configs.

## KASAN - Effect on the build

With `CONFIG_KASAN=y`, building the kernel source tree with passing the `V=1` parameter will show the details, the `GCC` flags being passed, and more. Here's a snippet of what you see focused on the `GCC` flags passed during the build due to KASAN being enabled:

```
make V=1
```

```
gcc -Wp,-MMD,[...] -fsanitize=kernel-address
-fasan-shadow-offset=0xffffffffc00000000000 --param asan-globals=1 --|
```

KASAN works essentially by being able to check every single memory access; it does this by using a technique called **Compile Time Instrumentation (CTI)**. Put very simplistically, the compiler inserts function calls (the `__asan_load*`(`)` and `__asan_store*`(`)`) before every 1,2,4,8 or 16 byte memory access. They can figure whether the access is valid or not (by checking the corresponding shadow memory bytes). Now, there are two broad ways the compiler can perform this instrumentation: outline and inline. *Outline instrumentation* has the compiler inserting actual function calls (as just mentioned); *inline instrumentation* achieves the same thing but in a time-optimized manner by directly inserting the code (and not having the overhead of a function call)!

You can set the kernel config option `Instrumentation type` to either `CONFIG_KASAN_OUTLINE` (the default) or `CONFIG_KASAN_INLINE`. It's the typical trade-off: the *outline* type, the default, will result in a smaller kernel image while the *inline* type will result in a larger image but is faster (by a factor 1.1x to 2x).

Also, (especially for your debug kernel), it's worth enabling the kernel config `CONFIG_STACKTRACE`, so that you also obtain stack traces of the allocation and freeing of affected slab objects in the report when a bug is detected. (Similarly, turning on `CONFIG_PAGE_OWNER` – here within the menu: `Kernel hacking | Memory Debugging | Track page owner` – will get you stack traces of the allocation and freeing of affected physical pages; it's off by default; you have to boot with the parameter `page_owner=on`).

As well, when configuring an `x86_64` for KASAN, you'll find an additional kernel config regarding `vmalloc` memory corruption detection; the option shows up like this:

```
[*] Back mappings in vmalloc space with real shadow memory
```

This helps detect `vmalloc`-related memory corruption issues (at the cost of higher

~~This helps detect various related memory corruption issues (at the cost of increased memory usage during runtime).~~

So much for the theory and config; do configure and (re)build your (debug) kernel and we're good to give it a spin!

## Bug hunting with KASAN

I'll assume that by now you've configured, built and booted into your (debug) kernel that's enabled with KASAN (as the previous section has described in detail). On my setup – an x86\_64 Ubuntu 20.04 LTS guest VM – this has been done.

To test whether KASAN works, we'll need to execute code that has memory bugs (I can almost hear some of you old timers say "*Yeah? That shouldn't be too hard*"). We can always write our own test cases but why reinvent the wheel? This is a good opportunity to look at a part of the kernel's test infrastructure! The following section shows you how we'll leverage the kernel's **KUnit** unit testing framework to run KASAN testcases.

## Using the kernel's KUnit test infrastructure to run KASAN testcases

Why take the trouble to write our own test cases to test KASAN when the community has already done the work for us (ah, the beauty of open source)?

The Linux kernel has by now evolved sufficiently to have many kinds of test infrastructure, including full-fledged test suites, built into it; testing various aspects of the kernel is now a matter of configuring the kernel appropriately and running the tests!

With regard to possible built-in test frameworks within the kernel, the two primary ones are the KUnit framework and the **kselftest** framework. (FYI, the official kernel documentation of course has all details; as a start, you can see this one – *Kernel Testing Guide*: <https://www.kernel.org/doc/html/latest/dev-tools/testing-overview.html#kernel-testing-guide> – it provides a rough overview of available testing frameworks and tooling (including dynamic analysis) within the kernel).

Again, FYI, there exist several other related and useful frameworks: the kernel

fault-injection, notifier error injection, the **Linux Kernel Dump Test Module (LKDTM)**, and so on. You'll find them under the kernel config here: [Kernel hacking | Kernel Testing and Coverage](#).

Again, we don't intend to delve into the details of how KUnit works here; the idea is to merely use KUnit to test KASAN (as a practical example at this point). For the details on using these test frameworks – it will probably prove useful! – do see the links within the *Further reading* section.

As a pragmatic thing to do, and to begin getting familiar with it, let's leverage the kernel's KUnit – **Unit Testing for the Linux kernel** – framework to execute KASAN test cases!

It's really very simple to do; first, ensure your debug kernel is configured to use KUnit: `CONFIG_KUNIT=y` (or `CONFIG_KUNIT=m`)

We intend to run KASAN test cases; thus, we must have the KASAN test module configured as well:

```
CONFIG_KASAN_KUNIT_TEST=m
```

The kernel's module code for the KASAN test cases we're going to run is here: `lib/test_kasan.c`. A quick peek will show you the various test cases (there are many of them; 38, as of this writing):

```
// lib/test_kasan.c
static struct kunit_suite kasan_kunit_test_suite = {
 .name = "kasan",
 .init = kasan_test_init,
 .test_cases = kasan_kunit_test_cases,
 .exit = kasan_test_exit,
};
kunit_test_suite(kasan_kunit_test_suite);
```

This sets up the suite of test cases to execute; the actual testcases are in the `kunit_suite` structure's member named `test_cases`; it's a pointer to an array of `kunit_case` structures:

```
static struct kunit_case kasan_kunit_test_cases[] = {
 KUNIT_CASE(kmalloc_oob_right),
 KUNIT_CASE(kmalloc_oob_left),
 [...]
 KUNIT_CASE(kmalloc_double_kzfree).
```

```

-----\-----,
KUNIT_CASE(vmalloc_oob),
{}
};
```

The KUNIT\_CASE() macro sets up the testcase. To help understand how it works, here's the code for the first of the test cases:

```
// lib/test_kasan.c
static void kmalloc_oob_right(struct kunit *test)
{
 char *ptr;
 size_t size = 123;
 ptr = kmalloc(size, GFP_KERNEL);
 KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ptr);
 KUNIT_EXPECT_KASAN_FAIL(test, ptr[size + OOB_TAG_OFF] = 'x');
 kfree(ptr);
}
```

Quite intuitively, the actual checking occurs within the KUNIT\_ASSERT|EXPECT\_\*() macros seen above. The first macro asserts that the return from the kmalloc() doesn't result in an error and isn't null; the second macro – KUNIT\_EXPECT\_KASAN\_FAIL() – has the KUnit code expect failure – a negative testcase. This is indeed what should be done here: we expect that writing *beyond* the *right* side of the buffer (a write overflow defect) should trigger KASAN to report a failure! (I leave it to you to study the implementation of these macros if interested).

Furthermore, and quite interestingly, the name and exit members of the kunit\_suite structure specifies functions to execute before and after each test case is run, respectively. The module leverages this to ensure that the kernel sysctl kasan\_multi\_shot is temporarily enabled and to set panic\_on\_warn to 0 (else, only the first invalid memory access would trigger a report and a possible kernel panic!).

Finally, let's try it out!

```
$ uname -r
5.10.60-dbg01
$ sudo modprobe test_kasan
```

This will cause all test cases within the KASAN test module to execute! Looking up the kernel log (via journalctl -k or dmesg ) will show you the detailed

KASAN reports for each of the testcases. As they're voluminous, I show a sampling of the output. The very first testcase – the KUNIT\_CASE(kmalloc\_oob\_right) – causes KASAN to generate this report (its output is truncated; see more of it below, following this one):

```
[164.772135] # Subtest: kasan
[164.772149] 1..38
[164.773166] =====
[164.776786] BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0x159/0x260 [test_kasan]
[164.780268] Write of size 1 at addr ffff8880316a45fb by task kunit_try_catch/1206

[164.787155] CPU: 2 PID: 1206 Comm: kunit_try_catch Tainted: G 0 5.10.60-dbg01 #6
[164.787166] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[164.787176] Call Trace:
[164.787204] dump_stack+0xbd/0xfa
[164.787232] print_address_description.constprop.0.cold+0xd4/0x4db
[164.787257] ? trace_preempt_off+0x2a/0xf0
[164.787303] ? kmalloc_oob_right+0x159/0x260 [test_kasan]
[164.787323] kasan_report.cold+0x37/0x7c
[164.787354] ? kmalloc_oob_right+0x159/0x260 [test_kasan]
[164.787384] __asan_store1+0x6d/0x70
[164.787402] kmalloc_oob_right+0x159/0x260 [test_kasan]
[164.787415] ? kvm_sched_clock_read+0x9/0x20
[164.787436] ? kmalloc_oob_left+0x270/0x270 [test_kasan]
[164.787449] ? sched_clock_cpu+0x1b/0x1f0
[164.787480] ? kunit_binary_str_assert_format+0x100/0x100 [kunit]
[164.787523] ? lock_downgrade+0x3c0/0x3c0
[164.787540] ? mark_held_locks+0x29/0xa0
[164.787558] ? _raw_spin_unlock_irqrestore+0x55/0x70
[164.787570] ? __kthread_parkme+0x71/0x100
[164.787585] ? __this_cpu preempt_check+0x13/0x20
[164.787600] ? trace_preempt_on+0x2a/0xf0
[164.787614] ? __kthread_parkme+0x71/0x100
[164.787653] kunit_try_run_case+0x8d/0x130 [kunit]
[164.787672] ? kunitCatch_run_case+0x120/0x120 [kunit]
[164.787691] ? kunit_try_catch_throw+0x40/0x40 [kunit]
[164.787712] kunit_generic_run_threadfn_adapter+0x2e/0x50 [kunit]
[164.787733] kthread+0x22a/0x260
[164.787751] ? kthread_cancel_delayed_work_sync+0x20/0x20
[164.787777] ret_from_fork+0x22/0x30

[164.791168] Allocated by task 1206:
[164.794501] kasan_save_stack+0x23/0x50
[164.794514] __asan_kmalloc.constprop.0+0xcf/0xe0
[164.794526] kasan_kmalloc+0x9/0x10
[164.794537] kmem_cache_alloc_trace+0x1a5/0x370
[164.794553] kmalloc_oob_right+0xa3/0x260 [test_kasan]
[164.794568] kunit_try_run_case+0x8d/0x130 [kunit]
[164.794584] kunit_generic_run_threadfn_adapter+0x2e/0x50 [kunit]
[164.794597] kthread+0x22a/0x260
[164.794615] ret_from_fork+0x22/0x30
```

*Figure 5.2 – First part of the KUnit - KASAN bug catching example*

Notice, in the screenshot above:

- The first two lines, KUnit shows the test title (as # Subtest: kasan ) and that it will run testcases 1..38
- KASAN successfully, as expected of it, detected the memory defect, the

write overflow, and generates a report. The report begins with  
BUG: KASAN: [ . . . ] and the details

- The following lines reveal the root cause: (the format it's displays the offending function is `funcname( )+0xs/0xy`, where, within the function named `funcname`, the error occurred at an offset of `0xs` bytes from the start of the function, and the kernel estimates the function length to be `0xy` bytes). So here, the code in the function `kmalloc_oob_right()`, at an offset of `0x159` bytes from the start of it (followed by an educated guess of the function's length as `0x260` bytes), within the kernel module `test_kasan` (shown within square brackets on the right extreme), attempted to illegally write at the specified address. The defect, the bug, is an OOB write to a slab memory buffer, as seen by the `slab-out-of-bounds` token:

```
BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0x159/0x260
Write of size 1 at addr ffff8880316a45fb by task kunit_try_catch
```

- The following line reveals the process context within which this occurred (we'll cover the meaning of the tainted flags in the following chapter):

```
CPU: 2 PID: 1206 Comm: kunit_tryCatch Tainted: G 0
```

- The next line shows the hardware detail (you can see it's a VM, VirtualBox)
- The majority of the output is the call stack (labelled `Call Trace: ;` by reading it bottom-up (and ignoring any lines prefixed with a `?`), you can literally see how control came to this, the buggy code!)
- The line `Allocated by task 1206:` and the following output reveals the call trace of the memory allocation code path; this can be very helpful, showing by whom and where the memory buffer was allocated to begin with.

The remainder of the output can be seen in the screenshot below:

```
[164.797882] The buggy address belongs to the object at ffff8880316a4580
 which belongs to the cache kmalloc-128 of size 128
[164.804507] The buggy address is located 123 bytes inside of
 128-byte region [ffff8880316a4580, ffff8880316a4600)
[164.811106] The buggy address belongs to the page:
[164.814441] page:000000001af581d3 refcount:1 mapcount:0 mapping:0000000000000000 index:0xfffff8880316a6b00 pfn:0x316a4
[164.814452] head:000000001af581d3 order:2 compound_mapcount:0 compound_pincount:0
[164.814464] flags: 0xfffffffcc010200 (slab|head)
[164.814478] raw: 0xfffffffcc010200 fffffea0000cd0c08 fffff888001040ada fffff88800104f4c0
[164.814491] raw: fffff8880316a6b00 00000000190018 00000001ffffffff 0000000000000000
[164.814500] page dumped because: kasan: bad access detected
```

*Figure 5.3 – Second part of the KUnit - KASAN bug catching example*

As `CONFIG_PAGE_OWNER=y` (as we suggested in the section *Configuring the kernel for generic KASAN mode*), this output turns up as well; it gives you insight into where the faulty-accessed page(s) is located and its ownership:

```
[164.817779] Memory state around the buggy address:
[164.821195] ffff8880316a4480: fc
[164.824828] ffff8880316a4500: fc
[164.828377] >ffff8880316a4580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
[164.831826] ^
[164.835291] ffff8880316a4600: fc
[164.838802] ffff8880316a4680: fc
[164.842251] ======
[164.845747] Disabling lock debugging due to kernel taint
[164.846982] ok 1 - kmalloc_oob_right
[164.847514] ======
[164.850583] BUG: KASAN: slab-out-of-bounds in kmalloc_oob left+0x159/0x270 [test_kasan]
[164.853608] Read of size 1 at addr ffff88800df70a8f by task kunit try catch/1207
```

*Figure 5.4 – Third (and final) part of the KUnit - KASAN bug catching example*

In the above screenshot, you can see KASAN justifying itself; it shows the actual memory region where the defect occurred and even points out the precise byte where it did (via the `^` symbol)! As a side effect of this bug, the kernel now disables all lock debugging (as it won't behave correctly). Further, KUnit says that running this first testcase went well: `ok 1 - kmalloc_oob_right`.

Interpreting this information is important; it helps you drill down to what actually triggered the bug; we do just this in the section that follows!

## Interpreting the KASAN shadow memory output

In *Figure 5.4*, you can see the KASAN shadow memory revealing the defect's cause; we print the key line – the one prefixed with a right arrow symbol  $>$  – below:

>fffff8880318ad980: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03

^

These are the KASAN *shadow memory* bytes, each one of represents 8 bytes of actual memory. The byte `03` is pointed at (by the symbol `^`) telling us where the issue lies. What do the bytes `00`, `03`, and so on, mean? The details follow:

- Generic KASAN assigns one shadow byte to track 8 bytes of kernel memory (think of an 8 byte chunk as a *memory granule*)
- A granule (an 8 byte region) is encoded as being accessible, partially accessible, part of a redzone or free
- The encoding of a memory granule (8 byte region) by the shadow byte tracking it is done as follows:
  - **Shadow memory = 00**: All 8 bytes are accessible (no error)
  - **Shadow memory = N (where N can be a value between 1 and 7)**: The first N bytes are accessible (fine), the remaining  $(8 - N)$  bytes aren't legally accessible
  - **Shadow memory < 0**: A negative value implies the entire granule (8 bytes) is inaccessible. The particular (negative) values and their meaning (already freed up memory, redzone region, and so on) is encoded in a header file (`mm/kasan/kasan.h`).

So, now you'll realize that the shadow byte `03` implies that the memory was partially accessible; the first 3 bytes (here,  $N = 3$ ) were legally accessible, the remaining 5 ( $8 - 3 = 5$ ) bytes weren't. Let's take the trouble to verify this in detail: this is the line of code that triggers the bug, of course (it's here within the kernel code base):

```
// lib/test_kasan.c
static void kmalloc_oob_right(struct kunit *test)
{
 [...]
 size_t size = 123;
 ptr = kmalloc(size, GFP_KERNEL);
 [...]
 KUNIT_EXPECT_KASAN_FAIL(test, ptr[size + OOB_TAG_OFF] = 'x');
```

Now, the variable `size` is set to the value `123` and `OOB_TAG_OFF` is `0` when `CONFIG_KASAN_GENERIC` is enabled; so, in effect, the (buggy) code is:

```
ptr[123] = 'x';
```

Now, generic KASAN's memory granule size is 8 bytes; so, among the 123

bytes allocated, the fifteenth memory granule is the one being written to (as  $8 * 15 = 120$ ). The diagram that follows clearly shows the memory buffer and how it's been overflowed:

```
ptr = kmalloc(123, GFP_KERNEL);
ptr[123] = 'x';
```

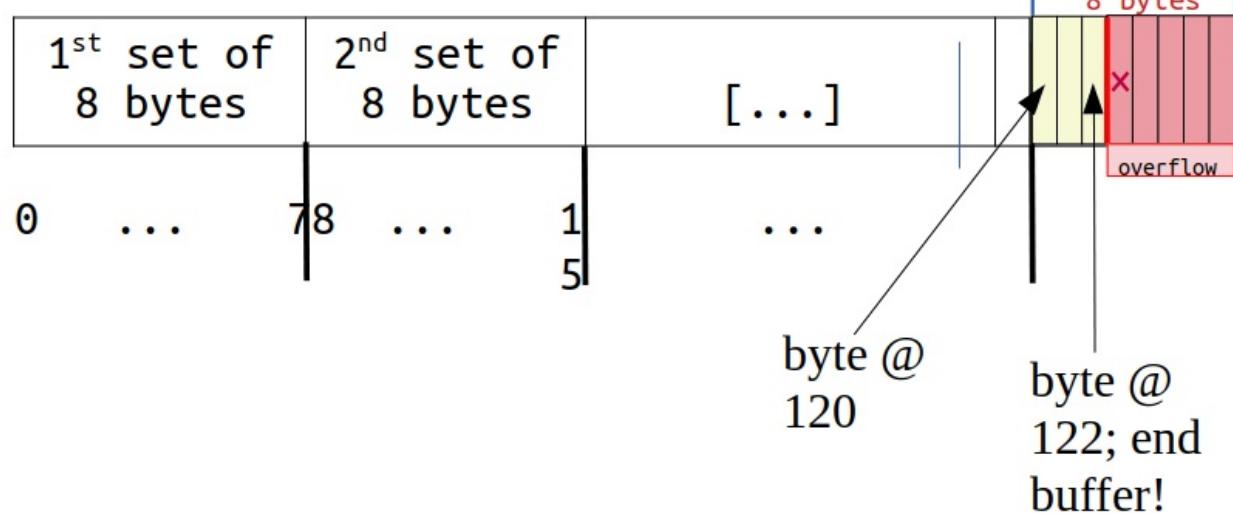


Figure 5.5 – The kmalloc’ed memory (slab) buffer that was overflowed

**Check it out:** towards the right end, byte positions 120, 121 and 122 are valid and legal to read/write; but, our KUnit KASAN test case deliberately wrote to byte position 123 – one byte beyond the end of the slab buffer, a *clear OOB write overflow violation*; and KASAN caught it! Not only that, as *Figure 5.4* and *Figure 5.5* clearly shows, the kernel is intelligent enough to show the shadow value of `03` here: implying that the first 3 bytes are valid, the remaining 5 aren’t; that’s precisely the case!

Further, the surrounding bytes are set to the value `0xfc` (see *Figure 5.4*); what does it mean? It’s clear from the header: it’s a red zone within the kernel SLUB object:

```
// mm/kasan/kasan.h
#ifndef CONFIG_KASAN_GENERIC
#define KASAN_FREE_PAGE 0xFF /* page was freed */
#define KASAN_PAGE_REDZONE 0xFE /* redzone for kmalloc_large */
#define KASAN_KMALLOC_REDZONE 0xFC /* redzone inside slab object */
#define KASAN_KMALLOC_FREE 0xFB /* object was freed (kmem_cach */
#define KASAN_KMALLOC_FREETRACK 0xFA /* object was freed and has fi
```

Back to our interpretation of *Figure 5.4*: the next line ( `BUG: KASAN: [ . . . ]` ) is just to show you that this continues with the next testcase... KASAN has now caught the second testcase's bug (the `KUNIT_CASE(kmalloc_oob_left)` ). The kernel log contains the same info as for the first defect: the bug summary by KASAN, the output of `dump_stack()` – the stack(s) call frames, who performed the allocation, the page-ownership info, the memory state around the buggy access. This continues all the way to the 38th testcase; fantastic.

A quick check of the kernel log shows what we expect: how the kernel's KUnit KASAN testcase module has caught all 38 testcases with memory defects:

```
$ journalctl -kb |grep -w "ok"
Oct 29 18:55:02 dbg-LKD kernel: ok 1 - kmalloc_oob_right
Oct 29 18:55:02 dbg-LKD kernel: ok 2 - kmalloc_oob_left
Oct 29 18:55:02 dbg-LKD kernel: ok 3 - kmalloc_node_oob_right
Oct 29 18:55:02 dbg-LKD kernel: ok 4 - kmalloc_pagealloc_oob_right
Oct 29 18:55:02 dbg-LKD kernel: ok 5 - kmalloc_pagealloc_uaf
Oct 29 18:55:02 dbg-LKD kernel: ok 6 - kmalloc_pagealloc_invalid_free
Oct 29 18:55:02 dbg-LKD kernel: ok 7 - kmalloc_large_oob_right
Oct 29 18:55:02 dbg-LKD kernel: ok 8 - kmalloc_oob_krealloc_more
Oct 29 18:55:02 dbg-LKD kernel: ok 9 - kmalloc_oob_krealloc_less
Oct 29 18:55:02 dbg-LKD kernel: ok 10 - kmalloc_oob_16
Oct 29 18:55:02 dbg-LKD kernel: ok 11 - kmalloc_uaf_16
Oct 29 18:55:02 dbg-LKD kernel: ok 12 - kmalloc_oob_in_memset
Oct 29 18:55:02 dbg-LKD kernel: ok 13 - kmalloc_oob_memset_2
Oct 29 18:55:02 dbg-LKD kernel: ok 14 - kmalloc_oob_memset_4
Oct 29 18:55:02 dbg-LKD kernel: ok 15 - kmalloc_oob_memset_8
Oct 29 18:55:02 dbg-LKD kernel: ok 16 - kmalloc_oob_memset_16
Oct 29 18:55:02 dbg-LKD kernel: ok 17 - kmalloc_memmove_invalid_size
Oct 29 18:55:02 dbg-LKD kernel: ok 18 - kmalloc_uaf
Oct 29 18:55:02 dbg-LKD kernel: ok 19 - kmalloc_uaf_memset
Oct 29 18:55:02 dbg-LKD kernel: ok 20 - kmalloc_uaf2
Oct 29 18:55:02 dbg-LKD kernel: ok 21 - kfree_via_page
Oct 29 18:55:02 dbg-LKD kernel: ok 22 - kfree_via_phys
Oct 29 18:55:03 dbg-LKD kernel: ok 23 - kmem_cache_oob
Oct 29 18:55:03 dbg-LKD kernel: ok 24 - memcg_accounted_kmem_cache
Oct 29 18:55:03 dbg-LKD kernel: ok 25 - kasan_global_oob
Oct 29 18:55:03 dbg-LKD kernel: ok 26 - kasan_stack_oob
Oct 29 18:55:03 dbg-LKD kernel: ok 27 - kasan_alloc_oob_left
Oct 29 18:55:03 dbg-LKD kernel: ok 28 - kasan_alloc_oob_right
Oct 29 18:55:03 dbg-LKD kernel: ok 29 - ksize_unpoisons_memory
Oct 29 18:55:03 dbg-LKD kernel: ok 30 - kmem_cache_double_free
Oct 29 18:55:03 dbg-LKD kernel: ok 31 - kmem_cache_invalid_free
Oct 29 18:55:03 dbg-LKD kernel: ok 32 - kasan_memchr
Oct 29 18:55:03 dbg-LKD kernel: ok 33 - kasan_memcmp
Oct 29 18:55:03 dbg-LKD kernel: ok 34 - kasan_strings
Oct 29 18:55:04 dbg-LKD kernel: ok 35 - kasan_bitops_generic
Oct 29 18:55:04 dbg-LKD kernel: ok 36 - kasan_bitops_tags
Oct 29 18:55:04 dbg-LKD kernel: ok 37 - kmalloc_double_kzfree
Oct 29 18:55:04 dbg-LKD kernel: ok 38 - vmalloc_oob
```

*Figure 5.6 – Screenshot showing how the kernel’s KUnit KASAN testcase module has caught all 38 testcases with memory defects*

As can be clearly seen from the above screenshot, all 38 testcases are reported as `ok`, passed.

## Exercise

Do perform what we've just done - running the kernel's KUnit KASAN testcases - on your box; note, from the kernel log, the various KASAN testcases and verify that all ran correctly.

By the way, notice this:

```
$ lsmod |egrep "kunit|kasan"
test_kasan 81920 0
kunit 49152 1 test_kasan
```

In my particular case, you can see from the `lsmod` output that KUnit has been configured as a kernel module.

You can learn how to write your own suite of KUnit test cases; do see the *Further reading* section for more on using KUnit!

## Remaining tests with our custom buggy kernel module

Did you notice? In spite of having run all the KASAN KUnit testcases, there appear to be a few remaining generic memory defects (as we identified both in *Chapter 4, Debug via Instrumentation – Using Kprobes*, as well as in the *What's the problem with memory anyway?* Section of this chapter) for which the KUnit testcases aren't there:

- The UMR (uninitialized memory read) bug
- The UAR (use-after-return) bug
- Simple memory leakage bugs (we'll discuss memory leakage in more detail later in this chapter)

So, I wrote a kernel module to exercise these testcases (with the *generic KASAN*-enabled debug kernel of course), along with some more interesting ones. To test against KASAN, remember to boot via your custom debug kernel, one that (obviously) has `CONFIG_KASAN=y`.

Due to space constraints, I won't show the entire code of our test LKM here (do refer to it on the book's GitHub repo and read the comments therein; you'll find it under the `ch5/kmembugs_test` folder). To get a flavor of it, let's take a peek at one of the testcases and how its invoked. Here's the code of the UAR testcase:

```

// ch5/kmembugs_test/kmembugs_test.c
/* The UAR - Use After Return - testcase */
static void *uar(void)
{
 volatile char name[NUM_ALLOC];
 volatile int i;
 for (i=0; i<NUM_ALLOC-1; i++)
 name[i] = 'x';
 name[i] = '\0';
 return name;
}

```

The module is designed to be loaded up via a bash script named `load_testmod` and the test cases are run interactively (via a bash wrapper script named `run_tests`). The `run_tests` script (which you must run as root) displays a menu of available tests and asks you to select any one, by typing in its assigned number. (You can see a screenshot of the menu – and thus all the test cases you can try out – in *Figure 5.8*, in the section that follows).

The script then writes this number to our debugfs pseudo file here:

`/sys/kernel/debug/test_kmembugs/lkd_dbgfs_run_testcase`. The debugfs write hook function then receives this data from userspace, validates it and invokes the appropriate test case routine (via a rather long `if-else-if` ladder). This design allows you to test interactively and execute any test case(s) as many times as you wish to.

Here's a code snippet showing how our debugfs module code invokes the above `uar()` test case:

```

// ch5/kmembugs_test/debugfs_kmembugs.c
static ssize_t dbgfs_run_testcase(struct file *filp, const char __user *u:
{
 char udata[MAXUPASS];
 volatile char *res1 = NULL, *res2 = NULL;
 [...]
 if (copy_from_user(udata, ubuf, count))
 return -EIO;
 udata[count-1]='\0';
 pr_debug("testcase to run: %s\n", udata);
 /* Now udata contains the data passed from userspace -
 * if (!strcmp(udata, "1", 2))
 * umr();
 * else if (!strcmp(udata, "2", 2)) {
 * res1 = uar();
```

```
pr_info("testcase 2: UAR: res1 = \"%s\"\n",
res1 == NULL ? "<whoops, it's NULL; UAR!>" : (char *)res1);
} else if (!strcmp(udata, "3.1", 4))
...
```

Clearly, this – test case #2 – is a defect, a bug: you know that local variables are valid only for their lifetime – while the function’s executing. This, of course, is as local (or automatic) variables are allocated on the (kernel mode) stack frame of the process context in execution. Thus, one must stop using a local variable once outside the scope of its containing function; we (deliberately) don’t! We attempt to fetch it as a return; the trouble is, by that time it’s gone...

Right, before diving into running the test cases (though there’s no reason you can’t run them now itself), we divert into a bit of an interesting dilemma: how a known bug (like our UAR one) can at times appear to work perfectly fine.

#### Stale frames - trouble in paradise

The amazing (or crazy) thing about bugs like this one – the UAR defect – is that the code will sometimes seem to work! How come? It’s like this: the memory holding the content of the local (automatic) variable is on the stack. Now, though we colloquially say that the stack frames are allocated on function entry and destroyed on function return (the so-called function **prologue** and **epilogue**), the reality isn’t quite so dramatic.

The reality is that memory is typically allocated at page level granularity; this includes the memory for stack pages. Thus, once a page of memory for the stack is allocated, there’s usually enough for several frames (this of course depends on the circumstances). Then, when more memory for the stack is needed, it’s grown (by allocating more pages, downwards, as it’s the stack). The system *knows* where the top of the stack is by having the **Stack Pointer (SP)** register track this memory location. (Also, you’ll realize that the so-called *top of the stack* is typically the lowest legal address). Thus, when frames are allocated and/or a function invoked, the SP register value reduces; when a function returns, the stack *shrinks* by adding to the SP register (remember, it’s a downward-growing stack!). The diagram below is a representation of a typical kernel-mode stack on a (32-bit) Linux system:

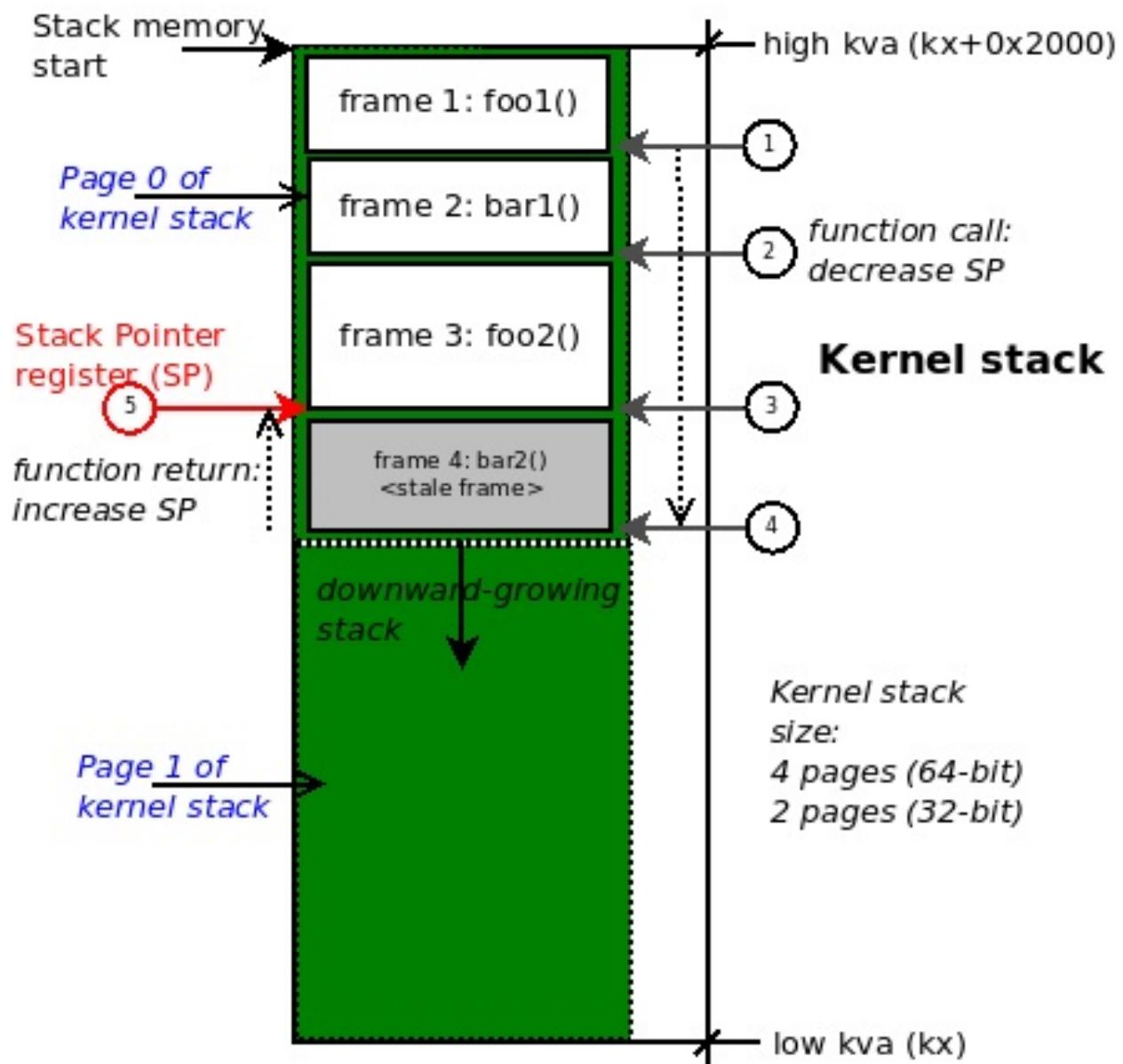


Figure 5.7 – A diagram of a typical kernel-mode stack on a 32-bit Linux

So, it could well happen at some point, that stale stack frames (and the corresponding data within them) exist underneath actually valid frames and can possibly be successfully referenced – without the system throwing a memory fault – even later.

Carefully study *Figure 5.7* above; as an example, we've shown the kernel-mode stack on a 32-bit Linux system; thus the size of the kernel stack will be 2 pages,

typically 8 KB. Now, let's say the process context in execution (within the kernel), invoked these functions in this order (this is the call chain, shown as the circled steps 1 to 4 above):

```
foo1() --> bar1() --> foo2() --> bar2()
```

Now, imagine we're at the leaf node, function `bar2()` in this example. It returns (circled step 5 above); this causes the SP register to get incremented back to the address of the call frame representing the function `foo2()`. So, though it remains intact on the stack, the stack memory of the call frame for function `bar2()` is now actually invalid! *But an incorrect – read buggy – access to it might still succeed.*

This should ideally not happen, but hey, it's an imperfect world right. The moral here: we require tools – and clear thinking is the best one – to catch tricky bugs like the UAR ones!

Right, back to our test cases! To run the tests, follow these steps:

1. Run the following command:

```
cd <book_src>/ch5/kmembugs_test
```

2. Load it up:

```
./load_testmod
[...]
```

This should have the kernel module built and loaded into memory with `dmesg` showing that the debugfs pseudo file here – `<debugfs_mountpt>/test_kmembugs/lkd_dbgfs_run testcase` – has been created

3. Run our bash script to test:

```
sudo ./run_tests
```

Below, see a screenshot showing that our `test_kmembugs` module is indeed loaded up (this was done via our `load_testmod` script), the menu shown via our `run_tests` script, and our running test case #2 – the UAR bug:

```
$ lsmod |grep test_kmembugs
test_kmembugs 61440 0
$ sudo ./run_tests
Debugfs file: /sys/kernel/debug/test_kmembugs/lkd_dbgfs_run_testcase

Generic KASAN: enabled
UBSAN: enabled
KMEMLEAK: enabled

Select testcase to run:
1 Uninitialized Memory Read - UMR
2 Use After Return - UAR

Memory leakage
3.1 simple memory leakage testcase1
3.2 simple memory leakage testcase2 - caller to free memory
3.3 simple memory leakage testcase3 - memleak in interrupt ctx

OOB accesses on static (compile-time) global memory + on stack local memory
4.1 Read (right) overflow
4.2 Write (right) overflow
4.3 Read (left) underflow
4.4 Write (left) underflow

OOB accesses on dynamic (kmalloc-ed) memory
5.1 Read (right) overflow
5.2 Write (right) overflow
5.3 Read (left) underflow
5.4 Write (left) underflow

6 Use After Free - UAF
7 Double-free

UBSAN arithmetic UB testcases
8.1 add overflow
8.2 sub overflow
8.3 mul overflow
8.4 negate overflow
8.5 shift OOB
8.6 OOB
8.7 load invalid value
8.8 misaligned access
8.9 object size mismatch

9 copy_[to|from]_user*() tests
10 UMR on slab (SLUB) memory

(Type in the testcase number to run):
2
Running testcase "2" via test module now...
[89638.348632] testcase to run: 2
[89638.350942] test_kmembugs:uar(): testcase 2: UAR:
[89638.352918] testcase 2: UAR: res1 = "<whoops, it's NULL; UAR!>"
$
```

*Figure 5.8 – Partial screenshot showing both the build and output of our kmembugs\_test LKM*

Here's an example screenshot of our test case framework catching the right OOB write buggy access via KASAN:

```

206 */
207 int global_mem_oob_left(int mode, char *p)
208 {
209 volatile char w, x, y, z;
210 volatile char local_arr[20];
211 char *volatile ptr = p - 3; // left OOB
212
213 if (mode == READ) {
214 /* Interesting: this OOB access isn't
215 w = *(volatile char *)ptr; // invalid
216
217 /* ... but these OOB accesses are caught
218 * We conclude that *only* the index-based
219 * And, KASAN compiled with clang 11
220 */
221 x = p[-3]; // invalid, OOB left read
222
223 y = local_arr[-5]; // invalid, not within bounds
224 z = local_arr[5]; // valid, within bounds
225 } else if (mode == WRITE) {
226 /* Interesting: this OOB access isn't
227 *(volatile char *)ptr = 'w';
228
229 copy_[to|from]_user*() tests
230 UMR on slab (SLUB) memory
231
232 (Type in the testcase number to run):
233 4.4
234 Running testcase "4.4" via test module now...
235 [13372.544725] testcase to run: 4.4
236 [13372.553282] =====
237 [13372.562448] BUG: KASAN: global-out-of-bounds in global_mem_oob_left+0x172/0x267 [test_kmembugs]
238 [13372.571100] Write of size 1 at addr ffffffc09aaabd by task run_tests/21489
239
240 [13372.581341] CPU: 0 PID: 21489 Comm: run_tests Tainted: G B D 0 5.10.60-dbg02-gcc #17
241 [13372.585154] Hardware name: innotech GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
242 [13372.588567] Call Trace:
243 [13372.591655] dump_stack+0xb0/0xfa
244 [13372.594362] print_address_description.constprop.0.cold+0x5/0x4db
245 [13372.597040] ? trace_preempt_off+0x2a/0xf0
246 [13372.599451] ? global_mem_oob_left+0x172/0x267 [test_kmembugs]
247 [13372.601856] kasan_report.cold+0x37/0x7c
248 [13372.603928] ? global_mem_oob_left+0x172/0x267 [test_kmembugs]
249 [13372.605993] _asan_store1+0x6d/0x70
250 [13372.607932] global_mem_oob_left+0x172/0x267 [test_kmembugs]

```

*Figure 5.9 – Partial screenshot showing KASAN catching a buggy right OOB on write to global memory*

A few things to realize:

- Firstly, the compilers, both GCC and Clang, are clever enough to warn us regarding the (here pretty obvious) bugs; both the UAR and UMR defects are indeed caught by them (at the precise place in the code where they occur), albeit as *warnings!* Here's one of the warnings emitted by GCC, clearly with regard to our UAR bug:

```

<...>/ch5/kmembugs_test/kmembugs_test.c:115:9: warning: function returns address of local variable
 115 | return (void *)name;
 ^

```

### This is important

it is your job as the programmer to carefully heed all compiler warnings and – as far as is humanely possible! – fix them.

- The script interrogates the kernel config file to see whether your current kernel is configured for KASAN and/or UBSAN, and displays what it finds; it also shows the path to the debugfs pseudo file where the test case number will be written (in order to invoke that test)

Here's a sample run of the UAR test case:

```
$ sudo ./run_tests
[...]
(Type in the testcase number to run):
2
Running testcase "2" via test module now...
[144.313592] testcase to run: 2
[144.313597] test_kmembugs:uar(): testcase 2: UAR:
[144.313600] testcase 2: UAR: res1 = "<whoops, it's NULL; UAR
$
```

- The output in the kernel log (seen via `dmesg` above) clearly tells the story: we've executed the UAR test case, and neither the kernel nor KASAN has caught it (if it had, we'd see plenty of complaints in the log!). Our own code checks for the variable `res1` being `NULL` and concludes that a UAR bug occurred; we can do this as we specifically initialized it to `NULL` and check after it's supposedly set to the string returned by the function `uar()`; else, we'd have not caught it.

All right, we're now done with several test cases with KASAN enabled; what's KASAN's scorecard like? The following section shows you just this.

## KASAN - tabulating the results

*What memory corruption bugs (defects) does KASAN actually manage to, and not manage to, catch?* From our test runs, we tabulated the results in the table that follows; do study it carefully along with the notes that go with it:

| Testcase # [1]                                                                     | Memory defect type<br>(below) / Infrastructure<br>used (right) | Compiler With<br>warning? [2] | KASAN<br>[3] |
|------------------------------------------------------------------------------------|----------------------------------------------------------------|-------------------------------|--------------|
| <b>Defects not covered by the kernel's KUnit <code>test_kasan.ko</code> module</b> |                                                                |                               |              |
| 1                                                                                  | Uninitialized Memory<br>Read - UMR                             | Y [C1]                        | N            |
| 2                                                                                  | Use After Return - UAR                                         | Y [C2]                        | N [SA]       |
| 3                                                                                  | Memory leakage [6]                                             | N                             | N            |
| <b>Defects covered by the</b>                                                      |                                                                |                               |              |

## **kernel's KUnit test\_kasan.ko module**

|     |                                                                 |        |        |
|-----|-----------------------------------------------------------------|--------|--------|
| 4   | OOB accesses on static global (compile-time) memory             |        |        |
| 4.1 | Read (right) overflow                                           | N      | Y [K1] |
| 4.2 | Write (right) overflow                                          | Y [K1] |        |
| 4.3 | Read (left) underflow                                           | Y [K2] |        |
| 4.4 | Write (left) underflow                                          | Y [K2] |        |
| 4   | OOB accesses on static global (compile-time) stack local memory |        |        |
| 4.1 | Read (right) overflow                                           | N      | Y [K3] |
| 4.2 | Write (right) overflow                                          |        |        |
| 4.3 | Read (left) underflow                                           |        |        |
| 4.4 | Write (left) underflow                                          |        |        |
| 5   | OOB accesses on dynamic (kmalloc-ed slab) memory                |        |        |
| 5.1 | Read (right) overflow                                           | N      | Y [K4] |
| 5.2 | Write (right) overflow                                          |        |        |
| 5.3 | Read (left) underflow                                           |        |        |
| 5.4 | Write (left) underflow                                          |        |        |
| 6   | Use After Free - UAF                                            | N      | Y [K5] |
| 7   | Double-free                                                     | N      | Y [K6] |
| 8   | Arithmetic UB ( <i>via the kernel's test_ubsan.ko module</i> )  |        |        |
| 8.1 | Add overflow                                                    | N      | N      |
| 8.2 | Sub(tract) overflow                                             |        |        |
| 8.3 | Mul(tiply) overflow                                             |        |        |
| 8.4 | Negate overflow                                                 |        |        |
|     | Div by zero                                                     |        |        |
| 8.5 | Bit shift OOB                                                   |        | Y [U3] |

## **Other than arithmetic UB**

## defects (copied from the kernel's KUnit test\_ubsan.ko module)

|     |                                  |        |        |
|-----|----------------------------------|--------|--------|
| 8.6 | OOB                              | N      | Y [U3] |
| 8.7 | Load invalid value               | Y [U3] |        |
| 8.8 | Misaligned access                | N      |        |
| 8.9 | Object size mismatch             | Y [U3] |        |
| 9   | OOB on<br>copy_[to from]_user*() | Y [C3] | Y [K4] |

Table 5.3 – Summary of memory defect and arithmetic UB test cases caught (or not) by KASAN

You'll find the explanations for the footnote notations seen in the table (like [C1], [U1], and so on) below.

## Test environment

- [1] The testcase number: do refer to the source of the test kernel module to see it: ch5/kmembugs\_test/kmembugs\_test.c , the debugfs entry creation and usage in debugfs\_kmembugs.c and the bash scripts load\_testmod and run\_tests , all within the same folder
- [2] The compiler used here is GCC version 9.3.0 on x86\_64 Ubuntu Linux. A later section covers using the **Clang 13** compiler
- [3] To test with KASAN, I had to boot via our custom debug kernel (5.10.60-dbg01) with CONFIG\_KASAN=y and CONFIG\_KASAN\_GENERIC=y ; we assume the *generic KASAN* variant is being used
- Testcases 4.1 through 4.4 work both upon static (compile-time allocated) global memory as well as stack local memory; that's why the testcase numbers are 4.x in both.

## Compiler warnings

- Versions: this is for GCC version 9.3.0 on x86\_64 Ubuntu:
  - [C1] The GCC compiler reports the UMR as a warning:

```
warning: '<var>' is used uninitialized in this function [-W
```

- [C2] GCC reports the potential UAF defect as a warning:

```
warning: function returns address of local variable [-Wretu
```

- [C3] GCC (quite cleverly) catches the illegal `copy_[to|from]_user()` here! it figures out that the destination size is too small:

```
* In function 'check_copy_size',
 inlined from 'copy_from_user' at ./include/linux/uaccess.h:160:10
 inlined from 'copy_user_test' at <...>/ch5/kmembugs_test.c:160:10
./include/linux/thread_info.h:160:4: error: call to '__bad_copy_to()'
 | ^~~~~~
```

- With the **Clang** 13 compiler (we cover using Clang to build the kernel and modules in the section *Building your kernel and modules with Clang*). The warnings are pretty much identical as with GCC; in addition it emits the *variable 'xxx' set but not used [-Wunused-but-set-variable]*

The section below delves into the details; don't miss out!

KASAN – detailed notes on the tabulated results

The footnote notations for KASAN ([K1], [K2], and so on) are explained in detail here. It's *really important* to read through all the notes, as we've mentioned certain caveats and corner cases as well:

- [K1] KASAN catches and reports the OOB access on global static memory as:

```
global-out-of-bounds in <func>+0xs/0xy [modname]
Read/Write of size <n> at addr <addr> by task <taskname/PID>
```

The report will contain one of `Read` or `Write` depending upon whether a read or write buggy access occurred.

- [K2] Here, there are a number of caveats to note:

- *The Out-Of-Bounds read/write left underflow on global memory test case is caught only when compiled with Clang version 11 or greater; it isn't caught even by GCC 10 or 11, due to the way it's red-zoning works.*

- *KASAN only catches global memory OOB accesses when compiled with Clang 11 and later!* Thus, in my test runs with GCC 9.3 and Clang 10, I see it fails to catch the read/write underflow (left OOB) accesses on a global buffer (testcases 4.3 and 4.4)! Here, it does seem to catch the overflow defects on global memory, though you shouldn't take this for granted... (By the way, Clang is pronounced as “clang” not “see-lang”). Also, though it's documented as supporting GCC from version 8.3.0 onward, this failed to catch (only) the read/write underflow bug testcases on global memory. (Be sure to read the upcoming section *Compiling your kernel and module with Clang!*).
  - However, even with GCC 9.3, the way the internal red-zoning and padding seems to work, it appears that the *first declared global (which variable exactly depends on how the linker sets it up) may not have a left redzone*, causing left OOB buggy accesses to be missed... This is why – as a silly workaround for now, until GCC's fixed – we use three global arrays; we pass the middle one as the test buffer to work upon (any but the first) in the test cases; hopefully, it's properly red-zoned and OOB accesses caught. This is indeed what happens in our test runs; *with this in place, the buggy left OOB accesses are caught on global memory, even when compiled with GCC 9.3!*
  - Do see the callout entitled *Why declare three global arrays and not just one?* for more on this
  - These observations, caveats, and what-have-you, are by their very nature, at times a bit *iffy*; they can end up working in one way on one system and quite another on a differently configured system or architecture. Thus, *we heartily recommend you test your workload using an appropriately configured debug kernel with all tools at your disposal, including the usage of more recent compiler technology like Clang, and the various tools and techniques covered in this book.* Yes, it's a lot of work; yes, it's worth it!
- [K3] KASAN catches and reports the OOB access on stack local memory as:
 

```
stack-out-of-bounds in <func>+0xs/0xy [modname]
Read/Write of size <n> at addr <addr> by task <taskname/PID>
```
- [K4] KASAN catches and reports the OOB access on dynamic slab memory as:

```
BUG: KASAN: slab-out-of-bounds in <func>+0xs/0xy [modname]
Read/Write of size <n> at addr <addr> by task <taskname/PID>
```

- [K5] KASAN catches and reports the **Use After Free (UAF)** as:

```
BUG: KASAN: use-after-free in <func>+0xs/0xy [modname]
Read/Write of size <n> at addr <addr> by task <taskname/PID>
```

- [K6] KASAN catches and reports the double-free as:

```
BUG: KASAN: double-free or invalid-free in <func>+0xs/0xy [modname]
```

In all the above cases, KASAN's report also shows the actual violation in detail along with the process context, (kernel-mode stack) call trace and the shadow memory map, showing which variable the OOB memory access belongs to (if applicable) and the Memory state around the buggy address.

### Tip – the all-results-in-one-place table

For your ready reference, in the Part 2 (the next chapter), in the section *Catching memory defects in the kernel – comparisons and notes (Part 2)* Table 6.\_ (in a later section), tabulates our test case results for our test runs with all the tooling technologies – vanilla/distro kernel, compiler warnings, with KASAN, with UBSAN and with SLUB debug – we employ in this chapter. In effect, it's a compilation of all the findings in one place, thus allowing you to make quick (and hopefully helpful) comparisons.

Did you notice (regarding the kernel's built-in KUnit-based test cases on KASAN) – the `test_kasan` kernel module does *not have* testcases for these three memory defects – the UMR, UAR and memory leaks; why? Simple: *KASAN does not catch these bugs!* Okay, so now what does one conclude? Well, the KUnit (and other) test suites are often run in an automated fashion where the expected end result is that all viable testcases are passed, in fact they *must* pass. This wouldn't have happened had they contained these three defects; so, they don't. Now don't read it wrong – this is simply the way the test suites are designed. There certainly exist other means besides KASAN by which these defects will be caught; relax, we'll get there and catch them.

Here and now, we're showing that KASAN itself doesn't catch these particular nasty bugs; later in the book, we'll see which tools do, allowing us to build a couple of nice tables comparing different tools and techniques for catching these

dreaded memory corruption bugs!

Good going, you now know how to leverage the power of KASAN to help catch those tricky memory bugs! Let's now move onto using UBSAN.

## Using the UBSAN kernel checker to find UB

One of the serious issues with a language like C is that the compiler produces code for the correct case, but, when the source code does something unexpected or just plain wrong, the compiler often does not understand what to do – it simply and blithely ignores such cases (this actually helps in the generation of highly optimized code at the cost of (possible security) bugs)! Examples of this are common: overflowing / underflowing an array, arithmetic defects (like dividing by zero or overflowing/underflowing a signed integer), and so on. Even worse, at times the buggy code seems to work (as we saw with accessing stale stack memory in the section *Stale frames – trouble in paradise*); similarly, bad code might work in the presence of optimization, or not. Thus, cases like these cannot be predicted and are called **Undefined Behaviour (UB)**.

The kernel's **Undefined Behaviour Sanitizer (UBSAN)** catches several types of runtime UB. As with KASAN, it uses **Compile Time Instrumentation (CTI)** to do so; with UBSAN enabled fully, the kernel code is compiled with the `-fsanitize=undefined` option switch. The UB caught by UBSAN includes:

- Arithmetic-related UB
  - Arithmetic overflow / underflow / divide by zero / and so on...
  - OOB accesses while bit shifting
- Memory-related UB
  - OOB accesses on arrays
  - NULL pointer dereferences
  - Misaligned memory accesses
  - Object size mismatches

Some of these defects in fact overlap with what generic KASAN catches as well. UBSAN instrumented code is certainly larger and slower (by a factor of 2 or 3 times); still, it's very useful – especially during development and unit testing – to catch UB defects. In fact, enabling UBSAN on production systems is feasible if you can afford the larger kernel text size and processor overheads (on everything besides tiny embedded systems, you probably can).

## Configuring the kernel for UBSAN

Within the `make menuconfig` UI, you'll find the menu system for UBSAN at  
Kernel hacking | Generic Kernel Debugging Instruments |  
Undefined behaviour sanity checker .

A screenshot of the relevant menu is seen below:

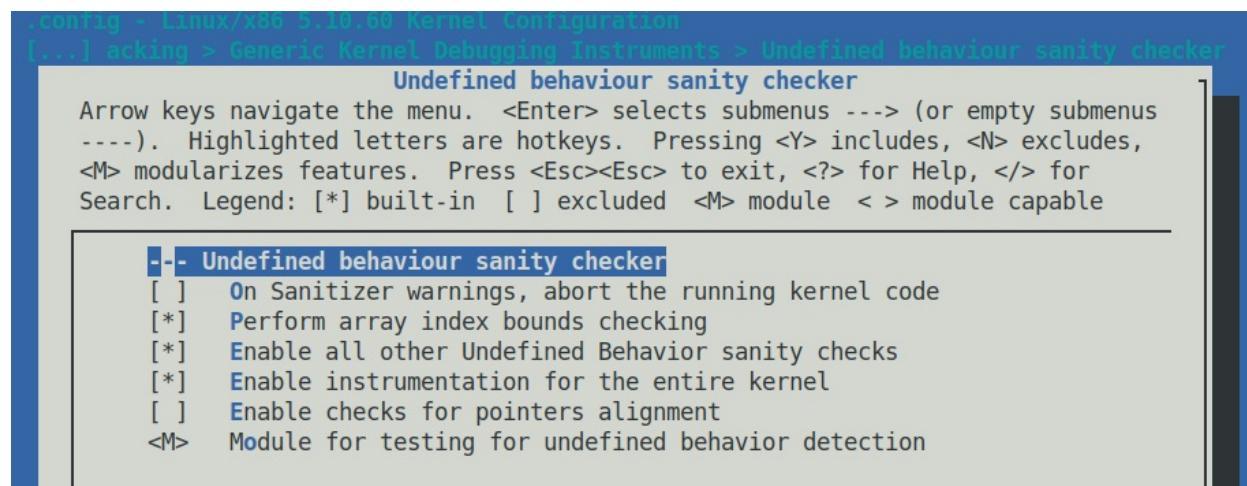


Figure 5.10 – Partial screenshot of the UBSAN menu for the Linux kernel config

To work with it, you should turn on the following kernel configs:

`CONFIG_UBSAN`, `CONFIG_UBSAN_BOUNDS` (performs bound checking on array indices for static arrays – very useful!), `CONFIG_UBSAN_MISC`, and `CONFIG_UBSAN_SANITIZE_ALL` (you can look up the details for each here: `lib/Kconfig.ubsan`). Setting `CONFIG_TEST_UBSAN=m` has the `lib/test_ubsan.c` code built as a module.

### UBSAN – Effect on the build

With `CONFIG_UBSAN=y`, building the kernel source tree with passing the `v=1` parameter will show the details, the GCC flags being passed, and more. Here's a snippet of what you see focused on the GCC flags passed during the build due to UBSAN being enabled:

```
make v=1

gcc -Wp,-MMD,[...] -fsanitize=bounds -fsanitize=shift -fsanitize=
```

## Hunting down UB with UBSAN

Detecting UB on OOB (static) array accesses (and the like) is where UBSAN shines. Take for example, our testcase #4.4; we define a few static global arrays like this:

```
static char global_arr1[10], global_arr2[10], global_arr3[10];
```

### Why declare three global arrays and not just one?

Well, as of this writing, there seems to be an issue with the way that the GCC compiler (at least as of version 9.3) sets up red zoning for global data. We observe that the redzone for the *first* global in a module may *not* have its left redzone correctly setup, causing the left OOB (underflow) buggy accesses to be missed as a side effect! So, by setting up three global arrays and passing the pointer to any but the first (we setup our testcases to pass the pointer to the second one), KASAN and UBSAN should be able to catch the buggy access! (Do note that the ordering of global variables within a module depends on the linker). This issue does not seem to occur with Clang 11+.

Interestingly, our efforts on this will eventually pay off: due to my reporting the issue - left OOB failing with GCC - as well as pointing out that the kernel's `test_kasan` module doesn't test for it, *Marco Elver* (the current KCSAN maintainer) has investigated this and added a patch to include this test case – *add globals left-out-of-bounds test* – to the `test_kasan` module (17 Nov 2021; see here: <https://lore.kernel.org/all/20211117110916.97944-1-elver@google.com/T/#u>). Further, this book's very able technical reviewer, *Chi-Thanh Hoang*, has figured out that this is essentially due to GCC's lack of a left redzone (as mentioned above), and added this information to the kernel Bugzilla ([https://bugzilla.kernel.org/show\\_bug.cgi?id=215051](https://bugzilla.kernel.org/show_bug.cgi?id=215051)). The hope is that GCC maintainers will pick this up and suggest or implement a fix.

Below, one of our buggy testcase – the right OOB accesses on global memory – access one of these global arrays, incorrectly of course, for both read and write (I only show a portion of its code here). Note that the parameter `p` is a pointer to a piece of global memory within this module, typically the second one, `global_arr2[]`:

It's invocation via our debugfs hook:

```
[...] else if (!strncmp(udata, "4.4", 4))
 global_mem_oob_left(WRITE, global_arr2);
```

The (partial) code:

```
int global_mem_oob_right(int mode, char *p)
{
 volatile char w, x, y, z;
 volatile char local_arr[20];
 char *volatile ptr = p + ARRSZ + 3; // OOB right
 [...]
} else if (mode == WRITE) {
 *(volatile char *)ptr = 'x'; // invalid, OOB right write
 p[ARRSZ - 3] = 'w'; // valid and within bounds
 p[ARRSZ + 3] = 'x'; // invalid, OOB right write
 local_arr[ARRAY_SIZE(local_arr) - 5] = 'y'; // valid and wi
 local_arr[ARRAY_SIZE(local_arr) + 5] = 'z'; // invalid, OOB
} [...]
```

Once it detects a buggy access to memory (like the ones above), UBAN displays an error report like this to the kernel log:

```
array-index-out-of-bounds in <C-source-pathname.c>:<line#>
index <index> is out of range for type '<var-type> [<size>]'
```

Here's a screenshot (below) showing just this (the right window shows the kernel log; for this, ignore the top portion of the log – it's part of the error report from KASAN; the remainder – what we're interested in – is from UBSAN):

```

167 * OOB on static (compile-time) mem: OOB read/write [13676.756743] The buggy address belongs to the variable:
168 * Covers both read/write overflow on both static g [13676.758424] global_arr2+0xd/0xffffffffffff6540 [test_kmembugs]
169 * The parameter p is a pointer to one of the global [13676.761739] Memory state around the buggy address:
170 * this module. [13676.763397] ffffffff09aa980: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
171 * Note: With gcc 10, 11 or clang < 11, KASAN isn't [13676.765267] ffffffff09aaa00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
172 * memory OOB on read/write underflow! [13676.767093] >fffffff09aaa80: 00 02 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9 f9
173 */ [13676.768736] ^
174 int global_mem_oob_right(int mode, char *p) [13676.770492] ffffffff09aab00: 00 02 f9 f9 f9 f9 f9 01 f9 f9 f9 f9 f9 f9 f9 f9 f9
175 { [13676.772293] ffffffff09aab80: 00 f9 f9 f9 f9 f9 f9 00 00 00 00 00 00 00 00 00 00
176 volatile char w, x, y, z; [13676.774652] =====
177 volatile char local_arr[20]; [13676.776211] =====
178 char *volatile ptr = p + ARRSZ + 3; // OOB right [13676.778116] UBSAN: array-index-out-of-bounds in /home/letsdebug/Linux-Kernel-Debu
179
180 if (mode == READ) { [13676.78158] t/kmembugs_test.c:194:12
181 w = *(volatile char *)ptr; // invalid, OOB [13676.78305] index 25 is out of range for type 'char [20]'
182 ptr = p + 3; [13676.783534] CPU: 5 PID: 21522 Comm: run_tests Tainted: G B D 0 5.10.60
183 x = *(volatile char *)ptr; // valid [13676.785334] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12
184
185 y = local_arr[ARRAY_SIZE(local_arr) - 5]; [13676.787197] Call Trace:
186 z = local_arr[ARRAY_SIZE(local_arr) + 5]; [13676.789020] dump_stack+0xbd/0xafa
187 } else if (mode == WRITE) { [13676.790882] ubsan_epilogue+0x9/0x45
188 *(volatile char *)ptr = 'x'; // invalid, [13676.792723] __ubsan_handle_out_of_bounds+0x70/0x80
189
190 p[ARRSZ - 3] = 'w'; // valid and within bound [13676.794687] global_mem_oob_right+0x1de/0x26f [test_kmembugs]
191 p[ARRSZ + 3] = 'x'; // invalid, OOB right w [13676.796543] ? leak_simple2+0x19b/0x19b [test_kmembugs]
192
193 local_arr[ARRAY_SIZE(local_arr) - 5] = 'y'; [13676.798693] ? __might_sleep+0x22d/0x2f0
194 local_arr[ARRAY_SIZE(local_arr) + 5] = 'z'; [13676.800734] ? __kasan_check_write+0x14/0x20
195 }
196 return 0;

```

Figure 5.11 – Partial screenshot 1 of 3 showing UBSAN catching the right OOB write to a stack local variable

Here you can see how UBSAN has precisely caught the UB on line 194 – the attempt to write after the end legal index of the local (stack-based) array! (Of course, it's entirely possible the line number we see here might change over time due to modifications to the code).

After this, our testcase # 4.3 adventurously – and disastrously – now attempts to underflow a read on a local stack memory variable; this too is cleanly caught by UBSAN! The following partial screenshot shows you the juicy bit:

```

211 char *volatile ptr = p - 3; // left OOB [13959.698401] >fffffff09aaa80: 00 02 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9 f9 f9
212
213 if (mode == READ) { [13959.700017] ^
214 /* Interesting: this OOB access isn't caught [13959.701726] ffffffff09aab00: 00 02 f9 f9 f9 f9 f9 01 f9 f9 f9 f9 f9 f9 f9 f9
215 w = *(volatile char *)ptr; // invalid, OOB [13959.703360] ffffffff09aab80: 00 f9 f9 f9 f9 f9 f9 00 00 00 00 00 00 00 00 00 00
216
217 /* ... but these OOB accesses are caught by [13959.705103] =====
218 * We conclude that *only* the index-based [13959.707343] =====
219 * And, KASAN compiled with clang 11 or later [13959.709187] UBSAN: array-index-out-of-bounds in /home/letsdebug/Linux-Kernel-Debugging/ch7/kmembugs_tes
220 */
221 x = p[-3]; // invalid, OOB left read [13959.712994] t/kmembugs_test.c:223:16
222
223 y = local_arr[-5]; // invalid, not within [13959.714762] CPU: 2 PID: 21538 Comm: run_tests Tainted: G B D 0 5.10.60-dbg02-gcc #17
224 z = local_arr[5]; // valid, within bounds [13959.716802] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
225 } else if (mode == WRITE) { [13959.718807] Call Trace:
226 /* Interesting: this OOB access isn't caught [13959.720696] dump_stack+0xbd/0xafa
227 */
228 }

```

Figure 5.12 – Partial screenshot 2 of 3 showing UBSAN catching the left OOB read on a stack local variable

Again, UBSAN even shows the source filename and line number where the buggy access was attempted!

There's more, though: UBSAN catches memory accesses when the variable in question *indexes the static memory array incorrectly* – when the index is out of bounds in any manner (left or right, underflow or overflow); it appears to miss buggy accesses made purely via pointers! KASAN has no issue with this and catches them all.

Just as we saw with KASAN (in the section *Remaining tests with our custom buggy kernel module*), UBSAN too cannot catch all memory defects. To prove this, we again run our custom buggy kernel module (in `ch5/kmembugs_test`), with pretty much identical results: *even on a UBSAN-enabled kernel, these three bugs – the UMR, UAR and memory leakage bugs - aren't caught!* The screenshot below tells the story (to capture this, I ran the `run_tests` script for the first three test cases with the `--no-clear` parameter, in order to preserve the kernel log content):

```
$ grep -w CONFIG_KASAN /boot/config-5.10.60-dbg02-gcc
CONFIG_KASAN=y
$ grep -w CONFIG_UBSAN /boot/config-5.10.60-dbg02-gcc
CONFIG_UBSAN=y
$ dmesg
[5147.233197] testcase to run: 1
[5147.233202] test_kmembugs:umr(): testcase 1: UMR (val=1039927376)
[5150.323534] testcase to run: 2
[5150.323541] test_kmembugs:uar(): testcase 2: UAR:
[5150.323546] testcase 2: UAR: res1 = "<whoops, it's NULL; UAR!>"
[5184.711447] testcase to run: 3.2
[5184.711455] test_kmembugs:leak_simple2(): testcase 3.2: simple memory leak testcase 2
[5184.711489] res2 = "leaky!!"
$
```

*Figure 5.13 – Screenshot 3 of 3: executing the first three – UMR, UAR, leakage – test cases with our test module reveal that both KASAN and UBSAN (enabled in-kernel) don't catch them*

Also, don't forget: *UBSAN is quite adept at catching arithmetic-related UB* too – things like overflowing or underflowing arithmetic calculations, the well-known **Integer OverFlow (IoF)** and the divide-by-zero bugs being common and dangerous ones indeed! (We mentioned the arithmetic UB that UBSAN can catch at the beginning of this section on UBSAN; we don't delve further into it as our topic is memory defects). To see more of UBSAN in action, you can always read the code of the UBSAN test module within the kernel (`lib/test_ubsan.c`) and try it out; I encourage you to do so. (On a somewhat related note, understanding what an *unaligned memory access* is, how it can

cause issues and how to avoid them, is the topic of this kernel documentation page: *Unaligned Memory Accesses*: <https://www.kernel.org/doc/html/latest/core-api/unaligned-memory-access.html#unaligned-memory-accesses>).

Okay, let's tabulate the result of our experiments with running various test cases with UBSAN enabled within the kernel; refer the table below:

| <b>Testcase # [1]</b>                                                 | <b>Memory defect type<br/>(below) / Infrastructure<br/>used (right)</b> | <b>Compiler<br/>warning? [2]</b> | <b>With UBSAN<br/>[3]</b> |
|-----------------------------------------------------------------------|-------------------------------------------------------------------------|----------------------------------|---------------------------|
| <b>Defects not covered by the kernel's KUnit test_kasan.ko module</b> |                                                                         |                                  |                           |
| 1                                                                     | Uninitialized Memory Read - UMR                                         | Y [C1]                           | N                         |
| 2                                                                     | Use After Return - UAR                                                  | Y [C2]                           | N [SA]                    |
| 3                                                                     | Memory leakage [6]                                                      | N                                | N                         |
| <b>Defects covered by the kernel's KUnit test_kasan.ko module</b>     |                                                                         |                                  |                           |
| 4                                                                     | OOB accesses on static global (compile-time) memory                     |                                  |                           |
| 4.1                                                                   | Read (right) overflow                                                   | N                                | Y [U1,U2]                 |
| 4.2                                                                   | Write (right) overflow                                                  |                                  |                           |
| 4.3                                                                   | Read (left) underflow                                                   |                                  |                           |
| 4.4                                                                   | Write (left) underflow                                                  |                                  |                           |
| 4                                                                     | OOB accesses on static global (compile-time) stack local memory         |                                  |                           |
| 4.1                                                                   | Read (right) overflow                                                   | N                                | Y [U1,U2]                 |

|     |                                                                |        |   |
|-----|----------------------------------------------------------------|--------|---|
| 4.2 | write (right) overflow                                         |        |   |
| 4.3 | Read (left) underflow                                          |        |   |
| 4.4 | Write (left) underflow                                         |        |   |
| 5   | OOB accesses on dynamic<br>(kmalloc-ed slab) memory            |        |   |
| 5.1 | Read (right) overflow                                          | N      | N |
| 5.2 | Write (right) overflow                                         |        |   |
| 5.3 | Read (left) underflow                                          |        |   |
| 5.4 | Write (left) underflow                                         |        |   |
| 6   | Use After Free - UAF                                           | N      | N |
| 7   | Double-free                                                    | N      | N |
| 8   | Arithmetic UB ( <i>via the kernel's test_ubsan.ko module</i> ) |        |   |
| 8.1 | Add overflow                                                   | N      | Y |
| 8.2 | Sub(tract) overflow                                            | N      |   |
| 8.3 | Mul(tiply) overflow                                            | N      |   |
| 8.4 | Negate overflow                                                | N      |   |
|     | Div by zero                                                    | Y      |   |
| 8.5 | Bit shift OOB                                                  | Y [U3] |   |

**Other than  
arithmetic UB  
defects (copied  
from the  
kernel's KUnit  
test\_ubsan.ko  
module)**

|     |                                                |        |        |
|-----|------------------------------------------------|--------|--------|
| 8.6 | OOB                                            | N      | Y [U3] |
| 8.7 | Load invalid value                             | Y [U3] |        |
| 8.8 | Misaligned access                              | N      |        |
| 8.9 | Object size mismatch                           | Y [U3] |        |
| 9   | OOB on<br><code>copy_[to from]_user*</code> () | Y [C3] | N      |

Table 5.4 - Summary of memory defect and arithmetic UB test cases caught (or not) by UBSAN

With respect to the numeric footnotes in the table above:

- [1] The testcase number: do refer to the source of the test kernel module to see it: `ch5/kmembugs_test/kmembugs_test.c`, the debugfs entry creation and usage in `debugfs_kmembugs.c` and the bash scripts `load_testmod` and `run_tests`, all within the same folder
- [2] The compiler used here is GCC version 9.3.0 on x86\_64 Ubuntu Linux. A later section covers using the **Clang 13** compiler
- [3] To test with UBSAN, I booted via our custom production kernel (5.10.60-prod01) with `CONFIG_UBSAN=y` and `CONFIG_UBSAN_SANITIZE_ALL=y`
- Testcases 4.1 through 4.4 work both upon static (compile-time allocated) global memory as well as stack local memory; that's why the testcase numbers are 4.x in both.

The section below delves into the details; don't miss out!

UBSAN – detailed notes on the tabulated results

The footnote notations seen in the table above (like [U1], [U2], and so on) are explained in detail here. It's *important* to read through all the notes, as we've mentioned certain caveats and corner cases as well:

- [U1] UBSAN catches and reports the OOB access on global static memory:

```
array-index-out-of-bounds in <C-source-pathname.c>:<line#>
index <index> is out of range for type '<var-type> [<size>]'
```

- [U2] When relevant, UBSAN also reports an object size mismatch for [U1] as:

```
object-size-mismatch in <C-source-pathname.c>:<line#>
store to address <addr> with insufficient space for an object o
```

In the above cases, UBSAN also reports the actual violation in some detail along with the process context and kernel-mode stack call trace

Note though, that with KASAN turned off (I rebuilt a test debug kernel with `CONFIG_KASAN=n`) and UBSAN turned on, the semantics seem a bit different: in this case, I got a segfault only, with of course the kernel log clearly showing the

source of the bug (via looking up what the instruction pointer register, here, RIP) was pointing to at the time of the fault.

### Note

As mentioned earlier, don't forget to look up Table 6.\_ in the following chapter, effectively, an all-results-in-one-place comparison table)

Great, now you're much better armed to catch memory bugs with both KASAN and UBSAN! I suggest you first take the time to absorb all this information, read the relevant detailed notes in the later section *Catching memory defects in the kernel – comparisons and notes* (pertaining to KASAN and UBSAN at least for now), and practice trying out these test cases on your own. But wait: we saw that some OOB defects are only caught when compiled with Clang 11 or later; this is a key thing. So, let's now learn how to use the modern Clang compiler.

## Building your kernel and modules with Clang

**Low Level Virtual Machine (LLVM)** is the original name given to this modular compiler tooling project; it now doesn't have much to do with traditional virtual machines and is instead a powerful backend for several compilers and toolchains.

**Clang** (the pronunciation rhymes with “slang”) is a modern compiler front-end technology for C-type languages (includes support for C, C++, CUDA, Objective C/C++, and more) and is based on the LLVM compiler. It's considered a drop-in replacement for GCC. Clang currently seems to have the significant advantage over GCC – especially from our point of view – of generating superior diagnostics as well as being able to intelligently generate code avoiding OOB accesses; this is critical, it paves the way to superior code. We saw (in the previous section on KASAN) that faulty left-OOB accesses on global memory, not reliably caught by GCC (versions 9.3, 10, 11), are caught with Clang! The Android project is a key user of Clang, among many others.

Attempting to build your kernel module with Clang while the target kernel itself is compiled via GCC is simply not good enough! *You'll have to use the same compiler for both* – the underlying ABI needs to be completely consistent (this was one of the many things pointed out to me by Marco Elver when I was puzzled and asked regarding why KASAN failed to catch certain test cases;

again, the beauty of opensource development). So, the upshot of it all is that we'll have to compile both our kernel and module with Clang 11.

Installing Clang and associated binaries in order to successfully compile your kernel module involves these steps is installed on our Ubuntu 20.04 LTS guest with:

```
sudo apt install clang-11 --install-suggests
```

Further, we seem to require setting up a soft link to `llvm-objdump-11` named `llvm-objdump` (this is likely as I have both Clang 10 and Clang 11 installed simultaneously):

```
sudo ln -s /usr/bin/llvm-objdump-11 /usr/bin/llvm-objdump
```

A simpler approach follows...

## Using Clang 13 on Ubuntu 21.10

For the purpose of using Clang on the kernel and module builds, instead of installing Clang 11 (or later) on Ubuntu 20.04 LTS, it might just be simpler to install Ubuntu 21.10 (I've done so as an x86\_64 VM) *as it ships with Clang 13 preinstalled*. I then built the very same 5.10.60 kernel as a debug kernel (applying a similar debug config as was discussed back in *Chapter 1, A general introduction to debugging software*, ), but this time with Clang.

When building the kernel, to specify using Clang (and not GCC) as the compiler, set the `CC` variable to it:

```
$ time make -j8 CC=clang
 SYNC include/config/auto.conf.cmd
*
* Restart config...
* Memory initialization
*
```

The first time you run this command, the `kbuild` system detects that with the Clang compiler, certain addons now become available and viable to use (which couldn't be used with GCC) and prompts us to configure it:

```
Initialize kernel stack variables at function entry
> 1. no automatic initialization (weakest) (INIT_STACK_NONE)
```

2. 0xAA-init everything on the stack (strongest) (INIT\_STACK\_ALL\_I)
3. zero-init everything on the stack (strongest and safest) (INIT\_choice[1-3?]):

Though it would be very useful to take advantage of this auto-initialization of kernel stack variables, I deliberately left it at the default (option 1) in order to check our tooling to catch the UMR defect. Similarly, the build asked regarding the following; here, I kept the defaults by simply pressing the *Enter* key; you could change it if you wish to:

```
Enable heap memory zeroing on allocation by default (INIT_ON_ALLOC_I)
Enable heap memory zeroing on free by default (INIT_ON_FREE_DEFAULT_*
*
* KASAN: runtime memory debugger
*
KASAN: runtime memory debugger (KASAN) [Y/n/?] y
 KASAN mode
 > 1. Generic mode (KASAN_GENERIC)
choice[1]: 1
[...]
 Back mappings in vmalloc space with real shadow memory (KASAN_VMAI
 KUnit-compatible tests of KASAN bug detection capabilities (KASAN_
[...]
```

Once built, perform the usual remaining steps, not forgetting to add the CC=clang :

```
sudo make CC=clang modules_install && sudo make CC=clang install
```

When done, reboot and ensure you boot into your spanking new Clang-built debug kernel!

```
$ cat /proc/version
Linux version 5.10.60-dbg02 (letsdebug@letsdebug-VirtualBox) (Ubuntu
```

Now, let's move onto building the kernel module.

```
cd <book_src>/ch5/kmembugs_test
make CC=clang
```

That's it (I've conditionally embedded this setting of the cc variable into the load\_testmod bash script, based on which compiler was used to build the current kernel. Also, FYI, to distinguish between our custom debug kernel built

with Clang and GCC, the former's `uname -r` output shows up as seen here, 5.10.60-dbg02, whereas the latter's name shows up as 5.10.60-dbg02-gcc).

## Exercise

I'll leave it as an exercise to you to build both a (debug) kernel as well as our `test_kmembugs.ko` kernel module with *Clang* and run the test cases.

With this, we complete the first part of our detailed coverage on understanding and catching memory defects within the kernel! Great going. Let's complete this chapter with a kind of summarization of the many tools and techniques we've used so far.

# Catching memory defects in the kernel – comparisons and notes (Part 1)

(As we've already mentioned in this chapter), the table below tabulates our test case results for our test runs with all the tooling technologies / kernels – vanilla/distro kernel, compiler warnings, with KASAN and UBSAN with our debug kernel – we employed in this chapter. In effect, *it's a compilation of all our findings so far in one place*, thus allowing you to make quick (and hopefully helpful) comparisons:

| Testcase # [1] Memory defect type<br>(below) / Infrastructure<br>used (right)             | Distro [2] | Compiler With<br>kernel warning? [3] | KASAN With<br>[4] | UBSAN  | [5] |
|-------------------------------------------------------------------------------------------|------------|--------------------------------------|-------------------|--------|-----|
| <b>Defects not<br/>covered by<br/>the kernel's<br/>KUnit<br/>test_kasan.ko<br/>module</b> |            |                                      |                   |        |     |
| 1      Uninitialized Memory<br>Read - UMR                                                 |            |                                      |                   |        |     |
| 2                                                                                         | N          | Y [C1]                               | N                 | N      |     |
| 3                                                                                         | N          | Y [C2]                               | N [SA]            | N [SA] |     |
| <b>Defects<br/>covered by<br/>the kernel's<br/>KUnit<br/>test_kasan.ko<br/>module</b>     |            |                                      |                   |        |     |
| 4      OOB accesses on static<br>global (compile-time)<br>memory                          |            |                                      |                   |        |     |
| 4.1                                                                                       | N [V1]     | N                                    | Y [K1]            | Y      |     |

[U1,U2]

|     |                                                                       |        |        |        |              |
|-----|-----------------------------------------------------------------------|--------|--------|--------|--------------|
| 4.2 | Write (right) overflow                                                | N      | Y [K1] |        |              |
| 4.3 | Read (left) underflow                                                 | N      | Y [K2] |        |              |
| 4.4 | Write (left) underflow                                                | N      | Y [K2] |        |              |
| 4   | OOB accesses on static<br>global (compile-time)<br>stack local memory |        |        |        |              |
| 4.1 | Read (right) overflow                                                 | N [V1] | N      | Y [K3] | Y<br>[U1,U2] |
| 4.2 | Write (right) overflow                                                | N      | Y [K2] |        |              |
| 4.3 | Read (left) underflow                                                 | N      | Y [K2] |        |              |
| 4.4 | Write (left) underflow                                                |        |        |        |              |
| 5   | OOB accesses on dynamic<br>(kmalloc-ed slab) memory                   |        |        |        |              |
| 5.1 | Read (right) overflow                                                 | N      | N      | Y [K4] | N            |
| 5.2 | Write (right) overflow                                                |        |        |        |              |
| 5.3 | Read (left) underflow                                                 |        |        |        |              |
| 5.4 | Write (left) underflow                                                |        |        |        |              |
| 6   | Use After Free - UAF                                                  | N      | N      | Y [K5] | N            |
| 7   | Double-free                                                           | Y [V2] | N      | Y [K6] | N            |
| 8   | Arithmetic UB (via the<br>kernel's test_ubsan.ko<br>module)           |        |        |        |              |
| 8.1 | Add overflow                                                          | N      | N      | N      | Y            |
| 8.2 | Sub(tract) overflow                                                   | N      |        |        |              |
| 8.3 | Mul(tiply) overflow                                                   | N      |        |        |              |
| 8.4 | Negate overflow                                                       | N      |        |        |              |
|     | Div by zero                                                           | Y      |        |        |              |
| 8.5 | Bit shift OOB                                                         | Y [U3] | Y [U3] | Y [U3] |              |

**Other than  
arithmetic  
UB defects  
(copied from  
the kernel's  
KUnit**

## **test\_ubsan.ko module)**

|     |                                                |        |        |        |        |
|-----|------------------------------------------------|--------|--------|--------|--------|
| 8.6 | OOB                                            | Y [U3] | N      | Y [U3] | Y [U3] |
| 8.7 | Load invalid value                             | Y [U3] | Y [U3] | Y [U3] |        |
| 8.8 | Misaligned access                              | N      | N      | N      |        |
| 8.9 | Object size mismatch                           | Y [U3] | Y [U3] | Y [U3] |        |
| 9   | OOB on<br><code>copy_[to from]_user*</code> () | N      | Y [C3] | Y [K4] | N      |

Table 5.5 – Summary of various common memory defects and how various technologies react in catching it (or not)

Of course, the explanation on the foot notes within this table (like [C1], [K1], [U1], and so on) will be found in the earlier relevant section.

So, a very brief summary:

- KASAN catches pretty much all OOB buggy memory accesses on global (static), stack local and dynamic (slab) memory; UBSAN doesn't catch the dynamic slab memory OOB accesses (testcases 4.x, 5.x)
- KASAN does not catch the UB defects (testcases 8.x), UBSAN does catch (most of) them
- Neither KASAN nor UBSAN catch the first three testcases – UMR, UAR and leakage bugs, *but the compiler(s) generate warnings and static analyzers (cppcheck) can catch some of them.*
- The kernel **Kmemleak** infrastructure catches kernel memory leaks allocated by any of `k{m|z}alloc()`, `vmalloc()` or `kmem_cache_alloc()` (and friends) interfaces.

Regarding the table above, a few remaining notes now follow...

## Miscellaneous notes

A few more points regarding *Table 5.5* above:

- [V1] the system could simply Oops or hang here or even appear to be remain unscathed; but that's not really the case... once the kernel is buggy, the system is buggy
- [V2] Please see the explanation for this detailed note in the following

chapter in the section *Running SLUB debug test cases on a kernel with slub\_debug turned off*.

A quick note on a KASAN alternative, especially for production follows.

## Introducing KFENCE - Kernel Electric-Fence

The Linux kernel has recent tooling named **Kernel Electric-Fence (KFENCE)**; it's available from kernel version 5.12 onward (very recent, as of this writing).

KFENCE is described as a *low-overhead sampling-based memory safety error detector of heap use-after-free, invalid-free, and out-of-bounds access errors*.

It's recently has support for both x86 and arm64 architectures with hooks to both the SLAB and SLUB memory allocators within the kernel. Why is KFENCE useful when we already have KASAN (which seems to overlap in function with it)? A few points to help differentiate between them:

- KFENCE has been designed for use in **production** systems; KASAN's overhead would be too high for typical production and is suitable only on debug / development systems; KFENCE's performance overhead is minimal, close to zero
- KFENCE works on a sampling-based design; it *trades precision for performance*; thus, with sufficiently long uptime, KFENCE will catch bugs! One way to have really long total uptime is by deploying it across a fleet of machines.
- In effect, KASAN will catch all memory defects, but at a rather high performance cost; KFENCE too can catch all memory defects, at virtually no performance cost but it takes time (high uptimes are required, as it's a sampling-based approach). Thus, to catch memory defects on debug and development systems, use KASAN (and KFENCE, perhaps), to do the same on production systems, use KFENCE.

To enable KFENCE, set `CONFIG_KFENCE=y` (note, though, that as it's very recent, this config option *isn't* present in the 5.10 kernel series we work upon in this book). You can see more options and fine-tune based on options present in the `lib/Kconfig.kfence` file.

We refer you to the details (including setup, tuning, interpreting error reports,

internal implementation, and more) in the official kernel documentation page on KFENCE here: <https://www.kernel.org/doc/html/latest/dev-tools/kfence.html#kernel-electric-fence-kfence>.

## Summary

With a non-managed programming language like ‘C’, a tradeoff exists: high power and ability to code virtually anything you can imagine, but at a significant cost: with memory being managed directly by the programmer, slipping in memory defects – bugs! – of all kinds, is rather easy to do, even for experienced folk.

In this chapter we covered many tools, techniques and approaches in this regard. First, you learned about the different (scary) types of memory defects. Then we delved into how to use various tools and techniques to identify, and thus be able to fix, them.

One of the most powerful tools in your arsenal for detecting memory bugs is KASAN; you learned how to configure and use it. We first learned how to sue the kernel’s built-in KUnit test framework to run memory test cases for KASAN to catch. We then developed our own custom module with test cases and even a neat way to test, via a debugfs pseudo file and custom scripts.

Catching UB with UBSAN came next; you learned how to configure it and leverage it to catch these kinds of defects, often overlooked, leading to not only buggy headaches but even security holes in production systems!

We learned that, while GCC is solid and been around for decades, a newer compiler, Clang, is in fact proving more adept at generating useful diagnostics (on our C code) and catching bugs that even GCC can miss! You saw how to use Clang to build the kernel and your modules, helping create more robust software, in effect.

As we covered these tools and frameworks, we tabulated the results, showing you the bugs a given tool can (or cannot) catch. To then summarize the whole thing, we built a larger table with columns covering all the test cases and all the tools – a quick and useful way for you to see and compare! (Note that we’ll add to this table in the following chapter!). Finally, we mentioned that the (very recent) KFENCE framework can (should) be used on production systems (in lieu

of KASAN).

So, congrats on completing this rather long – and really important – first chapter on catching memory bugs in kernel-space! Do take the time to digest it and practice all you've learned; when set, I encourage you to move onto the next chapter where we'll complete our coverage on catching kernel memory defects.

## Further reading

- Rust in the Linux kernel?
  - *Rust in the Linux kernel*, Apr 2021, Google security blog: <https://security.googleblog.com/2021/04/rust-in-linux-kernel.html>
  - *Let the Linux kernel Rust*, J Wallen, July 2021, TechRepublic: <https://www.techrepublic.com/article/let-the-linux-kernel-rust/>
  - *Linus Torvalds weighs in on Rust language in the Linux kernel*, ars technica, Mar 2021: <https://arstechnica.com/gadgets/2021/03/linus-torvalds-weighs-in-on-rust-language-in-the-linux-kernel/>
- Linux kernel security
  - Several links and info here; from my Linux Kernel Programming book's Further Reading section:  
[https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Further%20Reading.md#kernel\\_sec](https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Further%20Reading.md#kernel_sec)
  - *How a simple Linux kernel memory corruption bug can lead to complete system compromise*, Jann Horn, Project Zero, Oct 2021: <https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html>
- Undefined Behavior (UB) - what is it
  - Very comprehensive: A Guide to Undefined Behavior in C and C++, Part 1, John Regehr, July 2010: <https://blog.regehr.org/archives/213>
  - What Every C Programmer Should Know About Undefined Behavior #1/3, LLVM blog, May 2011: <http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>
- KASAN – the Kernel Address Sanitizer
  - Official kernel documentation: *The Kernel Address Sanitizer (KASAN)*: <https://www.kernel.org/doc/html/latest/dev-tools/kasan.html#the-kernel-address-sanitizer-kasan>
  - [K]ASAN internal working:  
<https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm>
  - The Arm64 memory tagging extension in Linux, Jon Corbet, LWN,

Oct 2020: <https://lwn.net/Articles/834289/>

- How to use KASAN to debug memory corruption in OpenStack environment: <https://www.slideshare.net/GavinGuo3/how-to-use-kasan-to-debug-memory-corruption-in-openstack-environment-2>
- Android AOSP: Building a Pixel kernel with KASAN+KCOV: <https://source.android.com/devices/tech/debug/kasan-kcov>
- FYI, the original V2 KASAN patch post: [RFC/PATCH v2 00/10] *Kernel address sainitzer (KASan) - dynamic memory error deetector.*, LWN, Sept 2014: <https://lwn.net/Articles/611410/>
- UBSAN
  - *The Undefined Behavior Sanitizer – UBSAN:* <https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html#the-undefined-behavior-sanitizer-ubsan>
  - Improving Application Security with UndefinedBehaviorSanitizer (UBSan) and GCC, Meiowitz, May 2021: <https://blogs.oracle.com/linux/post/improving-application-security-with-undefinedbehaviorsanitizer-ubsan-and-gcc>
  - Clang 13 documentation: UndefinedBehaviorSanitizer: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
  - Android AOSP: *Integer Overflow Sanitization:* <https://source.android.com/devices/tech/debug/intsan>
- Kernel built-in Test frameworks
  - KUnit - Unit Testing for the Linux Kernel: <https://www.kernel.org/doc/html/latest/dev-tools/kunit/index.html#kunit-unit-testing-for-the-linux-kernel>
  - Linux Kernel Selftests: <https://www.kernel.org/doc/html/latest/dev-tools/kselftest.html#linux-kernel-selftests>
- KFENCE: official kernel documentation (only from ver 5.12): <https://www.kernel.org/doc/html/latest/dev-tools/kfence.html#kernel-electric-fence-kfence>
- Though it's with respect to user-space, useful: Memory error checking in C and C++: Comparing Sanitizers and Valgrind, Red Hat Developer, May 2021: <https://developers.redhat.com/blog/2021/05/05/memory-error-checking-in-c-and-c-comparing-sanitizers-and-valgrind>.

# 6 Debugging Kernel Memory Issues – Part 2

Welcome to the second portion of our detailed discussions on a really key topic – understanding and learning how to detect kernel memory corruption defects. In the preceding chapter we introduced the reason why memory bugs are common and challenging and went on to cover some really important tools and technologies to help catch and defeat them – KASAN and UBSAN (along the way covering the usage of the newer Clang compiler).

In this chapter, we continue this discussion; here, we shall focus upon and cover the following main topics:

- Detecting slab memory corruption via SLUB debug
- Finding memory leakage issues with kmemleak
- Catching memory defects in the kernel – comparison and notes (Part 2)

## Technical requirements

The technical requirements and workspace remain identical to what's described in *Chapter 1, A General Introduction to Debugging Software*. The code examples can be found within the book's GitHub repository here:  
<https://github.com/PacktPublishing/Linux-Kernel-Debugging>.

## Detecting slab memory corruption via SLUB debug

**Memory corruption** can occur due to various bugs or defects: **the UMR, UAF, double-free, memory leakage** or **illegal OOB** accesses that attempt to work upon (read/write/execute) illegal memory regions. They're unfortunately a very common root cause of bugs; being able to debug them is a key skill. Having already checked out a few ways to help catch them (our detailed coverage of setting up and using KASAN and UBSAN in the previous section), let's now leverage the kernel's built-in SLUB debug features to help catch these bugs!

As you will know, memory is dynamically allocated and freed via the kernel's

engine – the *page (or Buddy System) allocator*. To mitigate the serious wastage (internal fragmentation) issues that it can face, the slab allocator (or slab cache) is layered upon it, serving two primary tasks – providing efficient fragments of pages (within the kernel, allocation requests for small pieces of memory, from a few bytes to a couple of kilobytes, tend to be very common), and serving as a cache for commonly used data structures.

Current Linux kernels typically have three mutually exclusive implementations of the slab layer – the original SLAB, *the newer and superior SLUB implementation*, and the seldom-used SLOB implementation. Its key to realize that the following discussion is with respect to *only* the SLUB (unqueued allocator) implementation of the slab layer; it's typically the default on most Linux installations (the config option is named `CONFIG_SLUB`; it's found in the menuconfig UI here: General setup | Choose SLAB allocator).

## Tip

Basic knowledge of the kernel memory management system, the page, slab allocator and the various APIs to actually allocate (and free) memory are a prerequisite for these materials. I've covered this (and much more) in the *Linux Kernel Programming* book (published by Packt in Mar 2021).

Let's quickly check out configuring the kernel for SLUB debug.

## Configuring the kernel for SLUB debug

The kernel provides a good deal of support for helping debug slab (SLUB, really) memory corruption issues; right, within the kernel config UI, you'll find:

- Under General Setup | Enable SLUB debugging support (`CONFIG_SLUB_DEBUG`)
  - Turning this on buys you plenty of SLUB debug support built-in, the ability to view all slab caches via `/sys/slab`, runtime cache validation support
  - This config is automatically turned on (auto-selected) when Generic KASAN is on
- Under Memory Debugging | SLUB debugging on by default (`CONFIG_SLUB_DEBUG_ON`; explained just below).

Let's look up the kernel config for SLUB on my x86\_64 Ubuntu guest running our custom debug kernel:

```
$ grep SLUB_DEBUG /boot/config-5.10.60-dbg02
CONFIG_SLUB_DEBUG=y
CONFIG_SLUB_DEBUG_ON is not set
```

This config implies that SLUB debugging is available but disabled by default (as `CONFIG_SLUB_DEBUG_ON` is off). While always enabling it is perhaps useful for catching memory corruption, it can have quite a large (and adverse) performance impact. To mitigate this, you can – should, really – configure your debug kernel with the `CONFIG_SLUB_DEBUG_ON` turned off by default (as seen here) and use the kernel command line parameter `slub_debug` to fine tune SLUB debugging as and when required.

The official kernel documentation here covers the usage of `slub_debug` in detail: <https://www.kernel.org/doc/html/latest/vm/slub.html>. We'll summarize the same along with some examples to demonstrate how to use this powerful feature.

## Leveraging SLUB debug features via the `slub_debug` kernel parameter

So, you'd like to leverage the `slub_debug` kernel command line parameter! To do so, let's first understand the various option flags you can pass via it at boot time:

| Flag to                          | Meaning                                                   | In more detail...                                                                            |
|----------------------------------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <b>slub_debug=</b>               |                                                           |                                                                                              |
| <b>on kernel</b>                 |                                                           |                                                                                              |
| <b>cmdline</b>                   |                                                           |                                                                                              |
| null (pass nothing after the = ) | Switch all SLUB debugging on                              | All checks as specified by all the = )                                                       |
| F                                | Sanity checks on (enables SLAB_DEBUG_CONSISTENCY_CHECKS ) | Performs (expensive) sanity checks minimally enables the double-pseudofile: /sys/kernel/slab |

|   |                                                   |                                                                                                                                                                                                                                                                                                  |
|---|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|   |                                                   | <b>Tip:</b> this option by itself can be useful for catching systems when a memory corruption occurs. See this from the official kernel documentation: <a href="https://www.kernel.org/doc/htmldocs/debugging.html#slub-debug">https://www.kernel.org/doc/htmldocs/debugging.html#slub-debug</a> |
| Z | Red zoning                                        | Cache objects will be red zone<br>Corresponding sysfs pseudofiles: /sys/kernel/slab/<slabname>/red_zoned                                                                                                                                                                                         |
| P | Poisoning (object and padding)                    | Corresponding sysfs pseudofiles: /sys/kernel/slab/<slabname>/poisoned                                                                                                                                                                                                                            |
| U | User tracking (free and alloc)                    | See the section <i>Understanding User Tracking</i><br>Stores last owner; useful for cache accounting                                                                                                                                                                                             |
| T | Trace (overhead; should only use on single slabs) | Traces alloc's and frees of objects<br>Corresponding sysfs pseudofiles: /sys/kernel/slab/<slabname>/tracing                                                                                                                                                                                      |
| A | Enable failslab filter mark for the cache         | For fault injection purposes                                                                                                                                                                                                                                                                     |
| 0 | Switch SLUB debugging off                         | Applies to caches that would have orders                                                                                                                                                                                                                                                         |
| - | Switch all SLUB debugging off                     | Can be very useful (to remove when the kernel is configured)                                                                                                                                                                                                                                     |

Table 6.1 – The `slub_debug=<NNN>` flags and corresponding sysfs entries if any

A brief description of pretty much every (pseudo) file under `/sys/kernel/slab/<slabname>` can be found in the kernel documentation here (a word of caution, it seems to be quite aged):

<https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-kernel-slab>.

# Understanding the SLUB layer's poison flags

The poison flags defined by the kernel are defines as follows:

```
// include/linux/poison.h
#define POISON_INUSE 0x5a /* for use-uninitialised poisoning */
#define POISON_FREE 0x6b /* for use-after-free poisoning */
#define POISON_END 0xa5 /* end-byte of poisoning */
```

Here's the nitty gritty on these poison values:

- A bit non-intuitively, when you use the `SLAB_POISON` flag when creating a slab cache (typically via the `kmem_cache_create()` kernel API), or set poisoning on via the kernel parameter `slob_debug=P`, the slab memory gets auto-initialized to the value `0x6b` (which is ASCII `k`)
  - The `POISON_INUSE` value (`0x5a` equals to ASCII `z`) is used to denote padding zones, before or after redzones
  - The last legal byte of the slab memory object is set to `POISON_END`, `0xa5`

(You'll come across a nice example of seeing these poison values in action a bit later in this topic, *Figure 6.4*).

Our `ch5/kmembugs_test.c` code has the function `umr_slub()`; it performs a `kmalloc()` for 32 bytes and then reads the just allocated memory to test **uninitialized memory reads (UMR)** on slab (SLUB) memory; here's the output seen when we run this test case:

Clearly, the uninitialized memory region (all 32 bytes of it) shows up as the value `0x6b`. Moreover, the last valid byte is set to `0xa5`, the value `POISON_END`, as expected.

## Passing the SLUB debug flags

Any and all SLUB debug flags – those seen in the table above – can be passed to a kernel configured with `CONFIG_SLUB_DEBUG=y` (as ours are, both the custom production and debug kernels) via the `slub_debug` kernel parameter. The format is as follows:

```
slub_debug=<flag1flag2...>,<slab1>,<slab2>,...
```

As can be seen, you can pass various flags (don't leave any space, just concatenate them together; to set all flags on, set it to `NULL`, to turn all off, set it to `-`). Any combination is possible; for example, passing the kernel parameter `slub_debug=FZPU`, enables, for all slab cache memory, the following SLUB features:

- Sanity checks ( F )
- Red zoning ( Z )
- Poisoning ( P )
- User tracking ( U )

Confirm this after boot with:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.10.60-dbg02-gcc root=UUID=<...> ro quiet
```

This is also reflected within the sysfs entry for the slab cache(s); let's look up the slab cache `kmalloc-32`, which of course provides generic 32-byte memory fragments to any requestor, as an example:

```
$ export SLAB=/sys/kernel/slab/kmalloc-32
$ sudo cat ${SLAB}/sanity_checks ${SLAB}/red_zone ${SLAB}/poison ${:
1
1
1
1
$
```

They're all set to `1`, indicating they're all on (the default is typically `0`, off).

All right, no dawdling, let's run our (relevant) test cases to see where the kernel's SLUB debug infrastructure can help us.

## Running and tabulating the SLUB debug test cases

All test cases are in the (same) module here:

`ch5/kmembugs_test/kmembugs_test.c` (as well as the companion `debugfs_kmembugs.c`). Here, as we're testing SLUB debug, we only run the testcases that pertain to slab memory. We'll test on our custom production kernel as well the distro kernel itself; why? This is as most distros (including the one I'm using here, Ubuntu 20.04 LTS) configures the kernel with `CONFIG_SLUB_DEBUG=y`; this is also the default choice within the `init/Kconfig` file where it's defined. (Another reason we don't test with our debug kernel is obvious – with KASAN and UBSAN turned on, they tend to first catch the bugs).

Importantly, to test, we'll boot the system with passing the kernel parameter as:

- `slub_debug=-` ; implying it's off
- `slub_debug=FZPU` ; implying these four flags and the SLUB debug features pertaining to them are turned on for *all slabs* on the system.

Then run the relevant testcases via our `test_kmembugs.ko` kernel module and associated `run_tests` script for each of the above scenarios. The following table summarizes the results!

| <b>Testcase #</b>                                     | <b>Memory defect type<br/>(below) / Infrastructure<br/>used (right)</b> | <b>Production<br/>kernel with<br/><code>slub_debug=-</code><br/>(off)</b> | <b>Production kernel<br/>with<br/><code>slub_debug=FZPU</code></b> |
|-------------------------------------------------------|-------------------------------------------------------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------|
| 5                                                     | OOB accesses on dynamic kmalloc-ed slab (SLUB) memory                   |                                                                           |                                                                    |
| 5.1                                                   | Read (right) overflow                                                   | N [V1]                                                                    | N                                                                  |
| 5.2                                                   | Write (right) overflow                                                  | Y [V4]                                                                    |                                                                    |
| 5.3                                                   | Read (left) underflow                                                   | N                                                                         |                                                                    |
| 5.4                                                   | Write (left) underflow                                                  | Y [V4]                                                                    |                                                                    |
| <b>Other<br/>memory<br/>corruption<br/>test cases</b> |                                                                         |                                                                           |                                                                    |
| 6                                                     | Use After Free - UAF                                                    | N                                                                         | Y [V5]                                                             |
| 7                                                     | Double-free                                                             | N [V2]                                                                    | Y [V6]                                                             |

|    |                                                               |        |        |
|----|---------------------------------------------------------------|--------|--------|
| 9  | OOB on<br>copy_[to from]_user*()                              | N      | N      |
| 10 | Uninitialized Memory<br>Read – UMR – on slab<br>(SLUB) memory | N [V3] | N [V7] |

Table 6.2 – Summary of findings when running relevant memory defect testcases against both our production kernel without slub\_debug features and with slub\_debug=FZPU

(As mentioned earlier, don't forget to look up *Table 6.4*, effectively, an all-in-one-place comparison table).

Okay, now let's dive into the details of running our test cases seen in the table above.

Test environment: x86\_64 guest running Ubuntu 20.04 LTS with our custom 5.10.60-prod01 production kernel (configured as mentioned just above).

### Running SLUB debug test cases on a kernel with slub\_debug turned off

First let's look at what occurs when we run our testcases without SLUB debug features enabled (corresponding to column 3 and points [V1], [V2] and [V3] in *Table 6.2*):

- No memory bugs are caught when `slub_debug=-`, I.e., is off (FYI, our first three test cases – the UMR, UAR and memory leakage – fail to be detected as well).
- [V1] the system could simply Oops or hang here or even appear to be remain unscathed; but that's not really the case... once the kernel is buggy, the system is buggy
- [V2] a segfault might occur on the double-free defect; the vanilla or production 5.x kernel indicates it like this:

```
kernel BUG at mm/slub.c:305!
```

With the instruction pointer register (RIP on the x86\_64) pointing at the `kfree()`. The report of course has the usual – the process context the bug occurred in and the kernel call trace.

Interestingly, what's at line 305 in `mm/slub.c`? I checked on the mainline kernel version 5.10.60, here:

<https://elixir.bootlin.com/linux/v5.10.60/source/mm/slub.c>:



```
297 return freelist_ptr(s, p, freepointer_addr),
298 }
299
300 static inline void set_freepointer(struct kmem_cache *s, void *object, void *fp)
301 {
302 unsigned long freeptr_addr = (unsigned long) object + s->offset;
303
304 #ifdef CONFIG_SLAB_FREELIST_HARDENED
305 BUG_ON(object == fp); /* naive detection of double free or corruption */
306 #endif
307 *(void **)freeptr_addr = freeblock_ptr(s, fp, freeptr_addr);
308 }
309 }
310 }
```

Figure 6.1 –Partial screenshot of the excellent Bootlin kernel source browser

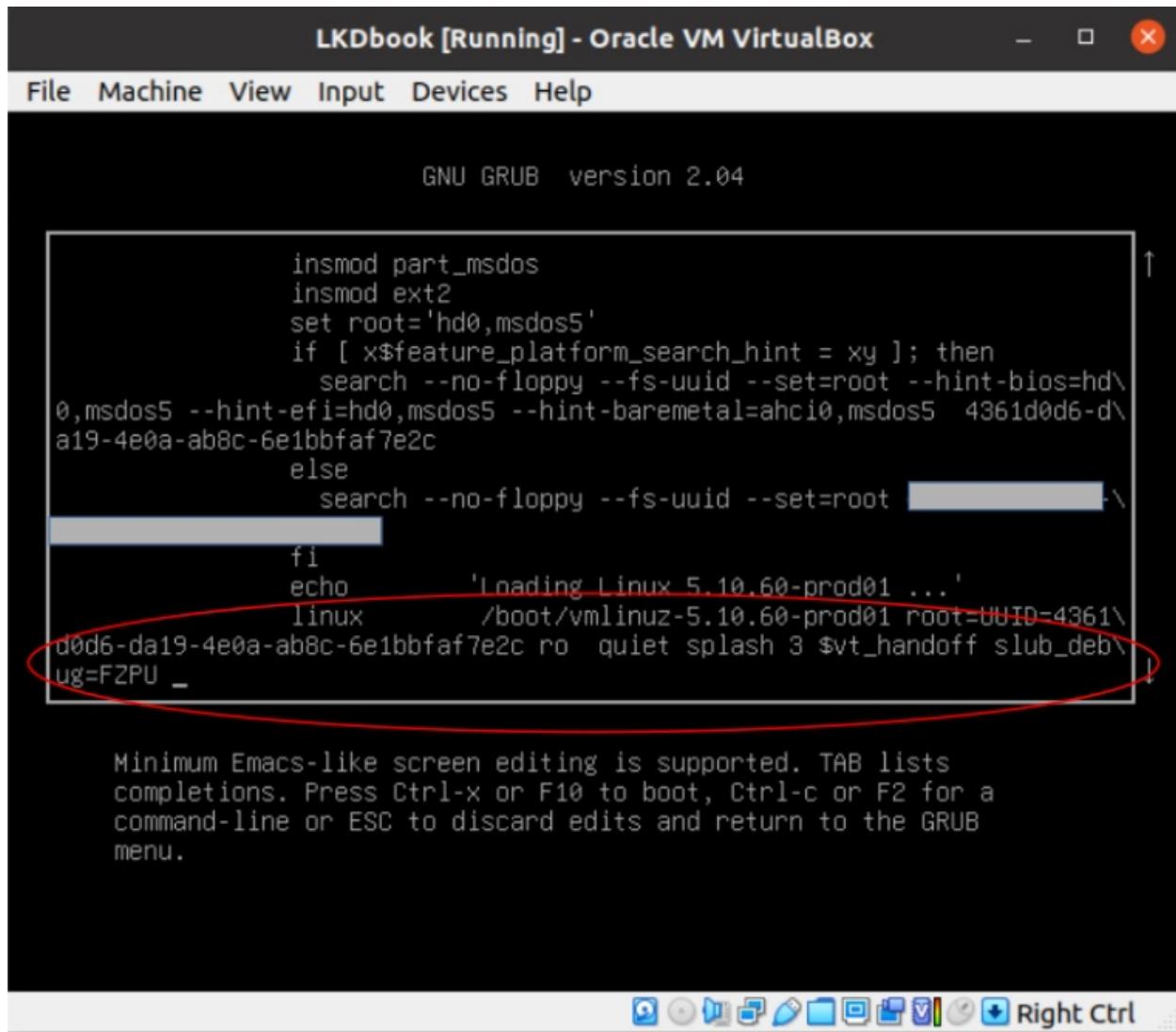
Above, you can see we're bang on target: line 305 is what triggered the double-free bug; the vanilla kernel has the intelligence to detect this (naïve) case of a double free, which is a form of memory corruption.

- [V3] The UMR on slab memory with our production kernel and no `slub_debug` set to - – implying it's off – isn't caught; the kmalloc-ed memory region appears to be initialized to 0x0 .

Okay, let's move along now to testing with the kernel SLUB debug feature(s) turned on.

Running SLUB debug test cases on a kernel with `slub_debug` turned on

Now, let's rerun our testcases, this time with the kernel's SLUB debug features enabled by passing along the `slub_debug=FZPU` kernel parameter. A screenshot showing our setting the kernel parameter `slub_debug=FZPU` in the GRUB bootloader on the production kernel (as seen on VirtualBox):



*Figure 6.2 –Screenshot showing the GRUB menu for editing the distro kernel parameters with the slub\_debug=FZPU highlighted*

Verify that the kernel command line we edited via the bootloader has made it intact:

```
$ dmesg |grep "Kernel command line"
[0.094445] Kernel command line: BOOT_IMAGE=/boot/vmlinuz-5.11.0-
```

It's fine; doing `cat /proc/cmdline` will reveal the same. We run our test cases again, this time with SLUB debug enabled; the results are as seen in *Table 6.2*, fourth column).

Refer back to *Table 6.2* seeing the places marked with [V4]; SLUB debug

catches both the write over and underflow (right and left) OOB accesses on slab memory. However, as we saw with UBSAN, it only seems able to catch it when the buggy access is via incorrect indices to the memory region, *not* when the OOB access is via a pointer! Also, the OOB reads do *not* seem to be caught.

Let's now learn a key skill: how to interpret the SLUB debug error report in detail.

## Interpreting the kernel's SLUB debug error report

Let's look in detail at catching a few of our buggy testcases; we load up and use our `run_tests` script to execute them.

### Interpreting the right OOB write overflow on slab memory

We begin with our test case #5.2. The right OOB access, here the write overflow (right) on the slab object (marked as [V4] in the table), has the kernel's SLUB debug framework leap into action and complain quite loudly as follows:

```
[620.764707] testcase to run: 5.2
[620.764760] =====
[620.764955] BUG kmalloc-32 (Tainted: G OE): Right Redzone overwritten
[620.765116] -----
[620.765370] Disabling lock debugging due to kernel taint
[620.765378] INFO: 0x0000000d0d6c75b-0x000000001b94c58a @offset=4640. First byte 0x78 instead of 0xcc
[620.765529] INFO: Allocated in dynamic_mem_oob_right+0x39/0x9c [test_kmembugs] age=0 cpu=5 pid=1697
[620.765659] _slab_alloc.isra.0+0x8b/0xf0
[620.765723] kmem_cache_alloc_trace+0x40b/0x450
[620.765791] dynamic_mem_oob_right+0x39/0x9c [test_kmembugs]
[620.765873] dbgfs run testcase+0x4d9/0x59a [test_kmembugs]
```

Figure 6.3 – Partial screenshot 1 of 3 showing SLUB debug catching the right OOB while writing

First off, following the word `BUG` is the name of the affected slab cache (here it's the `kmalloc-32` one, as our test case code performed a `kmalloc()` of, in fact, exactly 32 bytes).

Next, the kernel taint flags are followed by the issue at hand – the OOB access defect that caused the SLUB debug code to say `Right Redzone overwritten`; this is pretty self-explanatory, it's what actually did occur: within our `kmembugs_test.c`: `dynamic_mem_oob_right()` test case function, we did just this; performed a write at byte 32; the legal range is bytes 0 to 31 of course.

Next, the first `INFO` line spits out the start and end of the corrupted memory region (note that these kernel virtual addresses are hashed here, for security, preventing info leaks; recall, we ran this test case on our production kernel, after all).

Next, the second `INFO` line shows where the buggy access took place in the code - via the usual `<func>+0xs/0xy [modname]` notation. (Here it happens to be `dynamic_mem_oob_right+0x39/0x9c [test_kmembugs]`). This implies that the defect occurred in the function named `<func>` at an offset of `0xs` bytes from the start of the function, and the kernel estimates the function length to be `0xy` bytes. (In a later chapter we'll see how we can leverage this information!).

Further, the process context running – it's PID and the CPU core it ran upon, is displayed to the right.

This is followed by a (kernel-mode) stack trace:

```
kmem_cache_alloc_trace+0x40b/0x450
dynamic_mem_oob_right+0x39/0x9c [test_kmembugs]
dbgfs_run_testcase+0x4d9/0x59a [test_kmembugs]
full_proxy_write+0x5c/0x90
vfs_write+0xca/0x2c0
[...]
```

We haven't shown the full stack call trace here; read it bottom-up ignoring any lines that begin with a `? .` So, here, it's quite clear – the `dynamic_mem_oob_right()` function, located in the kernel module `test_kmembugs`, is where the trouble seems to be...

Next, the third `INFO` line provides information on which task performed the `free`; this can be useful, helping us identify the culprit as typically, the task that frees the slab is the one that allocated it in the first place! (This isn't the case in this particular run though):

```
INFO: Freed in vmw_marker_pull+0xaa/0x120 [vmwgfx] age=59 cpu=2 pid:
```

More information follows: a couple more `INFO` lines that display a few statistics on the slab and particular object within it that got corrupted, the content of the left and right redzones, any padding and the actual memory region content.

```

INFO: Slab 0x00000000d91ecea2 objects=19 used=5 fp=0x000000004fa4eb9d flags=0xfffffc0010201
INFO: Object 0x000000006489b63a @offset=4608 fp=0x0000000000000000

Redzone 000000003f2fee70: cc
Redzone 00000000da09c2a2: cc
Object 000000006489b63a: 6b kkkkkkkkkkkkkkkkk
Object 00000000bb2f628f: 6b a5 kkkkkkkkkkkkkkk
Redzone 00000000d0d6c75b: 78 cc cc 78 cc cc cc cc x..x....
Padding 000000008b49804: 5a ZZZZZZZZZZZZZZ
Padding 000000003a984ce1: 5a ZZZZZZZZZZZZZZ

```

*Figure 6.4 –Partial screenshot 2 of 3 of the SLUB debug interpretation of corrupted slab memory, the redzones, object memory and padding with faulty writes circled*

Take a look at a snippet of our buggy test case code, the one that ran here:

```

// ch5/kmembugs_test.c
int dynamic_mem_oob_right(int mode)
{
 volatile char *kptr, ch = 0;
 char *volatile ptr;
 size_t sz = 32;
 kptr = (char *)kmalloc(sz, GFP_KERNEL);
 [...]
 ptr = (char *)kptr + sz + 3; // right OOB
 [...]
} else if (mode == WRITE) {
/* Interesting: this OOB access isn't caught by UBSAN but is caught
*(volatile char *)ptr = 'x'; // invalid, OOB right write
/* ... but these below OOB accesses are caught by KASAN/UBSAN. We can
kptr[sz] = 'x'; // invalid, OOB right write
}

```

As you can see highlighted, in two places we (deliberately) perform an invalid right OOB access – writing the character `x`. Both are caught by the kernel SLUB debug infrastructure!

Do notice in Figure 6.4 above: the value `0x78` is our `x` character being (wrongly) written by the test case code – I've circled the incorrect writes in the figure above! Next, the poison values being used if the poison flag (`P`) is set, as is the case here, for the slab. Here, the poison value `0x6b` denoting UAF poisoning, `0xa5` denoting the end poisoning marker byte, and `0x5a` denoting use-uninitialized poisoning; useful indeed.

Further, the typical output when most kinds of kernel bugs occur now follows: a

detailed Call Trace (the kernel mode stack being unwound, read it bottom-up, ignoring lines that begin with ? ), and some of the CPU registers and their values:

```
CPU: 5 PID: 1697 Comm: run_tests Tainted: G B OE 5.10.60-prod01 #6
Hardware name: innoteck GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
Call Trace:
dump_stack+0x76/0x94
print_trailer+0x1de/0x1eb
check_bytes_and_report.cold+0x6c/0x8c
check_object+0x1c4/0x280
free_debug_processing+0x165/0x2a0
? dynamic_mem_oob_right+0x63/0x9c [test_kmembugs]
 _slab_free+0x2e3/0x4a0
? vprintk_func+0x61/0x1b0
? _raw_spin_unlock_irqrestore+0x24/0x40
kfree+0x4d8/0x500
? kmem_cache_alloc_trace+0x40b/0x450
? dynamic_mem_oob_right+0x63/0x9c [test_kmembugs]
dynamic_mem_oob_right+0x63/0x9c [test_kmembugs]
dbgfs_run testcase+0x4d9/0x59a [test_kmembugs]
full_proxy_write+0x5c/0x90
vfs_write+0xca/0x2c0
ksys_write+0x67/0xe0
_x64_sys_write+0x1a/0x20
do_syscall_64+0x38/0x90
entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP: 0033:0x72d33f4d31e7
Code: 64 89 02 48 c7 c0 ff ff ff eb bb 0f 1f 80 00 00 00 00 f3 0f 1e fa 64 8b 04 25 18 00
5 10 b8 01 00 00 00 0f 05 <48> 3d 00 f0 ff ff 77 51 c3 48 83 ec 28 48 89 54 24 18 48 89 74 24
RSP: 002b:00007ffdc666efd8 EFLAGS: 00000246 ORIG_RAX: 0000000000000001
RAX: ffffffffffffffd8 RBX: 0000000000000004 RCX: 000072d33f4d31e7
RDX: 0000000000000004 RSI: 0000558b9cde24e0 RDI: 0000000000000001
RBP: 0000558b9cde24e0 R08: 000000000000000a R09: 0000000000000003
R10: 0000558b9b2c4017 R11: 0000000000000246 R12: 0000000000000004
R13: 000072d33f5ae6a0 R14: 000072d33f5af4a0 R15: 000072d33f5ae8a0
FIX kmalloc-32: Restoring 0x000000000d0d6c75b-0x000000001b94c58a=0xcc

FIX kmalloc-32: Object at 0x0000000006489b63a not freed
```

*Figure 6.5 –Partial screenshot 3 of 3 of the SLUB debug error report (continued) showing the process context, hardware, kernel stack trace, CPU register values and FIX info*

Finally, the kernel SLUB debug framework even informs us as to what it restores and what's to fix; see the last two lines of the above screenshot (beginning with FIX kmalloc-32: ). With the F flag – the SLUB *sanity* checks feature – being enabled, the kernel attempts to clean up the mess and restore the slab object state to what would be deemed the correct form; of course, this may not always be possible to do. Also, this SLUB debug error report is generated before the slab object in question has been freed, hence the above ... not freed message (our code does free it).

(For more, do refer the official kernel documentation here: *SLUB Debug output*: <https://www.kernel.org/doc/html/latest/vm/slub.html#slub-debug-output>).

## Interpreting the UAF bug on slab memory

Now let's interpret the **Use After Free (UAF)** bug being caught (marked as [V5] in *Table 6.2*). The UAF bug is caught by the slub debug framework; the error report (within the syslog) looks like this:

```
BUG kmalloc-32 (Tainted: G B OE): Poison overwritten
[3747.701588] -----
[3747.707061] INFO: 0x0000000d969b0bf-0x0000000d969b0bf @offset=8
[3747.710110] INFO: Allocated in uaf+0x20/0x47 [test_kmembugs] age=
```

The format remains the same as described just above; this time, the UAF defect caused the SLUB debug code to say `Poison overwritten`; why? In our `uaf()` test case, we did just this; freed the slab object and then performed a write to a byte within it!

Next, the `INFO` line spits out the start and end of the corrupted memory region (note that these kernel virtual addresses are hashed here, for security, preventing info-leaks; recall, we ran this test case on our production kernel, after all).

The PID of the task performing the allocation (the process context) is seen – along with the kernel module name in square brackets, if applicable, as well as the function within it where the alloc took place.

This is followed by a stack trace (we don't show this here) and then information on which task performed the free; this can be useful, helping us identify the culprit as typically, the task that frees the slab is the one that allocated it in the first place!

```
INFO: Freed in uaf+0x34/0x47 [test_kmembugs] age=5 cpu=5 pid=2306
```

Moving along to the next test case, the double-free...

## Interpreting the double-free on slab memory

Finally, a quick note on the double-free defect, successfully caught again (marked by [V6] in *Table 6.2*): here, the kernel reports it as follows:

```
BUG kmalloc-32 (Tainted: G B 0E): Object already free
[3997.543154] -----
[3997.544129] INFO: Allocated in double_free+0x20/0x4b [test_kmemleak]
```

The very same template as described above follows this output... do try it out for yourself, read and interpret it.

By the way, we've already seen how the SLUB debug framework deals with uninitialized memory reads (the UMR defect) on slab cache memory (our testcase #10, marked as [V3] and [V7] in the table): when run with `slub_debug` on, though no error report is generated, a dump of the memory region shows the poison value `0x6b`, denoting use-after-free poisoning.

So, going by our experiments, while the kernel SLUB debug framework seems to catch most of the memory corruption issues on slab memory, it doesn't seem to catch the *read* OOB accesses on slab memory. Note that KASAN does (see Table 6.2)!

## Learning to use the slabinfo and related utilities

Another utility program that can prove to be very useful for understanding and helping debug the slab caches is one named `slabinfo`. Though a user mode app, its code is in fact part of the kernel source tree, here:

```
tools/vm/slabinfo.c
```

Build it by simply changing directory to the `tools/vm` folder within your kernel source tree and typing `make`. Running it does require root access. (Once built, for convenience, I like to create a soft or symbolic link to the binary executable named `/usr/bin/slabinfo`). Here it is on my system:

```
$ ls -l $(which slabinfo)
lrwxrwxrwx 1 root root 71 Nov 20 16:26 /usr/bin/slabinfo -> <...>/1:
```

Display its help screen by passing the `-h` (or—`help`) option switch; this reveals a large number of possible option switches. The screenshot below shows all (for the 5.10.60 kernel):

```
$ sudo slabinfo --help
slabinfo 4/15/2011. (c) 2007 sgi/(c) 2011 Linux Foundation.

slabinfo [-aABDefhillNnoPrsStTUvXz1] [N=K] [-dafzput] [slab-regexp]
-a|--aliases Show aliases
-A|--activity Most active slabs first
-B|--Bytes Show size in bytes
-D|--display-active Switch line format to activity
-e|--empty Show empty slabs
-f|--first-alias Show first alias
-h|--help Show usage information
-i|--inverted Inverted list
-l|--slabs Show slabs
-L|--Loss Sort by loss
-n|--numa Show NUMA information
-N|--lines=K Show the first K slabs
-o|--ops Show kmem_cache_ops
-P|--partial Sort by number of partial slabs
-r|--report Detailed report on single slabs
-s|--shrink Shrink slabs
-S|--Size Sort by size
-t|--tracking Show alloc/free information
-T|--Totals Show summary information
-U|--Unreclaim Show unreclaimable slabs only
-v|--validate Validate slabs
-X|--Xtotals Show extended summary information
-z|--zero Include empty slabs
-1|--1ref Single reference

-d | --debug Switch off all debug options
-da | --debug=a Switch on all debug options (--debug=FZPU)

-d[afzput] | --debug=[afzput]
 f | F Sanity Checks (SLAB_CONSISTENCY_CHECKS)
 z | Z Redzoning
 p | P Poisoning
 u | U Tracking
 t | T Tracing

Sorting options (--Loss, --Size, --Partial) are mutually exclusive
$
```

*Figure 6.6 – Screenshot showing the help screen of the kernel slabinfo utility*

A few things to note when running `slabinfo`:

- By default, this tool will only display slabs that have data within them (the same as with the `-l` switch, in effect. You can change this by running `slabinfo -e`; it displays only the empty caches; there can be quite a few!)
- All options may not work straight away; most require that the kernel is compiled with SLUB debug on (`CONFIG_SLUB_DEBUG=y`; typically, this is the case, even for distro kernels). Some options require the SLUB flags be passed on the kernel command line (via the usual `slob_debug` parameter)
- Need to run it as root.

Let's begin by doing a quick run (with no parameters), seeing the header line and a line of sample output (that of the `kmalloc-32` slabcache; I've spaced the lines to fit):

```
$ sudo slabinfo |head -n1
Name Objects Objsize Space Slabs/Part/Cpu O/S 0 %Fr %Ef Flg
[...]
kmalloc-32 35072 32 1.1M 224/0/50 128 0 0 100
$
```

A quick roundup of the header line columns is as follows:

- the name of the slab cache ( `Name` )
- the current number of objects currently allocated ( `Objects` )
- the size of each object is then shown (here, its 32 bytes of course) ( `Objsize` )
- the total space taken up in kernel memory by these objects (essentially, it's `Objects * Objsize`) ( `Space` )
- the slab cache memory distribution for this cache: number of full slabs, partial slabs and per-cpu slabs ( `Slabs/Part/Cpu` )
- the number of objects per slab ( `O/S` )
- the order of the page allocator from where memory is carved out for this cache (the order is from `0` to `MAX_ORDER`, typically it's the value 11, to give us a total of 12 orders; `2^order` is the size of free memory chunks on the page allocator freelist for that order) ( `0` )
- the amount of cache memory free in percentage terms ( `%Fr` )
- the effective memory usage as a percentage ( `%Ef` )

- the slab flags (can be empty) ( Flg ).

(Want to see the actual `printf()` that emits these slab cache stats? It's right here: <https://elixir.bootlin.com/linux/v5.10.60/source/tools/vm/slabinfo.c#L640>).

All possible values for the flags and their meaning are as follows (pertains to the column named `Flg` on the extreme right of `slabinfo` normal output):

- \* : aliases present
- d : for DMA memory slabs
- A : hardware cacheline (hwcache) aligned
- P : slab is poisoned
- a : reclaim accounting active
- z : slab is red zoned
- F : slab has sanity checking on
- U : slab stores user
- T : slab is being traced

Note that the columns change when the `-D` (display active) option switch is passed.

**A common FAQ, perhaps:** of the many slab caches that are currently allocated (and have some data content), *which takes up the most kernel memory?* This is easily answered by `slabinfo`: one way, run it with the `-B` switch, to display the space taken in bytes, allowing one to easily sort on this column. Even simpler, the `-S` option switch has `slabinfo` sort the slab caches by size (highest first) with nice human-readable size units displayed. The screenshot below shows our doing so, for the top ten highest kernel memory consuming slab caches:

| Name              | Objects | Objsize | Space | Slabs/Part/Cpu | 0/S | 0 | %Fr | %Ef | Flg |
|-------------------|---------|---------|-------|----------------|-----|---|-----|-----|-----|
| inode_cache       | 24726   | 600     | 15.5M | 912/0/39       | 26  | 2 | 0   | 95  | a   |
| buffer_head       | 132015  | 104     | 13.8M | 3369/0/16      | 39  | 0 | 0   | 99  | a   |
| ext4_inode_cache  | 7074    | 1176    | 8.5M  | 252/0/10       | 27  | 3 | 0   | 96  | a   |
| dentry            | 40572   | 192     | 7.9M  | 1883/0/49      | 21  | 0 | 0   | 98  | a   |
| kmalloc-4k        | 1591    | 4096    | 6.5M  | 189/8/12       | 8   | 3 | 3   | 98  |     |
| radix_tree_node   | 8603    | 576     | 5.0M  | 298/7/13       | 28  | 2 | 2   | 97  | a   |
| kernfs_node_cache | 30144   | 128     | 3.8M  | 899/0/43       | 32  | 0 | 0   | 100 |     |
| kmalloc-512       | 5040    | 512     | 2.5M  | 282/0/33       | 16  | 1 | 0   | 100 |     |
| filp              | 8816    | 256     | 2.2M  | 501/0/51       | 16  | 0 | 0   | 99  | A   |

Figure 6.7 – Screenshot showing the top ten slab caches sorted by total kernel

*memory space taken (4th column)*

Interestingly (as often happens with software), the

-U ‘Show unreclaimable slabs only’ option to `slabinfo` came into being due to a system getting panicked. This occurred when the *unreclaimable* slab memory usage went to close to a 100% and the **Out Of Memory (OOM)** killer was unable to find any candidate to kill! The patch has the utility – as well as the OOM kill code paths – display all unreclaimable slabs, to help with troubleshooting. (This patch got mainlined in the 4.15 kernel; here’s the commit, do take a peek at it:

<https://github.com/torvalds/linux/commit/7ad3f188aac15772c97523dc4ca3e8e5b>

Along with the -U switch, the -S option (sort by size), makes troubleshooting these corner cases easier!

The *sort-by-loss* (-L) option switch has `slabinfo` sort the slab caches by the amount of kernel memory lost. A better word perhaps, is *wasted*. This is the usual well-known *internal fragmentation* issue: when memory is allocated via the slab layer, it internally does so via a *best-fit* model; this often results in a small amount of memory being wasted or lost. For example, attempting to allocate 100 bytes via the `kmalloc()` will have the kernel actually allocate you memory from the `kmalloc-128` slab cache (as it can’t possibly give you less via `kmalloc-96` cache), with the result that your slab object actually consumes 128 bytes of kernel memory; thus, the loss or wastage in this case is 28 bytes. Thus, doing `sudo slabinfo -L | head`, will quickly show you (in descending order) the slabs with maximum wastage (or loss; look at the fourth column, labelled Loss ).

Once you’ve identified a slab cache that you’d like to further investigate, the -r (*report*) option will have `slabinfo` emit detailed statistics. By default, this is on all slabs; you can always pass a regular expression specifying which slabs you’re interested in! (For example, `sudo slabinfo -r vm.*` will display details on all slabs matching the regex pattern `vm.*` ).

Also, at times, you might see a slab cache with a name that’s unfamiliar; trying the -a (or— aliases ) option to show aliases can be useful to reveal what kernel object(s) it’s being used to cache.

The -T option has `slabinfo` display *overall totals*, a summary snapshot of all slab caches. This is useful to get a quick overview of how many slab caches exists, how many are active, how much kernel memory in all is being used, and

so on. This kind of information is *extended* when you use the `-X` option switch; it now shows even more detail. The screenshot below is an example of running `sudo slabinfo -X` on my x86\_64 Ubuntu guest:

```
$ sudo slabinfo -X
[sudo] password for letsdebug:
Slabcache Totals

Slabcaches : 216 Aliases : 0->0 Active: 133
Memory used: 90710016 # Loss : 2548968 MRatio: 2%
Objects : 401015 # PartObj: 1444 ORatio: 0%

Per Cache Average Min Max Total

#Objects 3015 10 132132 401015
#Slabs 88 1 3388 11833
#PartSlab 0 0 31 101
%PartSlab 0% 0% 38% 0%
PartObjs 0 0 670 1444
% PartObj 0% 0% 23% 0%
Memory 682030 4096 15581184 90710016
Used 662865 3072 14835600 88161048
Loss 19165 0 745584 2548968

Per Object Average Min Max

Memory 221 8 8192
User 219 8 8192
Loss 1 0 64

Slabs sorted by size

Name Objects Objsize Space Slabs/Part/Cpu O/S 0 %Fr %Ef Flg
inode_cache 24726 600 15581184 912/0/39 26 2 0 95 a

Slabs sorted by loss

Name Objects Objsize Loss Slabs/Part/Cpu O/S 0 %Fr %Ef Flg
inode_cache 24726 600 745584 912/0/39 26 2 0 95 a

Slabs sorted by number of partial slabs

Name Objects Objsize Space Slabs/Part/Cpu O/S 0 %Fr %Ef Flg
anon_vma 2970 80 331776 40/31/41 46 0 38 71

$
```

*Figure 6.8 – Screenshot showing extended summary information via slabinfo -X*

These can serve as useful diagnostics when troubleshooting a system (you'll find more in a similar vein in the later section *Practical stuff – who's eating my memory?*).

Running `slabinfo` with the `-z` (zero) option switch has it show all slab caches;

both the ones with data as well as the empty ones.

Debug-related options to slabinfo

For debug purposes, `slabinfo` has a `-d` and a `-v` option switch, allowing one to pass *debug flags* and *validate slabs*, respectively. Note that both these option switches will only work when system is booted with the `slub_debug` kernel parameter set to some non-NUL value.

It's interesting to see: when booted with `slub_debug=FZPU`, all the slab caches show up with (at least) these flags set!

| Name              | Objects | Objsize | Space | Slabs/Part/Cpu | 0/S | 0 | %Fr | %Ef | Flg   |
|-------------------|---------|---------|-------|----------------|-----|---|-----|-----|-------|
| inode_cache       | 24162   | 600     | 23.3M | 1425/4/0       | 17  | 2 | 0   | 62  | PaZFU |
| kmalloc-4k        | 1255    | 4096    | 20.7M | 634/13/0       | 2   | 3 | 2   | 24  | PZFU  |
| dentry            | 35230   | 192     | 18.6M | 1137/1/0       | 31  | 2 | 0   | 36  | PaZFU |
| kernfs_node_cache | 26301   | 128     | 12.7M | 1558/25/0      | 17  | 1 | 1   | 26  | PZFU  |
| ext4_inode_cache  | 4949    | 1176    | 7.7M  | 237/4/0        | 21  | 3 | 1   | 74  | PaZFU |
| kmalloc-32        | 16029   | 32      | 7.0M  | 856/78/0       | 19  | 1 | 9   | 7   | PZFU  |
| radix_tree_node   | 5192    | 576     | 5.0M  | 307/4/0        | 17  | 2 | 1   | 59  | PaZFU |
| buffer_head       | 10231   | 104     | 4.6M  | 570/6/0        | 18  | 1 | 1   | 22  | PaZFU |
| kmalloc-1k        | 1262    | 1024    | 4.2M  | 130/21/0       | 10  | 3 | 16  | 30  | PZFU  |

Figure 6.9 – Partial screenshot, focus is on the SLUB debug flags being set as we booted with `slub_debug=FZPU`

Notice how, for all slabs, the flags minimally contain `PZFU`.

Regarding the `-d` option switch, passing it by itself turns debugging *off* (quite non-intuitively, no? Then, again, this is consistent with the way the kernel parameter `slub_debug` behaves). When you want the kernel's SLUB debugging options on, pass along the usual SLUB debug flags like so:

`--debug=<flag1flag2...>`. The `slabinfo` help screen shows all of this clearly; look up *Figure 6.6* above, specifically the last few lines, the ones that describe the `--debug` option switch.

What really happens under the hood when you do pass, say, `--debug=fzput` (or, if a is passed, *all* these SLUB debug flags are set), as a parameter to `slabinfo`, is this: the utility opens (as root of course) the underlying `/sys/kernel/slab/<slabname>` pseudo file for that slab cache (if you passed one or more of them as a parameter), else for all slabs, and arranges to set these

to 1 , meaning, enabled:

- If f | F passed in— debug=<...> ,  
/sys/kernel/slab/<slabname>/sanity\_checks to 1
- If z | Z passed in— debug=<...> ,  
/sys/kernel/slab/<slabname>/red\_zone to 1
- [... similarly, for the rest ...]

(FYI, the code that does this is here:

<https://elixir.bootlin.com/linux/v5.10.60/source/tools/vm/slabinfoc#L717>).

The -v option switch to slabinfo can again be useful for debugging: it *validates all slabs*, and on any errors being detected, it spews out diagnostics / error reports to the kernel log. The format of the report is in fact identical to the error report format that the kernel’s SLUB debug infrastructure produces (we covered this in details here: *Interpreting the kernel’s SLUB debug error report*).

As with the debug option switch, the -v causes slabinfo to write 1 into the pseudo file /sys/kernel/slab/<slabcache>/validate . The kernel documents this as follows: *Writing to the validate file causes SLUB to traverse all of its cache’s objects and check the validity of metadata*. All slab objects will be checked; the detailed output is written to the kernel log. This can be useful when troubleshooting a live system that you suspect might suffer from slab (SLUB) memory corruption.

Finally, we’ll just mention the fact that there’s even a utility script, slabinfo-gnuplot.sh , to plot graphs to help visualize slab (SLUB) functioning over time! I’ll leave it to you to browse the kernel documentation that explains how to leverage it, here:

<https://www.kernel.org/doc/Documentation/vm/slub.txt>, under the section named *Extended slabinfo mode and plotting*.

## The /proc/slabinf o pseudo file

Also, the kernel of course exposes all this useful information on live slabs on the system via procfs, particularly, the pseudo file /proc/slabinf o (again, you’ll require root access to view it). Here’s a sampling of the large data available (internally, for each slab, it breaks the data into three types: statistics , tunables , slabdata ). First, the header shows the version number and the

columns:

```
$ sudo head -n2 /proc/slabinfo
slabinfo - version: 2.1
name <active_objs> <num_objs> <objsize> <objperslab> <i
```

And here's some data from it:

```
$ sudo grep -C2 "kmalloc-128" /proc/slabinfo
kmalloc-256 1982 2448 512 16 2 : tunables 0 0 0 : slabdata 153 153 0
kmalloc-192 3424 3424 256 16 1 : tunables 0 0 0 : slabdata 214 214 0
kmalloc-128 1968 1968 256 16 1 : tunables 0 0 0 : slabdata 123 123 0
kmalloc-96 1956 2368 128 32 1 : tunables 0 0 0 : slabdata 74 74 0
kmalloc-64 6907 8096 128 32 1 : tunables 0 0 0 : slabdata 253 253 0
$
```

Figure 6.10 – A screenshot showing some data from /proc/slabinfo

The man page on `slabinfo(5)` covers interpreting this data, in effect, all slab caches exposed via `/proc/slabinfo`; we'll leave it to you to check it out. (Unfortunately, it seems a bit dated in the sense that both the `statistics` and `tunables` information – the first two columns – pertain to only the older SLAB implementation; that's why we've highlighted the `slabdata` column headers).

By the way, the `vmstat` utility too has the ability to display the kernel slab caches and some statistics (via its `-m` option switch). It essentially does so by reading `/proc/slabinfo`, and thus implies you must run it as root. Try this on your box:

```
sudo vmstat -m
```

Check out the man page on `vmstat(8)` for more information.

## The slabtop utility

As with `top` (and the more modern `htop`) to see who's consuming CPU in real time, we have the `slabtop(1)` utility to see *live real time kernel slab cache usage*, sorted by the maximum number of slab objects by default (the `sort` field can always be changed via the `-s` or—`sort` option switch). It too, is based on data obtained from `/proc/slabinfo` and thus, as usual, you'll require root access to run it. Using `slabtop` you can see for yourself how, besides the caches for specific kernel data structures, the small sized generic caches (typically the ones named `kmalloc-*`) are often the ones being employed the

most. Do try it and check out its man page for details.

## eBPF's slabratetop utility

Finally, and a recent addition, is the *eBPF slabratetop* utility (it could be named *slabratetop-bpfcc*, as is the case on my system). It displays, in real time, the kernel's SLAB / SLUB memory cache allocation rate in a top-like manner, refreshing it once a second by default. It internally tracks the `kmem_cache_alloc()` API to track the rate and total bytes allocated via this commonly used interface to allocate slab objects within the kernel. Via option switches, you can control the output interval (in seconds) and the number of times to show it (along with a couple of other switches).

So, doing

```
sudo slabratetop-bpfcc 5 3
```

will have the utility display the active caches (allocation rate and number of bytes allocated) in 5 second interval summaries, thrice. Do refer to its man page and / or pass the `-h` option switch to see a brief help screen.

## Practical stuff – who's eating my memory?

So, knowing about these utilities, how can it practically help? Well, *one common case is needing to know who's eating up memory (RAM) and from where exactly is it being eaten up.*

The first question is very wide ranging; in terms of user mode processes and threads, utilities like `smon`, `ps`, and so on, can help. On a raw level, peeking at memory statistics under `procfs` can really help as well; for example, you can track memory usage of all threads by looking through `proc` with something like:

```
grep "^\w+.*:" /proc/*status
```

Within this output, the `VmRSS` number is a reasonable measure of physical memory usage (with the unit being kilobytes). So, doing:

```
grep -r "^\w+VmRSS" /proc/*status | sed 's/kB$//'| sort -t: -k3n | tail
```

Can quickly show you the PIDs of the top ten processes or threads consuming

the most RAM!

Here, rather than userspace, you're perhaps more interested in who or what is *eating kernel dynamic – slab cache – memory*, yes? Here's one investigative scenario:

- First, use `slabratetop` (or `slabratetop-bpfcc`) to figure which slab cache, among the many present within the kernel, is being consumed most
- Second, use *dynamic kprobes* to lookup the kernel-mode stack in real-time to see who or what within the kernel is eating into this cache!

We can easily achieve the first step like this:

```
sudo slabratetop-bpfcc
[...]
CACHE ALLOCS BYTES
names_cache 18 78336
vm_area_struct 176 46464
...
```

Okay, based on this sample output, there's a slab cache on my (guest) system named `names_cache` which is consuming the greatest number of bytes; as well, you can see that the `vm_area_struct` slab cache is currently seeing the greatest number of allocations, all within the given time interval (a second by default).

Let's say we want to dig deeper, investigate what code paths within the kernel are allocating memory from the `vm_area_struct` slab cache (pretty often, 176 times per second, as of right now). In other words, how can you figure who or what within the kernel is performing these allocations? Okay, let's see: we know that most specific slab cache objects are allocated via the `kmem_cache_alloc()` kernel interface. Thus, seeing the (kernel) stack of `kmem_cache_alloc()` in real time will help you pinpoint who or where the allocation is being performed from!

So how do we do that? That's the second part. Recall what you learned in the chapter on Kprobes, specifically using dynamic kprobes (we covered this in *Chapter 4, Debugging via instrumentation - Kprobes* section *Setting up a dynamic kprobe (via kprobe events) on any function*). Let's leverage that knowledge and look deeper into this; we'll begin by using `kprobe[-perf]()` command (bash script, really) to probe all running instances of the `kmem_cache_alloc()` API in real time and reveal the internal kernel mode stack

(by passing along the `-s` option switch):

```
sudo kprobe-perf -s 'p:kmem_cache_alloc name=+0(+96(%di)):string'
```

Also, do recall, from *Chapter 4, Debugging via instrumentation - Kprobes* section *Understanding the Basics of the Application Binary Interface (ABI)*, that on x86\_64, the `rdi` register holds the first parameter. Here, for the `kmem_cache_alloc()`, the first parameter is a pointer to `struct kmem_cache`; within this structure, at an offset of 96 bytes, is the thing we're after, the member named `name` – the name of the slab cache being allocated from!

Next point, this command above will probe for and show you all the slab cache allocations currently being performed by the popular `kmem_cache_alloc()` API; let's filter its output to see only the one of interest to us right now, the one for the `vm_area_struct` slab cache:

```
sudo kprobe-perf -s 'p:kmem_cache_alloc
name=+0(+96(%di)):string' | grep -A10
"name=.vm_area_struct"
```

A small portion of the output is seen in the screenshot below:

```
--> <...>-3154 [001] ...1 13294.620610: kmem_cache_alloc: (kmem_cache_alloc+0x0/0x8d0)
name="vm_area_struct"
<...>-3154 [001] ...1 13294.620616: <stack trace>
=> kmem_cache_alloc
=> do_brk_flags
=> __x64_sys_brk
=> do_syscall_64
=> entry_SYSCALL_64_after_hwframe
```

Figure 6.11 – Partial screenshot showing output from the `kprobe-perf` with the kernel-mode stack showing the lead up to the `kmem_cache_alloc()` of the VMA structure

(You might need to adjust the `grep -An` (I've kept `n` to `10` here) parameter to show a certain number of lines after the match). Quite clearly, this particular call chain shows us that the `kmem_cache_alloc()` has been invoked via a system call, the `sys_brk()`. FYI, this call is typically the one issued when a memory region of a process needs to be created, or an existing one grown or shrunk.

Now, internally, the kernel manages the memory regions (technically, the mappings) of a process via the **Virtual Memory Area (VMA)** metadata

structure; thus, when creating a new mapping of a process – as is the case here – the VMA object will naturally need to be allocated. As the VMA is a frequently used kernel structure, it's kept on a custom slab cache and allocated from it – *the one named vm\_area\_struct*! This is the call chain from above, that allocates a VMA object from this very slab cache:

```
sys_brk() --> do_brk() --> do_brk_flags() --> vm_area_alloc() --> km
```

(Here's the actual line of code where the `vm_area_alloc()` routine is invoked: <https://elixir.bootlin.com/linux/v5.10.60/source/mm/mmap.c#L3110>), which in turn issues the `kmem_cache_alloc()`, allocating an instance of a VMA object from its slab cache and then initializing it). Interesting.

### Security tip

Though unrelated to this coverage, I think security is important; to guarantee that slab memory is always wiped, both at allocation and free, pass these on the kernel command line:

`init_on_alloc=1 init_on_free=1`. More in a similar vein can be found here:

[https://kernsec.org/wiki/index.php/Kernel\\_Self\\_Protection\\_Project/Recommen](https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project/Recommen)

Good job on covering this section on SLUB debug and its offshoots. Now let's finish this large topic on kernel memory debugging by – finally! – learning how to catch those dangerous leakage bugs. Read on!

## Finding memory leakage issues with kmemleak

*What is a memory leak and why does it matter?* A memory leak is a situation where you have allocated memory dynamically but failed to free it. Well, you *think* you have succeeded in freeing it but the reality is that it hasn't been freed. The classic pedagogical case is a simple one like this (let's just make it a userspace example for simplicity):

```
static foo(void) {
 char *ptr = malloc(1024);
 /* ... work with it ... */
 // forget to free it
}
```

Now, once you return from the function `foo()`, it's basically impossible to free the memory pointed to by the variable `ptr`. Why? You know it, `ptr` is a local variable and is out-of-scope once you've returned from `foo()`. Now, the 1024 bytes of allocated memory is, in effect, locked up, inaccessible, wasted; we call this a **leak**. (Of course, once the user process dies, it's freed back to the system).

## Catching memory leaks in userspace apps

This book focusses purely on kernel debugging. Userspace app memory issues and debugging them has been covered in a lot of detail in my earlier book, *Hands-on System Programming with Linux*, Packt, Oct 2018 (see Chapters 5 and 6).

This can certainly happen within the kernel too (just substitute the user code above for a kernel module and the `malloc()` with a `kmalloc()`, or similar API!). Even a small leakage of a few bytes can become a huge issue when the code that causes the leak runs often (in a loop, perhaps). Not just that, unlike userspace, the kernel isn't expected to die... the leaked memory is thus lost forever (yes, even when a module has a leak and is unloaded, the dynamic kernel memory allocated isn't within the module, it's typically slab or page allocator memory!).

Now, you might say *Hey, I know, but come on, if I allocate memory, I'll certainly free it.* True, but when the project is large and complex, believe me, you can miss it. As a simple example, check out this pseudo code snippet:

```
static int kfoo_leaky(void) {
 char *ptr1 = kmalloc(GFP_KERNEL, 1024), *ptr2;
 /* ... work with ptr1 ... */
 // ...
 if (bar() < 0)
 return -EINVAL;
 // ...
 ptr2 = vmalloc(5120); [...]
 // ... work with ptr2 ...
 if (kbar() < 0)
 return -EIO;
 // ...
 vfree(ptr2);
 kfree(ptr1);
 return 0;
}
```

You see it don't you? At both the error return callsites before the final one, we've returned *without freeing* the memory buffers previously allocated; classic memory leaks! This kind of thing is in fact a pretty common pattern, so much so, that the kernel community has evolved a set of helpful *coding style guidelines*, one of which will certainly have helped avoid a disaster of this sort: using the (controversial) goto to perform clean-up (like freeing memory buffers!) before returning. Don't knock it till you try it – it works really well when used correctly. (This is formally called *Centralized exiting of functions*; read all about it here: <https://www.kernel.org/doc/html/latest/process/coding-style.html#centralized-exiting-of-functions>).

Here's a fix (via the *centralized exiting of functions* route):

```
static int kfoo(void) {
 char *ptr1 = kmalloc(GFP_KERNEL, 1024), *ptr2;
 int ret = 0;
 /* ... work with ptr1 ... */
 // ...
 if (bar() < 0) {
 ret = -EINVAL;
 goto bar_failed;
 }
 // ...
 ptr2 = vmalloc(5120); [...]
 // ... work with ptr2 ...
 if (kbar() < 0)
 ret = -EIO;
 goto kbar_failed;
}
// ...
kbar_failed:
vfree(ptr2);
bar_failed:
kfree(ptr1);
return ret;
}
```

Quite elegant, yes? The later you fail, the higher, the earlier (in the code) you jump, so as to perform all the required clean-up. So here, if the function `kbar()` fails, it goes to the `kbar_failed` label, performing the `vfree()` and then neatly falls through performing the `kfree()` as well. I'm betting you've seen code like this all over the kernel; it's a very common and useful technique. (Again, a random example: see the code of a function belonging to the Cadence

MACB/GEM Ethernet Controller network driver here:

<https://elixir.bootlin.com/linux/v5.10.60/source/drivers/net/ethernet/cadence/mac>

Another common root cause of leakage bugs: an interface is designed in such a way that it allocates memory, usually to a pointer passed by reference to it as a parameter. It's often deliberately *designed such that the caller is responsible for freeing the memory buffer* (after using it). But what happens when the caller fails to do so? A leakage bug of course! Typically, this will be well documented, but then, who reads documentation... (hey, that means read it!).

Can I see some real kernel memory leakage bugs?

Sure. First, head over to the kernel.org Bugzilla site <https://bugzilla.kernel.org/>. Go to the **Search | Advanced Search** tab there. Fill in some search criteria – in the **Summary:** tab, perhaps you'd want to type in something (like `memory leak`). You can filter down by **Product** (or subsystem), **Component** (within **Product**), even **Status** and **Resolution**! Then, click on the **Search** button. A screenshot of the search screen for a sample search I did, for your reference:

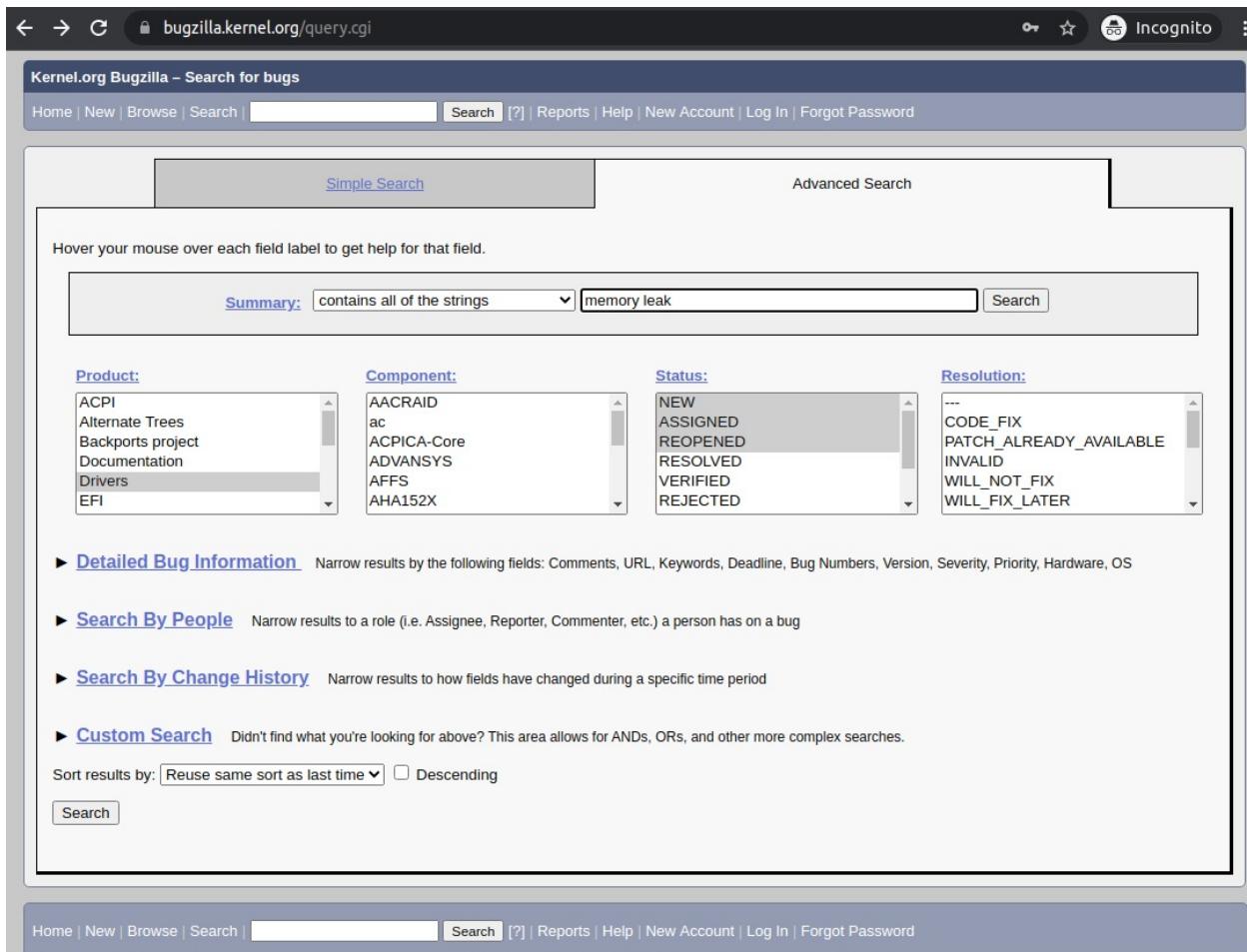


Figure 6.12 – Screenshot of the kernel.org Bugzilla, searching for 'memory leak' within bugs submitted under Drivers

(This particular search, at the time I made it, yielded 11 results).

Again, as an example, here's one memory leak bug report from the kernel *Bugzilla: backport-iwlwifi: memory leak when changing channels* ([https://bugzilla.kernel.org/show\\_bug.cgi?id=206811](https://bugzilla.kernel.org/show_bug.cgi?id=206811)). Look into it; download the attachment labelled dmesg.log after reboot ; the kernel log output shows that none other than kmemleak detected the leakage! (We cover the interpretation of the report in the section *Interpreting kmemleak's report* below).

**The real challenge is this:** without explicitly employing powerful tools – like **kmemleak** for the kernel (for userspace, there's ASAN, MSAN, Valgrind's memcheck, and so on) – memory leaks often go unnoticed – in development, even in test and in the field. But when they strike one fine day, the symptoms

can appear random – the system might run perfectly for a long while (even several months), when all of a sudden, it experiences random failures or even abruptly crashes. Debugging such situations can, at times, be next to impossible – the team blames power variances/outages, lightning, anything convenient to explain away the inexplicable random crash! (How often have you heard support say *Just reboot and try it again, it'll likely work*. And, the unfortunate thing is that it often does; thus, the real issue is glossed over). Unfortunately, this will simply not do on mission critical projects or products, it will eventually cause customer confidence to deteriorate and can ultimately result in failure.

It's a serious problem. Don't let it become one on your project! Take the trouble to perform long-spanning coverage testing (for long durations, a week or more).

## Configuring the kernel for kmemleak

Before going further, do realize that *kmemleak* is, again, not a magic bullet: it's designed to track and catch memory leakage for dynamic kernel memory allocations performed via the `kmalloc()`, `vmalloc()`, `kmem_cache_alloc()` and friends APIs *only*. These being the interfaces via which memory is typically allocated, it does serve an extremely useful purpose.

The key kernel config option, the one we need to enable, is `CONFIG_DEBUG_KMEMLEAK=y` (several related ones are mentioned below as well). Of course, there's the usual trade-off: being able to catch leakage bugs is a tremendous thing, but can cause pretty high overhead on memory allocations and freeing. Thus, of course, we recommend setting this up in your custom *debug* kernel (and/or, certainly, in your production kernel as well during intense testing, where performance isn't what matters).

The usual make menuconfig UI can be used to set up the kernel config; the relevant menu is here: **Kernel hacking | Memory Debugging | Kernel memory leak detector**.

### Tip

You can edit the kernel config file non-interactively by leveraging the kernel's built-in config script here: `scripts/config` (it's a bash script). Just run it, it will display a help screen.

A quick grep for `DEBUG_KMEMLEAK` on our debug kernel's config file (`/boot/config-5.10.60-dbg02` on my system) reveals that kmemleak is indeed all set and ready to go; a one-liner regarding each is shown below:

- Does the architecture support kmemleak? yes, as  
`CONFIG_HAVE_DEBUG_KMEMLEAK=y` (implies that kmemleak is supported on this CPU)
- Kmemleak config on? yes, as `CONFIG_DEBUG_KMEMLEAK=y`
- For allocations that might occur before kmemleak is fully initialized, a memory pool of this size is used to hold metadata:  
`CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE=16000`
- Build a module for testing kmemleak: yes, as  
`CONFIG_DEBUG_KMEMLEAK_TEST=m`
- Is kmemleak *disabled* by default? Yes, as  
`CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF=y`. Enable it by passing `kmemleak=on` on the kernel command line
- Is scanning memory for leaks enabled (every 10 minutes) by default? Yes, as `CONFIG_DEBUG_KMEMLEAK_AUTO_SCAN=y`. This is considered reasonable for most systems, except perhaps low-end embedded systems.

Better chances of finding leaks are gained by enabling SLUB debug features on the kernel (`CONFIG_SLUB_DEBUG=y`). This is mainly due to the poisoning of slabs, which helps the leak detector as well. (As you'll realize from the previous section, this config option is typically on by default in any case!).

## Using kmemleak

Using kmemleak is straightforward; here's a basic 5-step checklist and steps to follow:

1. First, verify that:
  - A. debugfs is mounted and visible: we'll assume it is and mounted under the usual location, `/sys/kernel/debug`
  - B. kmemleak's enabled and running: for now, we assume it's fine; what if it isn't? more on this in the section *Addressing the issue - unable to write to the kmemleak pseudo file* that follows...
2. Run your (possibly buggy) code, or test cases, or just let the system run...
3. Initiate a memory scan: as root, do:

```
echo scan > /sys/kernel/debug/kmemleak
```

This kicks a kernel thread (no prizes for guessing that it's named `kmemleak`) into action, actively scanning memory for leaks... Once done, if a leak (or the suspicion of one) is found, a message in this format is sent to the kernel log:

```
kmemleak: 1 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
```

4. View the result of the scan by looking up the `kmemleak` debugfs pseudo file:

```
cat /sys/kernel/debug/kmemleak
```

5. (Optional) Clear all the current memory leak results (as root, do):

```
echo clear > /sys/kernel/debug/kmemleak
```

Note that as long as `kmemleak` memory scanning is active (it is by default), new leaks could come up; they can be seen by again simply reading the `kmemleak` debugfs pseudo file.

Before going any further, it's quite possible that *step 1.B* comes up – `kmemleak` may not be enabled in the first place! The section below helps you troubleshoot and figure it out. Once you've read it, we'll move onto trying out `kmemleak` with our leaky test cases!

### Addressing the issue – unable to write to the `kmemleak` pseudo file

A common issue could turn up, when attempting to write to the `kmemleak` debugfs file; often, you get an error like this:

```
echo scan > /sys/kernel/debug/kmemleak
bash: echo: write error: Operation not permitted
```

Search the kernel log (via `dmesg` or `journalctl -k`) for a message like this:

```
kmemleak: Kernel memory leak detector disabled
```

If it does show up, it obviously shows that `kmemleak`, though configured, is still disabled at runtime. How come? *It usually implies that `kmemleak` hasn't been*

correctly or fully enabled yet.

A simple yet interesting way to debug what went wrong at boot (we certainly mentioned this technique in *Chapter 3, Debug via Instrumentation - printk and friends*): at boot, ensure you pass the `debug` and `initcall_debug` parameters on the kernel command line, in order to enable debug printk's and see details of all kernel init hooks. Now, once booted and running, do this:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.10.60-dbg02-gcc root=UUID=<...> ro quiet
```

Look up the kernel log, searching for `kmemleak`:

```
$ journalctl --output=short-unix -k |grep -iC2 "kmemleak"
1637844902.306232 dbg-LKD kernel: random: get_random_u64 called from __kmem_cache_create+0x2f/0x500 with crng_init=0
1637844902.306303 dbg-LKD kernel: SLUB: HWAlign=64, Order=0-3, MinObjects=0, CPUs=6, Nodes=1
1637844902.306367 dbg-LKD kernel: kmemleak: Kernel memory leak detector disabled
1637844902.306441 dbg-LKD kernel: Kernel/User page tables isolation: enabled
1637844902.306506 dbg-LKD kernel: ftrace: allocating 44433 entries in 174 pages
...
1637844902.629853 dbg-LKD kernel: calling split_huge_pages_debugfs+0x0/0x29 @ 1
1637844902.629942 dbg-LKD kernel: initcall split_huge_pages_debugfs+0x0/0x29 returned 0 after 23 usecs
1637844902.630024 dbg-LKD kernel: calling kmemleak_late_init+0x0/0xa1 @ 1
1637844902.630093 dbg-LKD kernel: initcall kmemleak_late_init+0x0/0xa1 returned -12 after 30 usecs
1637844902.630159 dbg-LKD kernel: calling check_early_ioremap_leak+0x0/0x9e @ 1
1637844902.630225 dbg-LKD kernel: initcall check_early_ioremap_leak+0x0/0x9e returned 0 after 872 usecs
```

Figure 6.13 – Screenshot showing how `kmemleak` failed at boot

We can see:

- The init function `kmemleak_late_init()` failed, returning the value `-12`; this is the negative `errno` value of course (recall the kernel's `0/-E` return convention)
- `errno` value `12` is `ENOMEM`, implying it failed as it ran out of memory
- The error probably occurred here, in the initialization code of `kmemleak`:

```
// mm/kmemleak.c
static int __init kmemleak_late_init(void)
{
 kmemleak_initialized = 1;
 debugfs_create_file("kmemleak", 0644, NULL, NULL, &kmemleak_fop);
 if (kmemleak_error) {
 /* Some error occurred and kmemleak was disabled.
 * There is a [...] */
 schedule_work(&cleanin_work);
```

```
 ...
 return -ENOMEM;
} [...]
```

One possible reason that the `kmemleak_error` variable gets set is that the early log buffer used by kmemleak at boot isn't quite large enough. The size is a kernel config, `CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE`, and typically defaults to 16000 bytes (as you can see a bit above). So, let's try changing it to a larger value and retry. (By the way, this config was earlier named `CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE`; it was renamed to `CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE` in the 5.5 kernel).

```
$ scripts/config -s CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE
16000
$ scripts/config-set-val CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE 32000
$ scripts/config -s CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE
32000
```

Build the (debug) kernel, reboot, and test.

Guess what? We get the very same error: `dmesg` again shows that kmemleak is disabled!

A dollop of thought will reveal the actual – and rather silly – issue: to enable kmemleak, we *must* pass `kmemleak=on` via the kernel command line. (In fact, we already mentioned this very point in the section on configuring kmemleak: *Is kmemleak disabled by default? Yes, as `CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF=y`.* Enable it by passing `kmemleak=on` on the kernel command line).

Once this is done, all seems well (I even set the value of `CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE` back to its default of 16000, rebuilt the kernel and rebooted). Then:

```
$ dmesg |grep "kmemleak"
[0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.10.60-dbg02
[...]
[6.743927] kmemleak: Kernel memory leak detector initialized (me
[6.743956] kmemleak: Automatic memory scanning thread started
```

All good! Here's the kmemleak kernel thread:

```
$ ps -e|grep kmemleak
1111 2222 3333 4444 5555 6666
```

It, in fact, is deliberately run at a lower priority (a nice value of 10) thus only running when most other threads yield (recall that on Linux, the nice value ranges from -20 to +19, with -20 being the highest priority. By the way, I checked the nice value by running `ps -el` instead of just `ps -e` ).

## The nature of debugging

So, this particular debug session turned out to be a bit of a non-event; that's okay, we eventually figured it out, got kmemleak enabled and that's what matters. It also shows a truth about the nature of debugging – quite often, we'll chase down a path (or several) that really doesn't lead anywhere (the so-called **red herrings**). Worry not, it's all part of the experience! In fact, it helps; one always learns something!

(For the curious: passing `kmemleak=on` as a kernel parameter caused this function – `mm/kmemleak.c:kmemleak_boot_config()` to set the variable `kmemleak_skip_disable` to `1`, which, well, skips disabling it at boot, which is what occurs otherwise).

## Running our test cases and catching leakage defects

Now that kmemleak's enabled and running, let's get to the interesting bit – running our buggy, leaky as heck, test cases! We have three of them: without further ado, let's get started with the first one.

Running test case 3.1 – simple memory leakage

The code of our first memory leakage testcase is as follows:

```
// ch5/kmembugs_test/kmembugs_test.c
void leak_simple1(void)
{
volatile char *p = NULL;
pr_info("testcase 3.1: simple memory leak testcase 1\n");
p = kzalloc(1520, GFP_KERNEL);
if (unlikely(!p))
return;
pr_info("kzalloc(1520) = 0x%px\n", p);
if (0) // test: ensure it isn't freed
kfree((char *)p);
```

```

#ifndef CONFIG_MODULES
pr_info("kmem_cache_alloc(task_struct) = 0x%px\n",
kmem_cache_alloc(task_struct, GFP_KERNEL));
#endif
pr_info("vmalloc(5*1024) = 0x%px\n", vmalloc(5*1024));
}

```

Clearly, this code has three memory leaks – the `kzalloc()` of 1520 bytes, the `kmem_cache_alloc()` of a `task_struct` object from its slab cache and the `vmalloc()` of 5 kilobytes. Notice though, that due to `CONFIG_MODULES` being set, the second case doesn't actually run, leaving us with two leaks. (Here, I haven't shown explicit code to check for the failure case for the latter two; you should check of course).

As mentioned at the beginning of this section, *Using kmemleak*, let's now perform steps 1 to 5 (1 to 4, as step 5 is optional; it's shown later):

1. Verify kmemleak's enabled and running already done, kmemleak is enabled, it's kthread alive and well.
2. Run the test case:

```

cd <booksrc>/ch5/kmembugs_test
./load_testmod
[...]
sudo ./run_tests-no-clear
• no_clear: will not clear kernel log buffer after running a
Debugfs file: /sys/kernel/debug/test_kmembugs/lkd_dbgfs_run_tes
Generic KASAN: enabled
UBSAN: enabled
KMEMLEAK: enabled

Select testcase to run:
1 Uninitialized Memory Read - UMR
[...]
Memory leakage
3.1 simple memory leakage testcase1
3.2 simple memory leakage testcase2 - caller to free memory
3.3 simple memory leakage testcase3 - memleak in interrupt ctx
[...]
(Type in the testcase number to run):
3.1
Running testcase "3.1" via test module now...
[...]
[4053 0001551 testcase to run] 3.1

```

```
[4053.909169] test_kmembugs:leak_simple1(): testcase 3.1: simple
[4053.909212] test_kmembugs:leak_simple1(): kzalloc(1520) = 0x
[4053.909390] test_kmembugs:leak_simple1(): vmalloc(5*1024) = 0x
```

You can see from the output: our `run_tests` bash script first does a few quick config checks and determines that KASAN, UBSAN and KMEMLEAK are all enabled (do browse through the script's code on the book's GitHub repo). It then displays the menu of available test cases and has us select one; we type in `3.1`. The debugfs write hook, upon seeing this, invokes the test function – in this case, `leak_simple1()`. It executes and you can see it's `printk` output above; of course, it's buggy, leaking memory twice, as expected.

3. The key part! Initiate a memory scan, as root:

```
sudo sh -c "echo scan > /sys/kernel/debug/kmemleak"
```

Hang on tight, the memory scan can take some time (on my x86\_64 Ubuntu VM running our custom debug kernel, it takes approximately 8 to 9 seconds) ...

4. We read the content within the `kmemleak` (pseudo)file (in the section immediately following this). Once done, and a potential leak(s) is found, the kernel log will show something like this:

```
dmesg | tail -n1
kmemleak: 2 new suspected memory leaks (see /sys/kernel/debug/km
```

It even prompts you to *now* lookup the `kmemleak` pseudo file `/sys/kernel/debug/kmemleak`. (You could always rig up a script to poll for a line like this within the kernel log and only then read the scan report; I'll leave stuff like this to you as an exercise).

**Interpreting `kmemleak`'s report:** So, let's look up the details, as `kmemleak`'s urging us to:

*Figure 6.14 – Screenshot: kmemleak showing the memory leakage report for our test case #3.1*

Aha. the screenshot shows that both leakage bugs have indeed been caught!

Bravo.

Let's interpret (the first of) kmemleak's report:

- unreferenced object `0xffff8880127f8000` (size 2048) : The **kva** – the **kernel virtual address** – of the unreferenced object, the orphaned memory chunk, the one that was allocated but not freed, is displayed, followed by the size in bytes
- But our test code did a `kzalloc(1520, GFP_KERNEL)`; asking for 1520 bytes whereas the report shows the allocated size as 2048 bytes? We know why this is: the slab layer allocates memory on a best-fit basis; the closest (higher) cache to the size we want is the `kmalloc-2k` one, thus the size shows up as 2048 bytes
- comm "run\_tests", pid 5498, jiffies 4296684851 (age 84.734s) This line (above) shows the (process) context in which the leak occurred, the value of the `jiffies` variable when it occurred and the age – this is the time elapsed from when the process context (the one that ran the leaky kernel code) ran... (doing `sudo cat /sys/kernel/debug/kmemleak` a little later will show that the age has increased!). Keeping an eye on the age field can be useful: it allows you to see if a detected leak is an old one (you can then clear the list by writing `clear` to the `kmemleak` pseudo file)
- Next, a hex dump of the first 32 bytes of the affected memory chunk is displayed (as we did a `kzalloc()`, the memory is initialized to all zeroes)
- This is followed by the crucial information – a stack backtrace , which of course, you read bottom-up; in this particular first leak test case, you can see from the trace that a write system call was issued (this originates from the `echo` command we issued, of course); it, as expected, ended up in our debugfs write routine.
- Next, we see: `dbgfs_run_testcase+0x1c7/0x51a [test_kmembugs]` ; as mentioned earlier, the code that ran was at an offset of `0x1c7` bytes from the start of this function and the length of the function is `0x51a` bytes. The module name is in square brackets, showing that this function lives within that module
  - It called the function `leak_simple1()` , again, within our module
  - This function, as we know, issued a `kzalloc()` which is a simple wrapper around the `kmalloc()` , which works by allocating memory from an existing slab cache (one of the `kmalloc-*()`)

slab caches; as mentioned above, it will be the one named  
kmalloc-2k )

- this allocation is internally done via the `kmem_cache_alloc()` API, which kmemleak tracks (thus it shows up in the stack backtrace as `kmem_cache_alloc_trace()` ).

So, there we are; we can see that our test case indeed caused the leak! The interpretation of the second leak is completely analogous; this time, the stack backtrace clearly shows that the `leak_simple1()` function in the `test_kmembugs` module invoked the `vmalloc()`, which we didn't free, causing the leak.

5. Optionally (and step 5 of the above procedure), we can clear (as root) all the current memory leak results:

```
$ sudo sh -c "echo clear > /sys/kernel/debug/kmemleak"
$ sudo cat /sys/kernel/debug/kmemleak
```

Done. Clearing the previous results is useful, allowing you to de-clutter the report output; this is especially true when running development code (or test cases) over and over.

Running testcase 3.2 – the ‘caller-must-free’ case

Our second memory leakage test case is interesting: here, we invoke a function (named `leak_simple2()`) which allocates a small 8-byte piece of memory via `kmalloc()` and sets it to the string `leaky!!`. It then returns the pointer to this memory object to the caller. This is fine. The caller then collects the result in another pointer and prints its value – it's as expected. Here's the code of the caller:

```
// ch5/kmembugs_test/debugfs_kmembugs.c
[...]
else if (!strncmp(udata, "3.2", 4)) {
 res2 = (char *)leak_simple2();
 // caller's expected to free the memory!
 pr_info(" res2 = \"%s\"\n", res2 == NULL ? "<whoops, it's NULL>" :
 if (0) // test: ensure it isn't freed by us, the caller
 kfree((char *)res2);
}
```

Run it (via our `run_tests` script), then perform the usual:

```
$ sudo sh -c "echo scan > /sys/kernel/debug/kmemleak"
$ sudo cat /sys/kernel/debug/kmemleak
unreferenced object 0xfffff8880074b5d20 (size 8):
comm "run_tests", pid 5779, jiffies 4298012622 (age 181.044s)
hex dump (first 8 bytes):
6c 65 61 6b 79 21 21 00 leaky!!.
backtrace:
[<00000000c0b84cb6>] slab_post_alloc_hook+0x78/0x5b0
[<00000000f76c1d8d>] kmem_cache_alloc_trace+0x16b/0x370
[<000000009f614545>] leak_simple2+0xc0/0x19b [test_kmembugs]
[<00000000747f9f09>] dbgfs_run_testcase+0x1e6/0x51a [test_kmembugs]
[...]
```

Great, it's caught. Interestingly, the first time I ran the scan, nothing seemed to be detected. Running it again after a minute or so yielded the expected result – it reported 1 new suspected memory leak. Also, having the address of the unreferenced (or orphaned) memory buffer lets you investigate more on it via the `kmemleak dump` command; we cover it and related stuff in the upcoming section *Controlling the kmemleak scanner*.

Running testcase 3.3 – memory leak in interrupt context

Until now we've been pretty much exclusively running our test cases by having a process run through our (buggy) kernel module code; this, of course, implies that the kernel code was run in *process context*. The other context in which kernel code can possibly run is *interrupt context*, literally, within the context of an interrupt.

## Types of interrupt contexts

More precisely, within interrupt context, we can have a hardirq (the actual hardware interrupt handler), the so-called softirq and the tasklet (the common way in which bottom halves are implemented; the tasklet is a type of softirq). These details (and a lot more!) are covered in depth in my earlier book *Linux Kernel Programming – Part 2* (hey, it's freely downloadable too).

So, what if we have a memory leak in code that runs in interrupt context? Will kmemleak detect it? The only way to know is to try – the empirical approach!

The code of our third – interrupt-context – memory leakage testcase is as follows:

```
// ch5/kmembugs_test/kmembugs_test.c
void leak_simple3(void)
{
pr_info("testcase 3.3: simple memory leak testcase 3\n");
irq_work_queue(&irqwork);
}
```

To achieve running in interrupt context without an actual device that generates interrupts, we make use of a kernel feature – the `irq_work*` functionality. It allows the ability to run code in interrupt (hardirq) context. Without going into details, to set this up, in the init code of our module, we called the `init_irq_work()` API; it registers the fact that our function named `irq_work_leaky()` will be invoked in hardirq context when the `irq_work_queue()` function triggers it. This is the code of the actual interrupt context function:

```
/* This function runs in (hardirq) interrupt context */
void irq_work_leaky(struct irq_work *irqwk)
{
int want_sleep_in_atomic_bug = 0;
PRINT_CTX();
if (want_sleep_in_atomic_bug == 1)
pr_debug("kzalloc(129) = 0x%px\n", kzalloc(129, GFP_KERNEL));
else
pr_debug("kzalloc(129) = 0x%px\n", kzalloc(129, GFP_ATOMIC));
}
```

The leakage bug is obvious; did you see the sneaky bug that we can cause to surface (if you set the variable `want_sleep_in_atomic_bug` to `1`; this leads to an allocation with the `GFP_KERNEL` flag in an *atomic* context, a bug! Okay, we'll ignore that for now as it won't trigger as the variable is set to `0` (try it out and see, though).

We execute the testcase (via our trusty `run_tests` wrapper script); to be safe, let's clear the kmemleak internal state first:

```
sudo sh -c "echo clear > /sys/kernel/debug/kmemleak"
```

Then do this:

```

$ sudo sh -c "echo scan > /sys/kernel/debug/kmemleak" ; dmesg |tail
[34619.682989] test_kmembugs:kmembugs_test_init(): KASAN configured
[34619.684794] test_kmembugs:kmembugs_test_init(): CONFIG_UBSAN configured
[34619.686614] test_kmembugs:kmembugs_test_init(): CONFIG_DEBUG_KMEMLEAK configured
[34619.688443] debugfs file 1 <debugfs_mountpt>/test_kmembugs/lkd_dbgfs_run testcase created
[34619.690270] debugfs entry initialized
[35412.528017] testcase to run: 3.3
[35412.530040] test_kmembugs:leak_simple3(): testcase 3.3: simple memory leak testcase 3
[35412.532750] test_kmembugs:irq_work_leaky(): 001 run_tests :11781 | d.h1 /* irq_work_leaky() */
[35412.537365] test_kmembugs:irq_work_leaky(): kzalloc(129) = 0xfffff88803c1e0e00
[35438.671971] kmemleak: 1 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
$
$ sudo cat /sys/kernel/debug/kmemleak
unreferenced object 0xfffff88803c1e0e00 (size 192):
comm "hardirq", pid 0, jiffies 4305500943 (age 34.834s)
hex dump (first 32 bytes):
 00
backtrace:
[<00000000c0b84cb6>] slab_post_alloc_hook+0x78/0x5b
[<00000000f76c1d8d>] kmem_cache_alloc_trace+0x16b/0x370
[<000000002912ff8c>] irq_work_leaky+0x1f3/0x226 [test_kmembugs]
[<00000000b094c375>] irq_work_single+0x8f/0xf0
[<000000005a10caf0>] irq_work_run_list+0x52/0x70
[<00000000e07f0913>] irq_work_run+0x6b/0x110
[<000000006d70efc1>] __sysvec_irq_work+0x75/0x2b0
[<00000000038851639>] asm_call_irq_on_stack+0x12/0x20
[<000000006e1838aa>] sysvec_irq_work+0xc3/0xe0
[<0000000043c320fa>] asm_sysvec_irq_work+0x12/0x20
[<000000007864aefa>] native_write_msr+0x6/0x30
[<0000000041cbb6ac>] x2apic_send_IPI_self+0x3c/0x50
[<00000000b30d6970>] arch_irq_work_raise+0x5d/0x90
[<00000000848d8ab3>] __irq_work_queue_local+0xf8/0x170
[<00000000a3bb972c>] irq_work_queue+0x32/0x50
[<000000005b977e7a>] leak_simple3+0x2f/0x31 [test_kmembugs]
$
```

*Figure 6.15 – Screenshot: kmemleak catches the leak in interrupt context*

Notice:

- The test case 3.3 runs; our convenient .h:PRINT\_CTX() macro shows the context: you can see the d.h1 token within, showing that the function irq\_work\_leaky() ran in hardirq interrupt context (we covered interpreting the PRINT\_CTX() macro's output in *Chapter 4, Debug via Instrumentation – Kprobes* section *Interpreting the PRINT\_CTX() macro's output*)
- The top line shows we ran the kmemleak scan command, getting it to check for any leakage
- The read of the kmemleak pseudo file shows the story: the orphaned or unreferenced object, the memory buffer we allocated but didn't free; this time the context shows as "hardirq" - perfect, the leak did indeed occur in an interrupt, not process, context. This is followed by the hex dump of the first 32 bytes and then the stack backtrace (whose output verifies the situation).

Next, let's check out the kernel's built-in kmemleak test module.

## The kernel's kmemleak test module

While configuring the kernel for kmemleak, we set

`CONFIG_DEBUG_KMEMLEAK_TEST=m`. This has the build generate the kmemleak-test kernel module; for my (guest) system, here:

`/lib/modules/$(uname -r)/kernel/samples/kmemleak/kmemleak-test.ko`.

The code's present within the samples folder of the kernel source tree:  
`samples/kmemleak/kmemleak-test.c`. Please do take a peek; though short and sweet (and full of leaks), it quite comprehensively runs memory leakage tests! I inserted it with:

```
sudo modprobe kmemleak-test
```

The `dmesg` output is seen below; I also did the scan and read the kmemleak report; it caught all 13 memory leaks ( ! ). The first of its reports (catching the first leak) is visible below as well:

```

[8825.985116] kmemleak: Kmemleak testing
[8825.985147] kmemleak: kmalloc(32) = 00000000cab708dd
[8825.985172] kmemleak: kmalloc(32) = 000000008d5c540a
[8825.985196] kmemleak: kmalloc(1024) = 0000000006d719a53
[8825.985221] kmemleak: kmalloc(1024) = 00000000de599e5e
[8825.985247] kmemleak: kmalloc(2048) = 00000000b5e60406
[8825.985272] kmemleak: kmalloc(2048) = 000000000309c294
[8825.985299] kmemleak: kmalloc(4096) = 000000009200f455
[8825.985324] kmemleak: kmalloc(4096) = 000000001cfde96d
[8825.985555] kmemleak: vmalloc(64) = 00000000b7894b61
[8825.985672] kmemleak: vmalloc(64) = 00000000bb401d6
[8825.985796] kmemleak: vmalloc(64) = 000000009c4e811f
[8825.985893] kmemleak: vmalloc(64) = 000000001e8fcc4a
[8825.985999] kmemleak: vmalloc(64) = 000000007f7b580a
[8825.986025] kmemleak: kzalloc(sizeof(*elem)) = 00000000d68f3627
[8825.986048] kmemleak: kzalloc(sizeof(*elem)) = 000000008bcc71cd
[8825.986070] kmemleak: kzalloc(sizeof(*elem)) = 00000000d90adbf5
[8825.986092] kmemleak: kzalloc(sizeof(*elem)) = 000000004c07e127
[8825.986115] kmemleak: kzalloc(sizeof(*elem)) = 00000000226b752f
[8825.986141] kmemleak: kzalloc(sizeof(*elem)) = 00000000d7eaeed8
[8825.986164] kmemleak: kzalloc(sizeof(*elem)) = 000000006ed69561
[8825.986187] kmemleak: kzalloc(sizeof(*elem)) = 00000000a79442e4
[8825.986209] kmemleak: kzalloc(sizeof(*elem)) = 0000000083a42752
[8825.986231] kmemleak: kzalloc(sizeof(*elem)) = 00000000412c4a56
[8825.986259] kmemleak: kmalloc(129) = 000000005c48a002
[8825.986281] kmemleak: kmalloc(129) = 000000000700d3c9
[8825.986304] kmemleak: kmalloc(129) = 0000000000e572f9
[8825.986327] kmemleak: kmalloc(129) = 000000002943f11c
[8825.986351] kmemleak: kmalloc(129) = 00000000f9236807
[8825.986372] kmemleak: kmalloc(129) = 00000000b9efae8e
$ time sudo sh -c "echo scan > /sys/kernel/debug/kmemleak"

real 0m8.950s
user 0m0.000s
sys 0m8.947s
$ dmesg |tail -n1
[8860.390327] kmemleak: 13 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
$ sudo cat /sys/kernel/debug/kmemleak
unreferenced object 0xfffff88800df30540 (size 32):
 comm "modprobe", pid 5647, jiffies 4297524992 (age 866.434s)
 hex dump (first 32 bytes):
 00
 backtrace:
 [<00000000c0b84cb6>] slab_post_alloc_hook+0x78/0x5b0
 [<00000000f76c1d8d>] kmem_cache_alloc_trace+0x16b/0x370
 [<00000000e1aa9887>] 0xfffffffffc080f058
 [<00000000deb5ae43>] do_one_initcall+0xcb/0x430
 [<00000000fc291604>] do_init_module+0x10f/0x3b0
 [<000000000977ca321>] load_module+0x3f49/0x4570
 [<00000000040c61d85>] __do_sys_finit_module+0x12a/0x1b0
 [<00000000d87c4816>] __x64_sys_finit_module+0x43/0x50
 [<0000000001a646102>] do_syscall_64+0x38/0x90
 [<0000000024b0a009>] entry_SYSCALL_64_after_hwframe+0x44/0xa9

```

*Figure 6.16 – Screenshot: output from trying out the kernel’s kmemleak-test module; the first kmemleak report is seen at the bottom*

Notice that the address of the allocated memory buffers are printed with the %p specifier, leading to the kernel hashing it (info-leak prevention, security), but the kmemleak report shows the actual kernel virtual address. (Do try it out yourself and read the full report).

## Controlling the kmemleak scanner

This is the kmemleak debugfs pseudo file, our means to work with kmemleak:

```
$ sudo ls -l /sys/kernel/debug/kmemleak
• rw-r-r-- 1 root root 0 Nov 26 11:34 /sys/kernel/debug/kmemleak
```

As you know by now, reading from it has the underlying kernel callback display the last memory leakage report, if any. We've also seen a few values that can be written to it, in order to control and modify kmemleak's actions at runtime. There are a few more; we summarize all values you can write (you'll need root access of course) in the table below:

| String to write to<br><b>/sys/kernel/debug/kmemleak</b><br><b>(as root)</b> | Effect                                                                                                                                                                                                                                                        |
|-----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| clear                                                                       | Clears all existing memory leak suspects from its disabled, frees all kmemleak meta objects                                                                                                                                                                   |
| dump=<kva>                                                                  | Dump information on the object (memory chunk) example,<br># echo dump=0xfffff88800df30540 > /sys/kernel/debug/kmemleak/dump<br>dumps information about the particular memory object (look up the address from the kmemleak report or kmemleak --find command) |
| off                                                                         | Disables kmemleak; note that this action is irreversible (the kmemleak meta objects might still be kept (and can sometimes occupy significant RAM)); to free them, write clear to kmemleak pseudo file                                                        |
| scan                                                                        | Kick starts a memory scan; kmemleak searches for memory leaks                                                                                                                                                                                                 |
| scan=on                                                                     | Start the automatic scan of memory via the kmemleak --auto parameter                                                                                                                                                                                          |
| scan=<seconds>                                                              | Sets the automated memory scan (by the kmemleak --auto parameter) to run every <seconds> seconds; default is 600, 0 disables automatic scan                                                                                                                   |

|           |                                                                |
|-----------|----------------------------------------------------------------|
| scan=off  | Stop the automatic scan of memory via the kmemleak pseudo file |
| stack=on  | Enable task stack scanning (default)                           |
| stack=off | Disable task stack scanning                                    |

Table 6.3 – Values to write to the kmemleak pseudo file to control it

The code that governs the action to be taken on these writes can be seen here: `mm/kmemleak.c:kmemleak_write()`.

**A quick tip:** if you need to test something specific and want a clean slate, as such, it's easy to do first clean the kmemleak internal list, run your module or test(s), run the scan command, followed by the read to the kmemleak pseudo file. So, something like this:

```
echo clean > /sys/kernel/debug/kmemleak
//... run your module / test cases(s) / kernel code / ...
// wait a bit ...
echo scan > /sys/kernel/debug/kmemleak
// check dmesg last line to see if new leak(s) have been found by kmemleak
// If so, get the report
cat /sys/kernel/debug/kmemleak
```

As with any such tool, the possibility of false positives is present; the official kernel documentation provides some tips on how you could deal with it (if required): <https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html#dealing-with-false-positives-negatives>. This document also covers some details on the internal algorithm used by kmemleak to detect memory leakage; do check it out (<https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html#basic-algorithm>).

### A few tips for developers regarding dynamic kernel memory allocation

Though not directly related to debug, we feel it's well worth mentioning a few tips with respect to dynamic kernel memory alloc and free, for a modern driver or module author. This is in line with the age-old principle – prevention is better than cure!

### Preventing leakage with the modern devres memory allocation APIs

Modern driver authors should definitely exploit the kernel's **resource-managed** (or **devres**) `devm_k{m,z}alloc()` APIs. The key point: *they allow you to allocate memory and not worry about freeing it!* Though there are several (they all are of the form `devm_*()`), let's focus on the common case, the following dynamic memory allocation APIs for you, the typical driver author:

```
void *devm_kmalloc(struct device *dev, size_t size, gfp_t gfp);
void *devm_kzalloc(struct device *dev, size_t size, gfp_t gfp);
```

(Why do we stress that only driver authors are to use them? Simple: the first required parameter is a pointer to the device structure, typical in all kinds of device drivers).

The reason why these resource-managed APIs are useful is that there is no need for the developer to explicitly free the memory allocated by them. The kernel resource management framework guarantees that it will automatically free the memory buffer upon driver detach, and/or if a kernel module, when the module is removed (or the device is detached, whichever occurs first).

As you'll surely realize, this feature immediately enhances code robustness. Why? Simple, we're all human and make mistakes. Leaking memory (especially on error code paths) is indeed a fairly common bug!

A few relevant points regarding the usage of these devres APIs:

- A key point – don't attempt to blindly replace `k[m|z]alloc()` with the corresponding `devm_k[m|z]alloc()` APIs! These resource-managed allocations are really designed to be used only in the `init` and/or `probe()` methods of a device driver (all drivers that work with the kernel's unified device model will typically supply the `probe()` and `remove()` (or `disconnect()`) methods. We will not delve into these aspects here).
- `devm_kzalloc()` is usually preferred as it initializes the buffer as well, thus eclipsing the, again all too common, uninitialized memory read (UMR) types of defects. Internally (as with `kzalloc()`), it is merely a thin wrapper over the `devm_kmalloc()` API. (It's popular: the 5.10.60 kernel has the `devm_kzalloc()` being invoked well over 5000 times).
- The second and third parameters are the usual ones, as with the `k[m|z]alloc()` APIs – the number of bytes to allocate and the GFP flags to use. The first parameter, though, is a pointer to `struct device`. Quite obviously, it represents the device that your driver is driving.

- As the memory allocated by these APIs is auto-freed (on driver detach or module removal), you don't have to do anything. It can, though, be freed via the `devm_kfree()` API. You are doing this, however, is usually an indication that the managed APIs are the wrong ones to use...
- The managed APIs are exported (and thus available) only to modules licensed (also) under the GPL (ah, the sweet revenge of the kernel community).

A few more tips on memory-related issues for developers follows...

### Other (more developer-biased) common memory-related bugs

Studies have shown that *<insert anything you'd like here>*. Okay, jokes aside, there's evidence to suggest that preventing bugs during the development cycle (and/or early unit testing) itself causes the least impact on the product (both cost-wise and otherwise). Good, solid coding practices are skills one continually hones as a developer. As we've seen, when it comes to working directly with memory, a non-managed language like C can be a nightmare, both bug and security-wise. Thus, a hopefully useful quick checklist with regard to the kernel's memory alloc/free slab APIs:

- Performing a kernel slab allocation with the wrong GFP flag(s); for example, with `GFP_KERNEL` when in an atomic context (like an interrupt context of any sort, or when holding a spinlock; here, you should use `GFP_ATOMIC` flag of course!)

For instance, here's the patch to one such bug:

<https://lore.kernel.org/lkml/1420845382-25815-1-git-send-email-khoroshilov@ispras.ru/>

- memory allocated via `k{m|z}alloc()` but freed with `vfree()`, and vice-versa
- Not checking the failure case (NULL, of course, for memory allocations); this might seem pedantic, but it can and does happen! Using the `if (unlikely(!p) [...])` kind of semantic is fine
- Doing things like:

```
if (p)
 kfree(p);
```

It's not required (but quite harmless; still, don't). The reverse isn't: only performing some action after a free *conditionally*, if the pointer is NULL; in other words, assuming that the free interface sets the pointer variable to NULL! It does not (though that would be quite intuitive)

- Failing to realize the wastage (internal fragmentation) that can occur when allocating memory via the slab layer; use the `ksize()` API to see the *actual* number of bytes allocated. (For example, in this pseudocode:  
`p = kmalloc(4097); n = ksize(p);` You'll find the value of `n` – the actual memory allocated – is 8192, implying a wastage of  $8192 - 4097 = 4095$  bytes, or almost a 100%! Ask yourself: could I not redesign to allocate 4096 bytes via `kmalloc()`, instead of 4097?).

With this, we complete our detailed coverage on understanding and catching the dangerous memory leakage defect within the kernel! Great going. Let's complete this long chapter with a kind of summarization of the many tools and techniques we've used.

# Catching memory defects in the kernel – comparisons and notes (Part 2)

The table that follows is an extension of the one in the previous chapter (*Table 5.5*), adding on the right-most column, that of employing the kernel’s SLUB debug framework. Here, we tabulate and hence summarize our test case results for our test runs with all the tooling technologies / kernels – vanilla/distro kernel, compiler warnings, with KASAN, with UBSAN and with SLUB debug with our debug kernel – we employed in the preceding and this chapter. In effect, *it’s a compilation of all the findings in one place*, thus allowing you to make quick (and hopefully helpful) comparisons.

| Testcase # [1] Memory defect type<br>(below) / Infrastructure used (right) | Distro Compiler With                                               |     |                |        | W<br>[5] |
|----------------------------------------------------------------------------|--------------------------------------------------------------------|-----|----------------|--------|----------|
|                                                                            | [2]                                                                | [3] | KASAN With [4] | UBSAN  |          |
| <b>Defects not covered by the kernel’s KUnit</b>                           |                                                                    |     |                |        |          |
| test_kasan.ko module                                                       |                                                                    |     |                |        |          |
| 1                                                                          | Uninitialized Memory Read - UMR                                    | N   | Y [C1]         | N      | N N      |
| 2                                                                          | Use After Return - UAR                                             | N   | Y [C2]         | N [SA] | N [SA] N |
| 3                                                                          | Memory leakage [6]                                                 | N   | N              | N      | N N      |
| <b>Defects covered by the kernel’s KUnit</b>                               |                                                                    |     |                |        |          |
| test_kasan.ko module                                                       |                                                                    |     |                |        |          |
| 4                                                                          | OOB accesses on static global (compile-time) <small>memory</small> |     |                |        |          |

| memory |                                                                       |           |        |        |              |   |
|--------|-----------------------------------------------------------------------|-----------|--------|--------|--------------|---|
|        |                                                                       | N<br>[V1] | N      | Y [K1] | Y<br>[U1,U2] | N |
| 4.1    | Read (right) overflow                                                 |           |        |        |              |   |
| 4.2    | Write (right) overflow                                                | N         | Y [K1] |        |              |   |
| 4.3    | Read (left) underflow                                                 | N         | Y [K2] |        |              |   |
| 4.4    | Write (left) underflow                                                | N         | Y [K2] |        |              |   |
| 4      | OOB accesses on static<br>global (compile-time)<br>stack local memory |           |        |        |              |   |
| 4.1    | Read (right) overflow                                                 | N<br>[V1] | N      | Y [K3] | Y<br>[U1,U2] | N |
| 4.2    | Write (right) overflow                                                | N         | Y [K2] |        |              |   |
| 4.3    | Read (left) underflow                                                 | N         | Y [K2] |        |              |   |
| 4.4    | Write (left) underflow                                                |           |        |        |              |   |
| 5      | OOB accesses on dynamic<br>(kmalloc-ed slab) memory                   |           |        |        |              |   |
| 5.1    | Read (right) overflow                                                 | N         | N      | Y [K4] | N            | N |
| 5.2    | Write (right) overflow                                                |           | Y [S1] |        |              |   |
| 5.3    | Read (left) underflow                                                 | N         |        |        |              |   |
| 5.4    | Write (left) underflow                                                |           | Y [S1] |        |              |   |
| 6      | Use After Free - UAF                                                  | N         | N      | Y [K5] | N            | Y |
| 7      | Double-free                                                           | Y<br>[V2] | N      | Y [K6] | N            | Y |
| 8      | Arithmetic UB (via the<br>kernel's test_ubsan.ko<br>module)           |           |        |        |              |   |
| 8.1    | Add overflow                                                          | N         | N      | N      | Y            | N |
| 8.2    | Sub(tract) overflow                                                   | N         |        |        |              |   |
| 8.3    | Mul(tiply) overflow                                                   | N         |        |        |              |   |
| 8.4    | Negate overflow                                                       | N         |        |        |              |   |
|        | Div by zero                                                           | Y         |        |        |              |   |
| 8.5    | Bit shift OOB                                                         | Y<br>[U3] | Y [U3] | Y [U3] |              |   |

## Other than arithmetic

**UB defects  
(copied from  
the kernel's  
KUnit  
test\_ubsan.ko  
module)**

|     |                                               |           |        |        |        |   |
|-----|-----------------------------------------------|-----------|--------|--------|--------|---|
| 8.6 | OOB                                           | Y<br>[U3] | N      | Y [U3] | Y [U3] | N |
| 8.7 | Load invalid value                            | Y<br>[U3] | Y [U3] | Y [U3] |        |   |
| 8.8 | Misaligned access                             | N         | N      |        | N      |   |
| 8.9 | Object size mismatch                          | Y<br>[U3] | Y [U3] | Y [U3] |        |   |
| 9   | OOB on<br><code>copy_[to from]_user*()</code> | N         | Y [C3] | Y [K4] | N      | N |

Table 6.4 – Summary of various common memory defects and how various technologies react in catching it (or not)

(As mentioned in the previous chapter) the explanation on the foot notes within this table (like [C1], [K1], [U1], and so on) will be found in the earlier relevant section (for the ones within the first four columns, do refer the previous chapter).

So, again, a very brief summary:

- KASAN catches pretty much all OOB buggy memory accesses on global (static), stack local and dynamic (slab) memory; UBSAN doesn't catch the dynamic slab memory OOB accesses (testcases 4.x, 5.x)
- KASAN does not catch the UB defects (testcases 8.x), UBSAN does catch (most of) them
- Neither KASAN nor UBSAN catch the first three testcases – UMR, UAR and leakage bugs, *but the compiler(s) generate warnings and static analyzers (cppcheck) can catch some of them.*
- The kernel's SLUB debug framework is adept at catching most of the slab memory corruption defects, but none others
- The kernel **kmemleak** infrastructure catches kernel memory leaks allocated by any of `k{m|z}alloc()`, `vmalloc()` or `kmem_cache_alloc()` (and friends) interfaces.

**Miscellaneous notes**

Again, a few more points on the footnotes regarding *Table 6.4* above:

- [V1] the system could simply Oops or hang here or even appear to be remain unscathed; but that's not really the case... once the kernel is buggy, the system is buggy
- [V2] Please see the explanation for this detailed note in the section *Running SLUB debug test cases on a kernel with slub\_debug turned off*
- [S1] The kernel's SLUB debug infrastructure – when `slub_debug=FZPU` is passed as a kernel parameter - catches both write over and underflow (right and left) OOB accesses on slab memory. However, just as we saw with UBSAN, it only seems able to catch it when the buggy access is via incorrect indices to the memory region, *not* when the OOB access is via a pointer! Also, the OOB reads do not seem to be caught, only the writes.

So, there we are! We (finally) went through our single summary table (*Table 6.4*) for pretty much all the common memory defects and how they're caught, or not, by the tooling we've discussed in some depth.

## Summary

Most dynamic memory allocation (and freeing) in the kernel is done via the kernel's powerful slab (internally, SLUB) interfaces; to debug them, the kernel provides a strong SLUB debug framework and several associated utilities (`slabtop`, `slabratetop`, `vmstat`, and so on). Here, you learned how to catch SLUB bugs via the kernel's SLUB debug framework as well as leverage these utilities.

Among memory bugs, the very mention of the leakage defect raises dread and fear, even in very experienced developers! It's a deadly one indeed, as we (hopefully) showed you in that section! The kernel's powerful kmemleak framework can catch these dangerous leakage bugs; be sure to test your product (for long durations) with it running!

As we covered these tools and frameworks, we tabulated the results, showing you the bugs a given tool can (or cannot) catch. To then summarize the whole thing, we built a larger table with columns covering all the test cases and all the tools (*Table 6.4*) – a quick and useful way for you to see and compare (a

superset of the similar table in the previous chapter)!

Good job! You've now completed the long but really important chapters on catching memory bugs in kernel-space! Whew, plenty to chew on, yes!? I'd definitely recommend you take the time to think on and digest these topics, practicing as you go (please do the exercises suggested as well!). Then, when you've done so, take a break and let's meet in the next very interesting chapter, where we'll tackle head-on the topic of what a kernel *Oops* is and how we diagnose it. See you there!

## Further reading

- SLUB debug
  - Kernel documentation: Short users guide for SLUB:  
<https://www.kernel.org/doc/html/latest/vm/slub.html#short-users-guide-for-slub>
  - `slub_debug` : Detect Kernel heap memory corruption, TechVolve, Mar 2014: <http://techvolve.blogspot.com/2014/04/slubdebug-detect-kernel-heap-memory.html>
  - `slabratetop` example by Brendan Gregg:  
[https://github.com/iovisor/bcc/blob/master/tools/slabratetop\\_example.t](https://github.com/iovisor/bcc/blob/master/tools/slabratetop_example.t)
  - Interesting: Network Jitter: An In-Depth Case Study, Alibaba Cloud, Jan 2020, Medium: <https://alibaba-cloud.medium.com/network-jitter-an-in-depth-case-study-cb42102aa928>
  - LLVM/Clang: LLVM FAQs, omnisci:  
<https://www.omnisci.com/technical-glossary/llvm>
- Kmemleak: *Kernel Memory Leak Detector*:  
<https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html#kernel-memory-leak-detector>
- Linux Kernel Memory Leak Detection, Catalin Marinas, 2011:  
[https://events.static.linuxfound.org/images/stories/pdf/lceu11\\_marinas.pdf](https://events.static.linuxfound.org/images/stories/pdf/lceu11_marinas.pdf)
- GRUB bootloader:
  - *How To Configure GRUB2 Boot Loader Settings In Ubuntu*, Sk, Sept 2019: <https://ostechnix.com/configure-grub-2-boot-loader-settings-ubuntu-16-04/>
  - *GRUB: How do I change the default boot kernel*:  
<https://askubuntu.com/questions/216398/set-older-kernel-as-default-grub-entry>

- The Heartbleed OpenSSL (TLS) vulnerability
  - <https://heartbleed.com/>
  - <https://xkcd.com/1354/> (Brilliantly illustrated here).

# 7 Oops! Interpreting the kernel bug diagnostic

Kernel code is supposed to be perfect. It mustn't ever crash. But, of course, it does, on occasion... welcome to the real world.

When user-space code hits a (typical) bug – an invalid memory access, say – the processor **Memory Management Unit (MMU)**, upon failing to translate the invalid user-space virtual address to a physical one (via the process's paging tables), raises a fault. The fault handler within the kernel now takes control; it ultimately (and typically) results in a fatal signal (often, `SIGSEGV`) being sent to the faulting process (or thread). This of course has the process possibly handle the signal and terminate.

Now take exactly the same case – except that this time, the invalid memory access occurs in kernel-space (in kernel mode)! Hey, that's not supposed to happen, right? Well, bugs do happen, within kernel-space too. This time, the kernel fault handler, on realizing that it's kernel mode code that triggered the fault, runs code to generate an **Oops** – *a kernel diagnostic that details what happened*. (The unfortunate process context can die as well, as a side effect).

Here you will learn a key topic - what a **kernel Oops** diagnostic message is, and more importantly, how to interpret it in detail. Along the way, you will generate a simple kernel Oops and understand exactly how to interpret it. Further along, several tools and techniques to help with this task will be shown. Getting to the bottom of the Oops often helps pinpoint the root cause of the kernel bug! To help you understand more, and better spot typical issues, a few actual kernel Oops'es will also be discussed and/or pointed to.

In this chapter, we continue this discussion; here, we shall focus upon and cover the following main topics:

- Generating a simple kernel bug and Oops
- A kernel Oops and what it signifies
- Devil in the details – decoding the Oops
- Tools and techniques to help determine the location of the Oops

- An Oops on an ARM Linux system and using netconsole
- A few actual Oops'es

## Technical requirements

The technical requirements and workspace remain identical to what's described in *Chapter 1, A General Introduction to Debugging Software*. The code examples can be found within the book's GitHub repository here: <https://github.com/PacktPublishing/Linux-Kernel-Debugging>. The only thing new: we'll show you how to clone and use the useful `procmap` utility as well.

## Generating a simple kernel bug and Oops

You've heard the quote *It takes a thief to catch a thief*. So, let's first learn how to generate a kernel bug (shouldn't be too much of a challenge, duh).

As you'll know, the classic pedagogical bug is the (in)famous NULL pointer dereference (a section soon follows that elaborates on it). So, here's the plan:

- We'll first write a very simple kernel module that performs the cardinal sin of dereferencing the NULL pointer (the address `0x0`); we'll call it our version 1, `oops_tryv1` module
- Once you try it out, we'll move on to a slightly more sophisticated version 2 (`oops_tryv2`) module; within it, we'll provide multiple – three distinct ways – by which to generate an Oops!

Before embarking on our *generate-an-Oops* quest, let's better understand what the `procmap` utility does and what the NULL trap page is. First, let's go with the utility.

## The `procmap` utility

Being able to *visualize* the complete memory map of the kernel **Virtual Address Space (VAS)** as well as any given process's user VAS is what the `procmap` utility is designed to do. (Full disclosure: I'm the original author).

The description on its GitHub page (<https://github.com/kaiwan/procmap>) sums it up:

*procmap is designed to be a console/CLI utility to visualize the complete memory map of a Linux process, in effect, to visualize the memory mappings of both the kernel and usermode Virtual Address Spaces (VAS). It outputs a simple visualization of the complete memory map of a given process in a vertically-tiled format ordered by descending virtual address (see screenshots below). The script has the intelligence to show kernel and user space mappings as well as calculate and show the sparse memory regions that will be present. Also, each segment or mapping is (very approximately) scaled by relative size and color-coded for readability. On 64-bit systems, it also shows the so-called non-canonical sparse region or 'hole' (typically close to a whopping 16,384 PB on the x86\_64).*

The utility includes options to see only kernel space or user space, verbose, and debug modes, the ability to export its output in convenient CSV format to a specified file, as well as other options. It has a kernel component as well (a module) and currently works on (auto-detects) x86\_64, AArch32, and Aarch64 CPUs.

Do note, though, that it's not complete in any real sense, development is ongoing; there are several caveats. Feedback and contributions are most appreciated!

Download/clone it from here: <https://github.com/kaiwan/procmap>.

## What's this NULL trap page anyway

On all Linux-based systems (indeed, pretty much on all modern virtual memory based operating systems), the kernel splits the virtual memory region available to a process into two portions – user and kernel VAS (we call it the *VM Split*; a very detailed discussion is in the *Linux Kernel Programming – Part 1* book, *Chapter 7*).

On the x86\_64, the size of the complete VAS per process is of course  $2^{64}$  bytes. Now that's a phenomenally huge number, it's 16 EB (EB is exabytes; 1 exabyte = 1000 petabytes = 1 million terabytes = 1 billion gigabytes!). The VAS is simply far too large; so, the kernel, by default on the x86\_64, splits it like this:

- Kernel VAS of size 128 TB anchored to the top of the VAS (from the **kernel virtual address (kva)** `0xfffffffffffff` at the very top of the

VAS to kva 0xfffff8000000000000 )

- User VAS of size 128 TB anchored to the bottom of the VAS (from the **user virtual address (uva)** 0x00007fffffffffffff to uva 0x0 at the very bottom of the VAS).

## Think on this

The 64-bit VAS is so big that, in this case, we end up using just a tiny fraction of the available address space. 16 EB is 16,384 PB; of that we're using  $128\text{ TB} + 128\text{ TB} = 256\text{ TB}$  (which is  $256/1024 = 0.25\text{ PB}$ ). This implies that about 0.0015% of the available VAS is being used

Now, to the point of interest here: at the low end of the user VAS, the very first virtual page – from byte 0 to byte 4095 – is called the **NULL trap page**. Let's quickly run the `procmap` utility on our shell process (which happens to have PID 1076) to see it display the NULL trap page:

```
$ procmap --pid=1076
[...]
```

We can see the NULL trap page in the following screenshot:

*Figure 7.1 – Partial screenshot of the lower portion of the user VAS from the procmmap utility*

You can spot the NULL trap page right at the bottom of the above screenshot (some mappings of the bash process are visible higher up). The NULL trap page works by having all permissions – `rwx` – set to `---` so that no process (or thread) can read or write or execute anything therein! This is why, when a process attempts to read or write the NULL byte at address `0x0`, it doesn't work. Briefly, what actually happens is this:

- A process attempts to access (read/write/execute), or dereference, the NULL byte
  - In fact, accessing *any byte* within this page will lead to this same sequence of events, as the --- mode applies to all bytes within the page; which is

- why it's called the **NULL trap page**! It traps access to any bytes within it
- The permissions for all bytes in the page are zero: no read, no write, no execute; now, all virtual addresses end up at the MMU. The MMU makes its checks and then performs runtime address translation, translating the virtual address to a physical one. Here, the MMU detects the fact that all bytes in the page have no permissions and thus raises a fault (typically on x86, a General Protection Fault)
  - The OS has fault (and trap/exception) handlers preinstalled; control is passed onto the appropriate fault handling function
  - This function – the fault handler – runs in the process context of the process that caused the fault; it, via a rather elaborate algorithm, figures out what the issue is
  - Here it will conclude that a process executing in user mode attempted a buggy access; it thus sends it a fatal signal (`SIGSEGV`); this is what can ultimately lead to the process dying and the `Segmentation fault [(core dumped)]` message showing on the console (of course, the process could install a signal handler to handle this signal; ultimately though, after cleanup, it must terminate).

Now that you understand what exactly the NULL trap page is and it's working, let's do what we're not supposed to: try and read/write the NULL address, causing a bug!

## A simple Oops v1 – dereferencing the NULL pointer

In this, our first simple version of a buggy kernel module, we simply read or write the NULL address. As you just learned in the previous section, any access – read, write or execute – on any byte within the NULL trap page will cause the MMU to jump up and trigger a fault; this remains true in kernel mode as well.

Here's the relevant code snippet for the buggy module (please do clone this book's GitHub repo, browse through and try things yourself!):

```
// ch7/oops_tryv1/oops_tryv1.c
[...]
static bool try_reading;
module_param(try_reading, bool, 0644);
MODULE_PARM_DESC(try_reading,
"Trigger an Oops-generating bug when reading from NULL; else, do so
```

We keep a Boolean module parameter named `try_reading`; it's `0` (or off) by default. If set to `1` (or the value `yes`), the module code will attempt to read the content of the NULL address. If left as `0`, the code, detecting this will instead try to write a byte (`x`) to the NULL address. Here's the code of the initialization function where this is done:

```
static int __init try_oops_init(void)
{
 size_t val = 0x0;
 pr_info("Lets Oops!\nNow attempting to %s something
 %s the NULL address 0x%p\n",
 !!try_reading ? "read" : "write",
 !!try_reading ? "from" : "to", // pedantic, huh
 NULL);
 if (!!try_reading) {
 val = *(int *)0x0;
 /* Interesting! If we leave the code at this, the compiler :
 pr_info("val = 0x%lx\n", val);
 } else // try writing to NULL
 *(int *)val = 'x';
 return 0; /* success */
}
```

It's pretty straightforward. Do read the detailed comment above regarding compiler optimization in the read case and how we can sidestep this.

The key point here, of course, is that both the read and write accesses are buggy – as described in detail in the previous section *What's this NULL trap page anyway*, any attempt to read/write/execute any byte in the NULL trap page is disallowed and results in a fault! Here and now, the kernel module code, running in the process context of the `insmod` process, will perform the buggy access.

Now, think on this, the kernel isn't a process; the fault handler code, upon detecting that a buggy access was made **in kernel mode** (yes, it can and does happen!), realizes that something's dramatically wrong – the kernel is buggy. It thus triggers an Oops!

### What's with the `!!<boolean>` syntax

It's one of the C coding features being taken advantage of: using `!!<boolean_expression>` guarantees the expression evaluates to either `0` or `1`, no matter what value is passed (for example, passing `5` makes it

```
!!(5); now !5 is 0 and !0 is 1. Clever.
```

The partial screenshot below shows just the initial portion of the Oops messages written to the kernel log; worry not, we'll definitely cover the rest and learn how to interpret it in detail. For now just take a look at it; here, in the example shown, we attempted to write into the NULL byte, triggering the Oops:

```
[302.546331] oops_tryv1:try_oops_init():37: Lets Oops!
Now attempting to write something to the NULL address 0x0000000000000000
[302.546351] BUG: kernel NULL pointer dereference, address: 0000000000000000
[302.546374] #PF: supervisor write access in kernel mode
[302.546388] #PF: error_code(0x0002) - not-present page
[302.546402] PGD 0 P4D 0
[302.546411] Oops: 0002 [#1] PREEMPT SMP PTI
[302.546424] CPU: 5 PID: 2903 Comm: insmod Tainted: G OE 5.10.60-prod01 #6
[302.546466] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[302.546489] RIP: 0010:try_oops_init+0xdb/0x1000 [oops_tryv1]
```

*Figure 7.2 – A partial screenshot showing the classic Oops that results when attempting to write to the NULL address (via our oops\_tryv1 module)*

The `Oops:` at the beginning of a kernel log message (do spot it above) denotes the kernel printk's as an Oops diagnostic message.

A perhaps useful (silly) workaround to rebooting

Did you notice: once buggy, the kernel module can't be unloaded (via `rmmmod`) as the reference count is non-zero (`lsmod` verifies this):

```
$ lsmod |grep oops
oops_tryv1 16384 1
```

This is typically as the Oops occurred prior to the process context (`insmod`, in our case) exiting and thus having the module reference count being decremented down to `0`. The `1` at the extreme right of the above output shows that the current module reference count is `1`, preventing the unload of this module.

Now, if you can't unload the module, you can't load it up again (to try it after editing the source file(s)). The correct approach is to reboot the box and start over. A very silly workaround to this problem is to simply clean up (`make clean`), rename the source file to another name, edit the `Makefile` to use the new name and build it. Now it will load up, under the new name! Very silly, but effective when you're in development and in a hurry to try things out.

## Doing a bit more of an Oops – our buggy module v2

As mentioned at the beginning of this chapter, in our version 2 buggy module we'll do a few more, slightly (and hopefully) more realistic things to trigger a kernel Oops. This module has three distinct ways to trigger an Oops:

- One, by writing to a randomly generated kva within the NULL trap page
- Two, by allowing the user to pass a (random) invalid kva and attempting to write something there (you can leverage the `procmap` utility to find an invalid kva)
- Three, we spin up a simple workqueue function; this will have a kernel worker thread run it's code when it's scheduled. Within the workqueue function, we'll trigger an Oops by attempting to write something to a member of a structure where the structure pointer is NULL (as this scenario's a bit realistic, we'll make it a use case pretty much throughout this chapter).

Let's begin by using the first approach mentioned above to trigger an Oops!

### Case 1 – Oops by writing to a random location within the NULL trap page

Being very similar to the first v1 module, I won't delve much into this; it suffices to say that we use a kernel interface (the `get_random_bytes()` API) to generate a random number and scale it down to numbers between 0 and 4095 (by using the modulo operator). The relevant code in the module's init function is seen below:

```
// ch7/oops_tryv2/oops_tryv2.c
[...]
static int __init try_oops_init(void)
{
 unsigned int page0_randptr = 0x0;
 [...]
} else { // no module param passed, write to random kva in NULL trap
 pr_info("Generating Oops by attempting to write to a random\n"
 "get_random_bytes(&page0_randptr, sizeof(unsigned int));\n"
 "bad_kva = (page0_randptr %= PAGE_SIZE);");
 }
 pr_info("bad_kva = 0x%lx; now writing to it...\n", bad_kva);
 *(unsigned long *)bad_kva = 0xdead;
[...]
```

The last line seen above is where we attempt to write into this **bad** kva; this of course triggers an Oops. To try this out, simply `insmod` the module without passing any parameters; this will have the code go to this use case (I'll leave it to you to try it out for yourself and see the kernel log).

## Case 2 – Oops by writing to an invalid unmapped location within the kernel VAS

For this second use case, we have a module parameter named `mp_randaddr`. To run this case, you're to pass it to the module the usual way, setting it to an invalid kernel address (or kva).

```
// ch7/oops_tryv2/oops_tryv2.c
[...]
static unsigned long mp_randaddr;
module_param(mp_randaddr, ulong, 0644);
MODULE_PARM_DESC(mp_randaddr, "Random non-zero kernel virtual address");
```

Now, when the module's init function detects that you've passed a non-zero value in this parameter, it invokes the following code:

```
} else if (mp_randaddr) {
 pr_info("Generating Oops by attempting to write to the invalid
 bad_kva = mp_randaddr;
} else {
 [... << code of the first case above >> ...]
}
pr_info("bad_kva = 0x%lx; now writing to it...\n", bad_kva);
*(unsigned long *)bad_kva = 0xdead;
```

The approach is pretty much identical to the first case; what makes it interesting is this: how will I know which kernel address (or kva) to pass? How will I know it's an invalid (or unmapped) location in the kernel VAS?

Ah, this is where the `procmap` utility comes into play! Simply run `procmap` (passing any PID and specifying the `--only-kernel` option switch, as we're not interested in user VAS now). Here's how I invoked it, for example, on my x86\_64 guest VM (you will need to update the `PATH` environment variable to include the directory where you installed `procmap`):

```
$ procmap --pid=1 --only-kernel
...
```

Here's a partial screenshot of the output it displays, focused on the upper portion of the kernel VAS:

```
[===== P R O C M A P =====]
Process Virtual Address Space (VAS) Visualization utility
https://github.com/kaiwan/procmap

Wed Dec 15 14:55:03 IST 2021
[===== Start memory map for 1:systemd =====]
[Pathname: /usr/lib/systemd/systemd]
+----- K E R N E L V A S end kva -----
|<... K sparse region ...> [8.00 MB,---]
|
+-----+ ffffffffffffffff
| fixmap region [2.52 MB,r--]
|
+-----+ fffffffffff7ff000
|<... K sparse region ...> [5.47 MB,---]
|
+-----+ fffffffffff579000 <- FIXADDR_START
| module region [1008.00 MB,rwx]
|
+-----+ fffffffffff000000 <- MODULES_END
|<... K sparse region ...> [37.78 TB,---]
|
+-----+ ffffffffc0000000 <- MODULES_VADDR
| vmalloc region [31.99 TB,rw-]
|
+-----+ fffffda377fffffff <- VMALLOC_END
```

Figure 7.3 – (Partial) Screenshot showing the procmap utility's output focused on the upper portion of the kernel VAS; some sparse (unmapped) regions are clearly visible

Okay, look carefully at the above screenshot; the regions marked like `<... K sparse region ...>`, these are empty holes in the kernel VAS. There's nothing mapped here; this is quite common, memory like this is often referred to as a *sparse region* or a *hole* in the address space.

The point is this: sparse regions are unmapped regions, thus, if you attempt to

access any of these locations in any manner – read, write or execute – it's a bug! So, let's pick a kva within a sparse region; I'll pick one between the module region (where kernel modules live) and the kernel vmalloc region (where the `vmalloc()` allocates memory from), i.e., any address between `0xfffffffffc000000` and `0xfffffdada377fffff`. So, I'll take the kva `0xfffffffffc000dead` as the value for my invalid kernel address and run with it.

Right; ensure you've built the `oops_tryv2` module, then load it up passing the parameter as just discussed:

```
$ modinfo -p ./oops_tryv2.ko
mp_randaddr:Random non-zero kernel virtual address; deliberately in-
bug_in_workq:Trigger an Oops-generating bug in our workqueue functio-
$
```

We use the `modinfo` utility to show that our module accepts two parameters (please ignore the second one for now, it's our next topic). Let's (finally!) get going:

```
$ sudo insmod ./oops_tryv2.ko mp_randaddr=0xfffffffffc000dead
Killed
$
```

Aha! Our module (deliberately) attempting to write to the invalid kernel address `0xfffffffffc000dead` (passed via the parameter) has it run headlong into a buggy ending. We got what we wanted, an Oops has hit; the (partial) screenshot below shows you a good portion of it:

```

[49132.584848] oops_tryv2:try_oops_init():92: Generating Oops by attempting to write to the invalid kernel address passed
[49132.585606] oops_tryv2:try_oops_init():100: bad kva = 0xffffffffc000dead; now writing to it...
[49132.586023] BUG: unable to handle page fault for address: ffffffff000dead
[49132.586450] #PF: supervisor write access in kernel mode
[49132.586961] #PF: error code(0x0002) - not-present page
[49132.587417] PGD 33c15067 P4D 33c15067 PUD 33c17067 PMD 182d067 PTE 0
[49132.587875] Oops: 0002 [#2] PREEMPT SMP PTI
[49132.588296] CPU: 5 PID: 15255 Comm: insmod Tainted: G D OE 5.10.60-prod01 #6
[49132.588727] Hardware name: innoteck GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[49132.589134] RIP: 0010:try_oops_init+0xf4/0x1000 [oops_tryv2]
[49132.589543] Code: 42 64 0d 00 b9 64 00 00 00 48 c7 c2 d0 93 6d c0 48 c7 c6 3c 90 6d c0 48 c7 c7 90 92 6d
c0 e8 98 35 a8 c6 48 8b 05 1c 64 0d 00 <48> c7 00 ad de 00 00 e9 78 ff ff ff b9 5f 00 00 00 48 c7 c2 d0 93
[49132.590928] RSP: 0018:ffffba3783dfffc20 EFLAGS: 00010246
[49132.591423] RAX: ffffffff000dead RBX: 0000000000000000 RCX: 0000000000000000
[49132.591954] RDX: 0000000000000000 RSI: 0000000000000027 RDI: 00000000ffffffffff
[49132.592398] RBP: fffffba3783dfffc38 R08: 0000000000000000 R09: fffffba3780e9f020
[49132.592864] R10: 0000000000000001 R11: 00000000ffffffffff R12: ffffffc0604000
[49132.593322] R13: fffff8f90766f6530 R14: fffffba3783dfffe70 R15: ffffffc06da158
[49132.593769] FS: 0000785ef7e11540(0000) GS:fffff8f90bdd40000(0000) knlGS:0000000000000000
[49132.594258] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[49132.594711] CR2: ffffffc000dead CR3: 00000005a5e4001 CR4: 00000000000706e0
[49132.595199] Call Trace:
[49132.595739] do_one_initcall+0x48/0x210
[49132.596217] ? kmem_cache_alloc_trace+0x3ae/0x450
[49132.596666] do_init_module+0x62/0x240
[49132.597119] load_module+0x2a04/0x3080
[49132.597596] ? security_kernel_post_read_file+0x5c/0x70
[49132.598078] __do_sys_finit_module+0xc2/0x120
[49132.598648] ? __do_sys_finit_module+0xc2/0x120
[49132.599087] __x64_sys_finit_module+0x1a/0x20
[49132.599549] do_syscall_64+0x38/0x90
[49132.600066] entry_SYSCALL_64_after_hwframe+0x44/0xa9
[49132.600557] RIP: 0033:0x785ef7f5689d
[49132.600987] Code: 00 c3 66 2e 0f 1f 84 00 00 00 00 90 f3 0f 1e fa 48 89 f8 48 89 f7 48 89 d6 48 89 ca
4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d 01 fo ff ff 73 01 c3 48 8b 0d c3 f5 0c 00 f7 d8 64 89 01 48

```

*Figure 7.4 – Screenshot showing the Oops generated by our attempting to write to an invalid kernel address*

I hope the key point is clear: of course we’re going to have a bug, an Oops. We wrote to an invalid unmapped kernel address within a sparse region of the kernel VAS – which procmap literally helped us see.

Why does attempting to access an invalid address cause the Oops? The answer’s very similar to what we discussed regarding the NULL trap page. Here’s what essentially occurs:

1. The virtual address being worked upon (read, written or executed) goes to the MMU
2. The MMU, knowing where the current process context’s paging tables are (for x86, the physical address of the base of the paging tables is in the CR3 register), proceeds to now translate this virtual address (kva) to a physical address (here, we’ll ignore hardware optimizations like the CPU caches and the **Translation Lookaside Buffers (TLBs)**) that might already hold the physical address, thus short-circuiting the lengthy translation and providing

a speed-up)

3. Normally, it will find a mapping and perform the translation, placing the physical address on the bus; the CPU takes over and the work gets done. In this case though, the kernel address passed along is invalid (deliberately so) – it's literally part of a hole in the kernel VAS! So, the address translation fails; the MMU, being hardware, does the best it can: it informs the OS that something's wrong by raising a (page) fault
4. The OS's page fault handler takes over (running in the context of the process that caused the fault; here, it's `insmod` of course). It figures that *an invalid write was attempted while in kernel mode* – it thus triggers an Oops!

(What about understanding and interpreting this messy Oops thingy in detail? That's precisely what we do in the coming section *Devil in the details – decoding the Oops*; hang tight, we'll get there!).

### Case 3 – Oops by writing to a structure member when the structure pointer's NULL

This use (or test) case is a bit more involved, helping making it a bit more realistic as well. The end result's the same as the prior two cases though – we get the kernel to trigger an Oops.

This time, we'd like the buggy code path to run not in the context of `insmod`; to arrange for this to happen, we initialize a (kernel default) workqueue and schedule it, having its code execute. The execution of the kernel default workqueue is done in the context of a kernel worker thread. We arrange for the work function to have a bug – a write to an invalid memory location, a pointer (to a structure) that hasn't been assigned any memory. This of course causes an Oops to trigger. Here's the relevant code snippets (as usual, I urge you to browse the full code and try these things out yourself as well):

```
// ch7/oops_tryv2/oops_tryv2.c
[...]
static bool bug_in_workq;
module_param(bug_in_workq, bool, 0644);
MODULE_PARM_DESC(bug_in_workq, "Trigger an Oops-generating bug in o
```

This time we have a module parameter named `bug_in_workq`, data type Boolean. It's false by default; set it to `1` (or `yes`) to have this use case get underway.

```

static struct st_ctx {
 int x, y, z;
 struct work_struct work;
 u8 data;
} *gctx, *oopsie; /* careful, pointers have no memory! */

```

Above, is a structure we use; notice the pointers to it. In our module's init function, if the `bug_in_workq` parameter is set, we call a function `setup_work()` which sets up some work on the kernel-default workqueue:

```

if (!bug_in_workq) {
[...]
 setup_work();
 return 0;
}

```

The function allocates memory to the `gctx` pointer, calls the `INIT_WORK()` macro to setup work – the function `do_the_work()` - on the kernel-default workqueue:

```

static int setup_work(void)
{
 gctx = kzalloc(sizeof(struct st_ctx), GFP_KERNEL); [...]
 gctx->data = 'C';
 /* Initialize our workqueue */
 INIT_WORK(&gctx->work, do_the_work);
}

```

Next, we call `schedule_work()` on our workqueue to have the kernel actually run the code of our work function:

```

// Do it!
schedule_work(&gctx->work); [...]
}

```

Finally, here's the actual workqueue function that's run (by a kernel worker thread) when the `schedule_work()` triggers; it's buggy, of course (quick, spot the bug!):

```

static void do_the_work(struct work_struct *work)
{
 struct st_ctx *priv = container_of(
 work, struct st_ctx, work);
[...]
if (!bug_in_workq) {
 nr_info("Generating OOPS by attempting to
}

```

```

 write to an invalid kernel
 memory pointer\n");
oopsie->data = 'x';
}
kfree(gctx);
}

```

Well, it's obvious in retrospect; the pointer to our structure named `oopsie` (appropriate huh) has no memory (its value is NULL as it's a global static variable within our module), yet we attempt to try and write into a member of the structure via it. This triggers the Oops; here's how I invoke it:

```
sudo insmod ./oops_tryv2.ko bug_in_workq=yes
```

Did you notice? This time the `Killed` message does not appear; this is as the `insmod` process isn't killed; instead, the kernel worker thread that consumes our workqueue function will suffer the consequences of the bug.

Here's a partial screenshot:

```

[448.049270] oops_tryv2:try_oops_init():87: Generating Oops via kernel bug in workqueue function
[448.049408] oops_tryv2:do_the_work():57: In our workq function: data=67
[448.049409] oops_tryv2:do_the_work():59: delta: 137891 ns
[448.049410] oops_tryv2:do_the_work():59: 137 us
[448.049411] oops_tryv2:do_the_work():61: Generating Oops by attempting to write to an invalid kernel memo
ry pointer
[448.049414] BUG: kernel NULL pointer dereference, address: 0000000000000030
[448.049435] #PF: supervisor write access in kernel mode
[448.049449] #PF: error code(0x0002) - not-present page
[448.049462] PGD 0 P4D 0
[448.049471] Oops: 0002 [#1] PREEMPT SMP PTI
[448.049483] CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G OE 5.10.60-prod01 #6
[448.049504] Hardware name: innoteck GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[448.049547] Workqueue: events_do_the_work [oops_tryv2]
[448.049562] RIP: 0010:do_the_work+0x124/0x15e [oops_tryv2]
[448.049578] Code: c0 e8 d0 1d ad df f6 c3 01 74 27 b9 3d 00 00 00 48 c7 c2 c0 63 5a c0 48 c7 c6 3c 60 5a
c0 48 c7 c7 18 61 5a c0 e8 61 25 0e e0 <c6> 04 25 30 00 00 00 78 48 8b 3d cd 23 00 00 e8 a8 aa 79 df 5b 41
[448.049680] RSP: 0018:fffffb6e1c008be48 EFLAGS: 00010246
[448.049704] RAX: 0000000000000067 RBX: 0000000000000001 RCX: 0000000000000000
[448.049734] RDX: 0000000000000000 RSI: 0000000000000027 RDI: 00000000ffffffff
[448.049775] RBP: ffffb6e1c008be58 R08: 0000000000000000 R09: ffffffff9c988
[448.049801] R10: ffffffff9c3820 R11: 3fffffff9c988 R12: 00000000000021aa3
[448.049827] R13: fffff9ddffdc31700 R14: 0000000000000000 R15: fffff9ddffdc2b9c0
[448.049853] FS: 0000000000000000(0000) GS:ffff9ddffdc00000(0000) knlGS:0000000000000000
[448.049882] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[448.049904] CR2: 0000000000000030 CR3: 000000005f410003 CR4: 00000000000706f0
[448.049934] Call Trace:
[448.049949] process_one_work+0x1b8/0x3b0
[448.049967] worker_thread+0x50/0x3a0
[448.049984] ? process_one_work+0x3b0/0x3b0
[448.050002] kthread+0x154/0x180
[448.050018] ? kthread_unpark+0xa0/0xa0
[448.050034] ret_from_fork+0x22/0x30
[448.050050] Modules linked in: oops_tryv2(OE) intel_rapl_msr snd_intel8x0 snd_ac97_codec intel_rapl_commo

```

Figure 7.5 – Partial screenshot of the Oops triggered by our `oops_tryv2` module due to a bug in our workqueue function

By the way, if required, you can learn the details on setting up and using kernel workqueues, timers and kernel threads in *Chapter 5* of my earlier *Linux Kernel Programming, Part 2* book (the e-book is freely downloadable).

Of course, here we always assume that you do have access to the kernel log (via `dmesg`, `journalctl`, on a safe place on a flash chip, whatever). What if you don't know where the Oops message is in the first place? Well, the kernel community has documented what you can do about this here: *Where is the Oops message is located?* (<https://www.kernel.org/doc/html/latest/admin-guide/bug-hunting.html#where-is-the-oops-message-is-located>). (Also, we shall cover some of the techniques mentioned therein later; `netconsole` is covered in the section *An Oops on an ARM Linux system and using netconsole* and `kdump/crash` is briefly covered in *Chapter 12, A few more Kernel Debug Approaches*).

Okay, you now know how to trigger a kernel bug, an Oops, in several ways! A quick look at what a kernel Oops is and isn't follows.

## A kernel Oops and what it signifies

A quick few things to realize regarding a kernel Oops.

First off, an Oops is not the same as a segfault – a segmentation fault... It might, as a side effect, cause a segfault to occur, and thus the process context might receive the `SIGSEGV` signal. This of course has the poor process caught in the crossfire.

Next, an Oops is not the same thing as a full-fledged kernel panic; a panic implies the system is in an unusable state. It might lead up to this, especially on production systems (we cover kernel panic in *Chapter 10, Kernel Panic, hangcheck and watchdogs*). Note though, that the kernel provides several `sysctl` tunables (editable by root of course) regarding what circumstances can lead to the kernel panicking; we can check them out – on my x86\_64 Ubuntu 20.04 guest running our custom production kernel, here they are:

```
$ cd /proc/sys/kernel/
$ ls panic_on_*
panic_on_io_nmi panic_on_oops panic_on_rcu_stall panic_on_unreco\
```

And, as you can see if you `cat` them, all of their values are zero by default,

implying that a kernel panic will *not* be triggered. It also shows us that setting, for example, the `panic_on_oops` tunable to `1` will cause the kernel to panic on any Oops, no matter how trivial it might seem.

It's important to understand that this can be the right thing to do on many installations. When a system Oops'es, we usually want a bright red flag going up – a way to understand that the system's in (or was in) an unhealthy state! (This does depend on the nature of the project or product: a deeply embedded system might not afford to remain down due to a kernel panic; there, a *watchdog* will typically detect the system's in an unhealthy state and reboot it. We shall cover using watchdogs and what not in *Chapter 10, Kernel Panic, hangcheck and watchdogs*).

Even though an Oops isn't a kernel panic, depending on the circumstances and the severity of the bug, the kernel can be rendered unresponsive or unstable or both. Or it might continue to work as though nothing alarming has occurred! Whatever the case, *an Oops is, ultimately, a kernel-level bug; it must be detected, interpreted and fixed!*

Right; let's get to the juicy bit, learning how to interpret, in detail, the Oops kernel output. Let's go!

## Devil in the details – decoding the Oops

We use the third scenario (or use/test case), covered in the section *Case 3 – Oops by writing to a structure member when the structure pointer's NULL*. To quickly recap, this is what we did to trigger this particular kernel Oops (our case #3):

```
cd ch7/oops_tryv2
make
sudo insmod ./oops_tryv2.ko bug_in_workq=yes
```

As seen earlier, it triggers an Oops. Now we get to the interesting part – deciphering the Oops, step by step, line by line.

Before starting, it's important to realize that the detailed discussion below is necessarily arch-specific, here and now pertaining to the x86\_64 platform (as portions of the Oops output are of course very arch-specific). We shall also show how a typical Oops appears on the ARM platform (both 32 and 64-bit) in a later section.

## Line-by-line interpretation of an Oops

The initial, and really key, portion of the Oops we get is seen in *Figure 7.5*. Now, to help refer to it line-by-line, here's an annotated diagram of the same screenshot (zoomed in a bit more, for clarity):

```
[448.049411] oops_tryv2:do_the_work():61: Generating Oops by attempting to write to an invalid kernel memory pointer
[448.049414] BUG: kernel NULL pointer dereference, address: 0000000000000000
[448.049435] #PF: supervisor write access in kernel mode
[448.049449] #PF: error code(0x0002) - not-present page
[448.049462] PGD 0 P4D 0
[448.049471] Oops: 0002 [#1] PREEMPT SMP PTI
[448.049483] CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G OE 5.10.60-prod01 #6
[448.049504] Hardware name: innotech GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[448.049547] Workqueue: events do_the_work [oops_tryv2]
[448.049562] RIP: 0010:do_the_work+0x124/0x15e [oops_tryv2]
[448.049578] Code: c0 e8 d0 1d ad df f6 c3 01 74 27 b9 3d 00 00 00 48 c7 c2 c0 63 5a c0 48 c7
c6 3c 60 5a c0 48 c7 c7 18 61 5a c0 e8 61 25 0e e0 <c6> 04 25 30 00 00 78 48 8b 3d cd 23 00
00 e8 a8 aa 79 df 5b 41
[448.049680] RSP: 0018:fffffb6e1c008be48 EFLAGS: 00010246
[448.049704] RAX: 0000000000000067 RBX: 0000000000000001 RCX: 0000000000000000
[448.049734] RDX: 0000000000000000 RSI: 0000000000000027 RDI: 00000000ffffffff
[448.049775] RBP: fffffb6e1c008be58 R08: 0000000000000000 R09: ffffffff9c88
[448.049801] R10: ffffffff9c88 R11: 3fffffff9c88 R12: 0000000000021aa3
[448.049827] R13: fffff9ddffdc31700 R14: 0000000000000000 R15: fffff9ddffdc2b9c0
[448.049853] FS: 0000000000000000(0000) GS:ffff9ddffdc0000(0000) knlGS:0000000000000000
[448.049882] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[448.049904] CR2: 0000000000000030 CR3: 000000005f410003 CR4: 00000000000706f0
[448.049934] Call Trace:
[448.049949] process_one_work+0x1b8/0x3b0
[448.049967] worker_thread+0x50/0x3a0
[448.049984] ? process_one_work+0x3b0/0x3b0
[448.050002] kthread+0x154/0x180
[448.050018] ? kthread_unpark+0xa0/0xa0
[448.050034] ret_from_fork+0x22/0x30
[448.050050] Modules linked in: oops_tryv2(OE) intel_rapl_msr snd_intel8x0 snd_ac97_codec intel_rapl_common rapl ac97_bus snd_pcm joydev input_leds serio_raw snd_seq snd_timer snd_seq_device snd soundcore video mac_hid msr parport_pc ppdev lp parport ip_tables x_tables autofs4 hid_generic usbhid hid vmwgfx drm_kms_helper syscopyarea sysfillrect sysimgblt fb_sys_fops crct10dif_pclmul cec crc32_pclmul ghash_clmulni_intel rc_core aesni_intel glue_helper ttm crypto_simd psmouse cryptd drm ahci libahci i2c_piix4 e1000 pata_acpi
[448.050937] CR2: 0000000000000030
[448.051593] ---[end trace cc44ad6c5fd2bc79]---
```

*Figure 7.6 – Annotated (Full) screenshot of the Oops output from our oops\_tryv2 workqueue (case 3) function bug*

Now, for clarity and to make it more approachable, we'll split up this diagram and the discussion into several parts (you can see how we intend the splitting via the rectangles in *Figure 7.6*). Let's begin with the first of them, here it is:

```

[448.049411] oops_tryv2:do_the_work():61: Generating Oops by attempting to write to an invalid kernel memory pointer
[448.049414] BUG: kernel NULL pointer dereference, address: 000000000000000030
[448.049435] #PF: supervisor write access in kernel mode
[448.049449] #PF: error code(0x0002) - not-present page
[448.049462] PGD 0 P4D 0
[448.049471] Oops: 0002 [#1] PREEMPT SMP PTI ← 2
[448.049483] CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G OE 5.10.60-prod01 #6
[448.049504] Hardware name: innoteck GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[448.049547] Workqueue: events do_the_work [oops_tryv2]
[448.049562] RIP: 0010:do_the_work+0x124/0x15e [oops_tryv2]
[448.049578] Code: c0 e8 d0 1d ad df f6 c3 01 74 27 b9 3d 00 00 00 48 c7 c2 c0 63 5a c0 48 c7
c6 3c 60 5a c0 48 c7 c7 18 61 5a c0 e8 61 25 0e e0 <c6> 04 25 30 00 00 00 78 48 8b 3d cd 23 00
00 e8 a8 aa 79 df 5b 41 5

```

Figure 7.7 – Annotated screenshot 1 of 3 : Oops output from our oops\_tryv2 workqueue function bug

Okay, lets delve into the details! (Also, the code shown from here on is arch-specific and applies *only* to the x86 platform).

### Interpreting Oops line(s) 1

The brightly visible lines painted in a red background color emanate from this code (it's arch-specific, this is for the x86\_64); it's a portion of the code of the OS fault handler, the code that, when it's detected an abnormal condition within the kernel, a bug, begins to write the Oops diagnostic message:

```

arch/x86/mm/fault.c
static void
show_fault_oops(struct pt_regs *regs,
 unsigned long error_code,
 unsigned long address)
{
 [...]
 if (address < PAGE_SIZE && !user_mode(regs))
 pr_alert("BUG: kernel NULL pointer
 dereference, address: %px\n",
 (void *)address);
 else
 pr_alert("BUG: unable to handle page fault
 for address: %px\n", (void *)address);
}

```

Above, check out the if condition; it's now amply clear why we got this output:

BUG: kernel NULL pointer dereference, address: 0000000000000030

It's emitted when the faulting address is within the first page and we're running in kernel mode. Recall, the very first page of the user VAS is the NULL trap page, one where all addresses are within `PAGE_SIZE` (typically 4096 bytes). If the condition is false, the kernel prints an alternate message. Further, here, the address that caused the fault – the one within the first NULL trap page – is then printed (it's always in hexadecimal); here it's the value `0x30`.

Now this too is important: why `0x30` and not `0x0`? Think back to the code that generated this particular Oops (you can refer back to the section *Case 3 – Oops by writing to a structure member when the structure pointer's NULL* to see this). The buggy line of code is here:

```
ch7/oops_tryv2/oops_tryv2.c:do_the_work():oopsie->data = 'x';
```

Now, `oopsie` is a pointer to the `st_ctx` structure in our code, but it's value is `NULL` (recall, it was never allocated). So, the value of `0x30` is the offset from the beginning of the structure to the member being referenced! A key point we thus learn is: *when the faulting address shows up as a small integer value within the size of a page (as is the case here), it's very likely that the structure (or other) pointer was NULL and the number displayed is the offset from the beginning of the structure (or whatever) to the member being referenced.*

The next line of output in the Oops is:

```
#PF: supervisor write access in kernel mode
```

This is generated from the code that continues (in the same `show_fault_oops()` function; by the way, `PF` stands for Page Fault):

```
pr_alert("#PF: %s %s in %s mode\n",
(error_code & X86_PF_USER) ?
 "user" : "supervisor",
(error_code & X86_PF_INSTR) ?
 "instruction fetch" :
(error_code & X86_PF_WRITE) ? "write access" :
 "read access",
user_mode(regs) ? "user" : "kernel");
```

Take the trouble to read the code and match it with the output we obtained; it clearly shows us that the kernel figured a lot out: the code was executed in *supervisor* mode (which means kernel mode), there was a *write* attempt, again

executing in *kernel* mode.

The line after that:

```
#PF: error_code(0x0002) - not-present page
```

Is from the code that immediately follows the code seen just above:

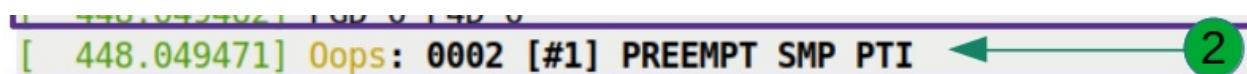
```
pr_alert("#PF: error_code(0x%04lx) -\n %s\\n", error_code,\n !(error_code & X86_PF_PROT) ? "not-present page" :\n (error_code & X86_PF_RSVD) ? "reserved bit violation" :\n (error_code & X86_PF_PK) ? "protection keys violation" : "perm:\n %s\\n", error_code);
```

So, there, we figured out how each of these three lines of output came to be. How did the color show up as red background? Ah, that's easy: `dmesg` interprets the `pr_alert()` log level and colors it accordingly.

(We'll skip the details on the **Page Global Directory (PGD)** and *P4D* (a level inserted between the PGD and **Page Upper Directory (PUD)** in 4.11 Linux) these are references to the paging tables of the process context the process was running in. See the code of the `dump_pagetables()` function if you're interested).

## Interpreting Oops line(s) 2

(For your convenience, this portion of the screenshot is duplicated from *Figure 7.7* below). The next line of output (line 2) in the Oops is:



[ 448.049471] Oops: 0002 [#1] PREEMPT SMP PTI ← 2

*Figure 7.8 – Line(s) 2 of the Oops output from our buggy oops\_tryv2 module, test case 3*

Clearly, this tells us that an Oops has occurred (it's not so ridiculous, grepping for the string `Oops:` can be useful). The number immediately following this string – here, it's `0002` – is important; it's the arch-specific Oops bitmask; learning to interpret it will definitely help.

Interpreting the (arch-specific) Oops bitmask

The overall function responsible for displaying the Oops content from here on is named `arch/x86/kernel/dumpstack.c:__die()`. It's split into two portions – the `__die_header()` and the `__die_body()` functions. The Oops bitmask and remaining tokens on that line (like `PREEMPT SMP ...`) are displayed from the header function; to give you a sampling of how the actual Oops-diagnostic-display kernel code works, here's a screenshot of the `__die_header()` function (for the 5.10.60 kernel). Refer to `Arch/x86/kernel/dumpstack.c`:

```
static void __die_header(const char *str, struct pt_regs *regs, long err)
{
 const char *pr = "";

 /* Save the regs of the first oops for the executive summary later. */
 if (!die_counter)
 exec_summary_regs = *regs;

 if (IS_ENABLED(CONFIG_PREEMPTION))
 pr = IS_ENABLED(CONFIG_PREEMPT_RT) ? "PREEMPT_RT" : "PREEMPT";

 printk(KERN_DEFAULT
 "%s: %04lx [%#d] %s%s%s%s\n", str, err & 0xffff, ++die_counter,
 pr,
 IS_ENABLED(CONFIG_SMP) ? " SMP" : "",
 debug_pagealloc_enabled() ? " DEBUG_PAGEALLOC" : "",
 IS_ENABLED(CONFIG_KASAN) ? " KASAN" : "",
 IS_ENABLED(CONFIG_PAGE_TABLE_ISOLATION) ?
 (boot_cpu_has(X86_FEATURE_PTI) ? " PTI" : " NOPTI") : "");
}

NOKPROBE_SYMBOL(__die_header);
```

*Figure 7.9 – Screenshot of a part of the kernel's Oops output functionality on the x86\_64*

As mentioned above, the **arch-specific Oops bitmask** is actually very meaningful, further clueing us in as to why the kernel bug occurred!

You can see it being emitted from the above `printk` (it's the second parameter). How do we interpret this bitmask? Here's how – but, again, remember, it's arch-specific; this interpretation applies *only* to the x86 platform.

The MMU sets up a page fault error as an encoded value. On the x86 platform, this is how the encoding of the **page fault error code bits** is done:

```
#:# -- #:# no page found 1: protection fault
```

```

bit 0 == 0. no page found 1. protection fault
bit 1 == 0: read access 1: write access
bit 2 == 0: kernel-mode access 1: user-mode access
bit 3 == 1: use of reserved bit detected
bit 4 == 1: fault was an instruction fetch

```

(This information in fact used to be a comment in the codebase in earlier kernel versions).

A nicer way, perhaps, to more easily visualize and thus interpret the particular error – the reason why the Oops occurred – is to examine the LSB five bits of the page fault error code in a tabular format:

| <b>Value Bit 4<br/>of bit</b> | <b>Bit 3</b>      | <b>Bit 2</b> | <b>Bit 1</b> |
|-------------------------------|-------------------|--------------|--------------|
| 0 -na-                        | -na-              | kernel mode  | Read at      |
| 1 Instruction fetch fault     | Reserved bit used | user mode    | Write ↗      |

Table 7.1 – The meaning of the LSB 5 bits of the page fault error code on the x86 platform

So, now it's easy! We got the Oops bitmask as `0002` (it's in hexadecimal; see the `printf`: the format specifier is `%04lx`). This translates to `00010` in binary; by the table above this implies (here as bits 3 and 4 are zero, they don't matter):

```

Bit 2 = 0 : kernel mode
Bit 1 = 1 : write attempt
Bit 0 = 0 : no page found

```

Well, well, no surprise there; this is *exactly* what the Oops diagnostic tells us (point 1 in *Figure 7.7*):

```

#PF: supervisor write access in kernel mode
#PF: error_code(0x0002) - not-present page

```

The remainder of the line is as follows:

```
... [#1] PREEMPT SMP PTI
```

This line is easy to interpret:

- [#1] : It's the number of the Oops that has occurred during this system session; #1 tells us it's the first Oops (it's a session value; a power cycle

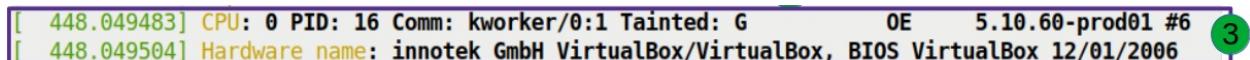
resets it)

- PREEMPT : The code was running on a kernel configured for preemption (`CONFIG_PREEMPT=y`)
- SMP : The kernel has **Symmetric Multi-Processing (SMP)** enabled; it supports multicore
- PTI : PTI is short for **Page Table Isolation**; the **Meltdown/Spectre** hardware bugs circa early 2018 had the kernel developers build a protection mechanism against this serious vulnerability called PTI (see the *Further reading* section for more on this).

Let's move on to interpreting the following two lines within the Oops diagnostic.

### Interpreting Oops line(s) 3

For your convenience, this portion of the screenshot is duplicated from *Figure 7.7* below:



```
[448.049483] CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G OE 5.10.60-prod01 #6
[448.049504] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
```

*Figure 7.10 – Line(s) 3 of the Oops output from our buggy oops\_tryv2 module, test case 3*

The first of these lines essentially informs us as to the process context – the process or thread that executed the buggy code in kernel mode, that caused the fault (along with a few other details). We'll take it a token at a time:

- CPU : Denotes the CPU core the code was running on at the time of the Oops (here it's `CPU 0`)
- PID : The PID of the process or thread that was executing the code at the time of the Oops
- Comm : The name of the process or thread that was executing the code at the time of the Oops
- Tainted : The kernel tainted flags bitmask; we cover this shortly
- The output of `uname -r`, the kernel release, followed by a number prefixed with the `#` symbol; this is the number of times this kernel has been built (here, it's `6`).

It's important to note, though, that when the Oops is triggered *from an interrupt context*, parts of the data seen here become suspect (see more on this in the

upcoming section *Leveraging the console device to get the kernel log* where we examine the Oops when triggered in interrupt context).

### Interpreting the kernel tainted flags

The Linux kernel community likes to know if the running kernel is clean or dirty; a *dirtied* kernel, a *tainted* kernel (the word *tainted* means polluted) is one that isn't in a pristine state. This state information is kept as bits within a bitmask; the entire bitmask, currently consisting of a total of 18 bits or flags, is called the tainted flags.

In our particular use case, the tainted flags show up like so, following the string `Tainted:` (highlighted below):

```
CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G OE 5.10.6
```

You can interpret the letters – the tainted flags, as they're called – as shown here:

| Bit # | Logged as: | Meaning if bit is set (1)             |                                                                         |
|-------|------------|---------------------------------------|-------------------------------------------------------------------------|
|       |            | When bit is Cleared (_)<br>or Set (X) |                                                                         |
| 0     | G or P     |                                       | A Proprietary module was loaded (P); or, only GPL'ed modules loaded (G) |
| 1     | _ or F     |                                       | Forced loading of one or more modules                                   |
| 2     | _ or S     |                                       | Out of specification system                                             |
| 3     | _ or R     |                                       | Forced unloading of one or more modules                                 |
| 4     | _ or M     |                                       | Machine Check Exception (MCE) reported by a CPU core                    |
| 5     | _ or B     |                                       | Bad page referenced, or,                                                |

|    |        |                                                                    |
|----|--------|--------------------------------------------------------------------|
|    |        | unexpected page flags                                              |
| 6  | _ or U | User space app requested taint to be set                           |
| 7  | _ or D | The kernel Died recently (due to an Oops or BUG() )                |
| 8  | _ or A | ACPI table overridden by user                                      |
| 9  | _ or W | A Warning was issued (via one of the WARN*() macros) by the kernel |
| 10 | _ or C | A staging (experimental) driver was loaded                         |
| 11 | _ or I | Platform firmware bug, workaround applied                          |
| 12 | _ or O | Out-of-tree / externally built module was loaded                   |
| 13 | _ or E | An unsigned module was loaded                                      |
| 14 | _ or L | A soft Lockup occurred                                             |
| 15 | _ or K | Live patch applied on the Kernel                                   |
| 16 | _ or X | Distros use this flag, called auxiliary taint                      |
| 17 | _ or T | The kernel was built with structure randomization enabled          |

Table 7.2 – Interpreting the kernel tainted flags

The \_ symbol in the second column implies a blank, a space to indicate that that particular taint bit is cleared (notice how the tainted flags are printed in the Oops with blank spaces as required, to show that a particular bit(s) is/are unset:  
Tainted: G O E ).

So, in our use case, the G|O|E tainted flags imply: all GPL’ed modules ( G ), an externally-built (*out-of-tree*) module was loaded ( O ), an unsigned module was loaded ( E ). Indeed, our oops\_tryv2 module has a dual license that includes

GPL, is an out-of-tree one and is unsigned.

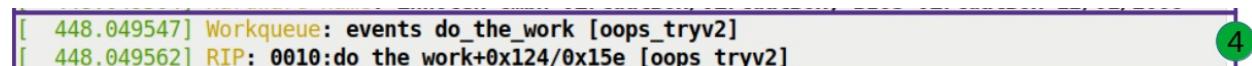
So, it's easy to lookup the table and figure out what the tainted flags imply. It's even easier to employ a helper script that does the work for you! – that's exactly what the `tools/debugging/kernel-chktaint` script (within the kernel source tree) is designed to do. We cover using this script in the section *Are we clean? The kernel-chktaint script*.

The official kernel documentation covers these flags (along with deeper details) here: <https://www.kernel.org/doc/html/latest/admin-guide/tainted-kernels.html>.

The second line (as seen in *Figure 7.10*) is simply some hardware platform details; useful.

## Interpreting Oops line(s) 4

For your convenience, this portion of the screenshot is duplicated from *Figure 7.7* below:



```
[448.049547] Workqueue: events do_the_work [oops_tryv2]
[448.049562] RIP: 0010:do_the_work+0x124/0x15e [oops_tryv2]
```

*Figure 7.11 - Line(s) 4 of the Oops output from our buggy oops\_tryv2 module, test case 3*

In this particular case, let's begin with interpreting the second of the two lines seen above first – the one beginning with `RIP:` .

Finding the code where the Oops occurred

Perhaps the key line in the Oops output is this:

```
RIP: 0010:do_the_work+0x124/0x15e [oops_tryv2]
```

Let's interpret it a token at a time; again, I remind you, much of this is very arch-specific (here, for the x86\_64 platform/CPU):

- `RIP:` : This, of course, is the name of the CPU register that holds the address of the text (code) to execute; on the x86 platform, it's called the **Instruction Pointer** register. On the x86\_64, it's a 64-bit register (in order

to hold 64-bit virtual addresses) and is named `RIP`. So, what is it holding? Read on!

- `0010` : On the x86, the notion of a hardware segment still exists (it's mostly historical and continues in a vestigial form even on the modern 64-bit x86 today). This value, `0010`, represents the code segment; makes sense, the CPU core's `RIP` register should be pointing to code
- `do_the_work+0x124/0x15e [oops_tryv2]` : Perhaps the *most important* thing here; this is the format being employed above:

`function_name+off_from_func/size_of_func [module-name]`

It identifies which function was being executed on the processor at the time of the Oops. Following it is a `+` sign followed by two numeric (hexadecimal) values (so, in the form `+x/y`):

- The first number (`x`) is the offset (in bytes) from the beginning of the function; in other words, it references the actual byte of machine code that was being executed at the time of the Oops!
- The second number (`y`) is the (kernel's best guess of) the size of the function (in bytes)

Next, if this string – `funcname+x/y` – is followed by a name in square brackets (of the form `[modulename]`), that is the kernel module where this function (`funcname`) resides! If not, the function is part of the kernel image itself.

So, for our particular Oops, we can now conclude that the bug (likely) occurred in the function `do_the_work()` which belongs to the module `oops_tryv2`; further, the instruction pointer was `0x124` (decimal 292) bytes from the start of this function; the size of the function, as estimated by the kernel (and it's typically dead right) is `0x15e` (decimal 350) bytes.

*This information – the precise location of the CPU Instructor Pointer register – can be, and often is, critical in figuring out exactly where in the code the bug occurred!*

So, now that we know more-or-less the exact location in the code where the Oops occurred, how exactly do we leverage this information: 292 bytes from the start of `oops_tryv2:do_the_work()` – to find the C source line where the issue, and likely the root cause, lies?

Ah, that's the crux of it, isn't it! We cover precisely this in-depth in the upcoming section *Tools and techniques to help determine the location of the Oops*. It's very important to read the detailed discussion there on the usage of various tools, technologies, and indeed, kernel helper scripts to literally pinpoint the buggy code's location in the (module or kernel) source.

Is this code location (the `funcname+x/y`) always guaranteed to be the root cause of the defect, the bug? No. There are no guarantees! With the more difficult, subtle bugs, the root cause can be miles away; the symptoms have shown up here perhaps. Debugging these is where you really earn your money. With the really wide breadth of tools, techniques and internal details that this book covers, you'll be in a much better position to do so.

In fact, this use case (our `oops_tryv3` case 3 bug) is itself a bit involved; the bug occurred not in the context of the `insmod` process but rather, within the kernel-default `events` workqueue's **kthread (kernel thread)** worker. This information is in fact revealed by the first line in *Figure 7.11*:

```
Workqueue: events do_the_work [oops_tryv2]
```

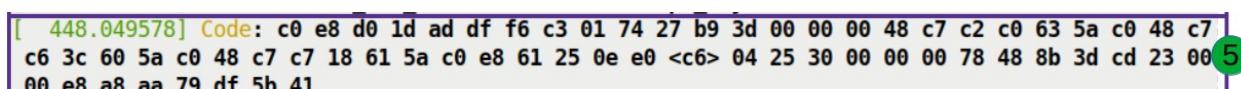
The word `events` in the line above is significant; it's the name of the `workqueue: events` is the name of the kernel-default workqueue.

By the way, the kernel has a `printk` format specifier to print function pointer symbols in this common and useful format: `func+x/y`; the format specifiers include `%pF`, `%pS[R]` and `%pB` (do refer the kernel documentation here for the gory details: <https://www.kernel.org/doc/Documentation/printk-formats.txt>).

Let's continue along the line-by-line exploration of the Oops output.

## Interpreting Oops line(s) 5

For your convenience, this portion of the screenshot is duplicated from *Figure 7.7* below:



```
[448.049578] Code: c0 e8 d0 1d ad df f6 c3 01 74 27 b9 3d 00 00 00 48 c7 c2 c0 63 5a c0 48 c7
c6 3c 60 5a c0 48 c7 18 61 5a c0 e8 61 25 0e e0 <c6> 04 25 30 00 00 00 78 48 8b 3d cd 23 00 5
00 e8 a8 aa 79 df 5b 41
```

*Figure 7.12 – Line(s) 5 of the Oops output from our buggy `oops_tryv2` module, test case 3*

The bytes following the string `Code:` are indeed just that – the machine code running on the CPU core at the time of the Oops! Decoding this machine code is now automated via kernel helper scripts (both `scripts/decodecode` and `scripts/decode_stacktrace.sh`); please refer our coverage in the section *Interpreting machine code with the decodecode script*.

We now move onto the next portion of our Oops test case interpretation.

## Interpreting Oops line(s) 6

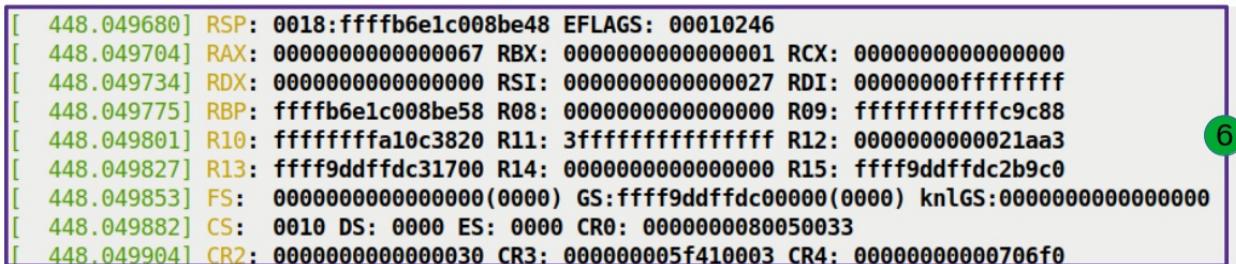
It's the processor registers and their runtime values.

CPU registers and the Oops

The Oops diagnostic includes the value of all general-purpose CPU registers on the processor core that ran the buggy code at the time the Oops occurred.

How is this useful? Recall our quite detailed discussion on the processor **Application Binary Interface (ABI)** (if you're hazy on it do refer back to *Chapter 4, Debug via Instrumentation – Kprobes* section *Understanding the basics of the Application Binary Interface (ABI)*). This can be crucial to understanding what happened at the level of the bare metal.

For your convenience, this portion of the screenshot is duplicated from *Figure 7.7* below:



A screenshot of a terminal window displaying the Linux kernel's Oops output. The output shows various CPU register values in hex format. A green circle with the number '6' is drawn around the first few lines of the output, specifically around the RSP, RAX, RDX, RBP, and R10 registers. The output is as follows:

```
[448.049680] RSP: 0018:fffffb6e1c008be48 EFLAGS: 00010246
[448.049704] RAX: 0000000000000067 RBX: 0000000000000001 RCX: 0000000000000000
[448.049734] RDX: 0000000000000000 RSI: 0000000000000027 RDI: 00000000ffffffff
[448.049775] RBP: fffffb6e1c008be58 R08: 0000000000000000 R09: ffffffffffffc9c88
[448.049801] R10: ffffffffffffa10c3820 R11: 3fffffffffffffff R12: 0000000000021aa3
[448.049827] R13: fffff9ddffdc31700 R14: 0000000000000000 R15: fffff9ddffdc2b9c0
[448.049853] FS: 0000000000000000(0000) GS:fffff9ddffdc00000(0000) knlGS:0000000000000000
[448.049882] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[448.049904] CR2: 0000000000000030 CR3: 000000005f410003 CR4: 00000000000706f0
```

Figure 7.13 – Line(s) 6 of the Oops output from our buggy oops\_tryv2 module, case 3, showing the CPU register values (in hex) at the time of the Oops

The first register to be shown is `RSP` – obviously, it's the stack pointer register. Being able to access the (kernel) stack and interpret the frames therein is crucial,

again, to understanding why and where the Oops occurred. (Here, the top of the kernel mode stack happens to be the kernel virtual address

`0xfffffb6e1c008be48`). Remember, pretty much all arch's (CPU's) follow the *stack-grows-down* semantic, so the top of the stack is really the lowest virtual address on it.

The `EFLAGS` register content follows; it contains the value of various state flags on the processor (for example, the sign flag, carry flag, irq-enable flag, and so on. See a link in the *Further reading* section for a detailed look at the x86 registers). (Also, look at the value of the **CS – Code Segment** – register: it's `0x0010`; as we saw, this is the value prefixed to the instruction pointer).

As an example, for our particular Oops, some of the x86 **control registers** are (perhaps) useful to lookup; let's check them out.

The control registers on the x86\_64

The x86\_64 has sixteen control registers, named `CR0` through `CR15`. Of these, eleven are reserved (`CR1`, `CR5-CR7`, `CR9-CR15`). We mention the few that the x86\_64 kernel Oops diagnostic designs to reveal (and are thus meaningful):

- `CR0` : can be programmed (only in kernel mode); contains control bits like protected mode enable, emulation, write protect, alignment mask, cache disable, paging, and so on
- `CR2` : contains the kva, which when accessed, caused the MMU to raise the page fault, that led to the Oops. Hence, the `CR2` content is a key value! (Look it up in *Figure 7.13*; here `CR2` contains the value `0x30`, which, as you'll recall, is the offset from the beginning of the structure we looked up – in the line of code that caused this Oops: `oopsie->data = 'x'`;
- `CR3` : holds (among other stuff) the physical address of the base of paging tables (called **PML4**) for the process context running at the time of the Oops; in effect, it tells the MMU how to get to the paging tables
- `CR4` : various control bits (for example, the V86 mode extension, debug extensions, page size extension, **Physical Address Extension (PAE)** bit, **Performance Monitoring Counter Enable (PCE)** bit, security feature bits like **Supervisor Mode Executions Protection Enable (SMEP)**, **Supervisor Mode Access Protection Enable (SMAP)**, and so on).

What about CPU register values earlier in the execution path? Read the callout below for more on this .

[View full image](#)

## CPU register values on other call frames

So, think on this: the CPU register values we saw above (*Figure 7.13*) are those that were on-CPU at the time the Oops occurred. But what about the CPU register values on all the other call frames (on the kernel mode stack) that led up to the crash or Oops? Aren't they important? Yes indeed; in fact, their values might be just what's required to decipher the really deep details of how the bug occurred, the root cause. But the Oops diagnostic doesn't reveal them – it's limited to the top-of-the-stack: the function and register values in place at the instant the Oops occurred. So, is there any way to get to the deeper details, to be able to examine all the call frames and register values? Yes: the **Kdump** kernel feature, when enabled, allows saving all pertinent data (in fact it lets you access the entire kernel memory segment!). Along with the powerful user space **crash** app to investigate the dump, you're in business! (We briefly cover Kdump/crash in this book's last chapter).

Right, let's move to interpreting the last portion of the Oops diagnostic.

### Interpreting Oops line(s) 7, 8, 9

For your convenience, this portion of the screenshot is duplicated from *Figure 7.6* below:

The screenshot shows the terminal output of an Oops error. The output is as follows:

```

[448.049934] Call Trace:
[448.049949] process_one_work+0x1b8/0x3b0
[448.049967] worker_thread+0x50/0x3a0
[448.049984] ? process_one_work+0x3b0/0x3b0
[448.050002] kthread+0x154/0x180
[448.050018] ? kthread_unpark+0xa0/0xa0
[448.050034] ret_from_fork+0x22/0x30
[448.050050] Modules linked in: oops_tryv2(0E) intel_rapl_msr snd_intel8x0 snd_ac97_codec int
el_rapl_common rapl ac97_bus snd_pcm joydev input_leds serio_raw snd_seq snd_timer snd_seq_devi
ce snd soundcore video mac_hid msr parport_pc ppdev lp parport ip_tables x_tables autofs4 hid_
eneric usbhid hid vwgfx drm_kms_helper syscopyarea sysfillrect sysimgblt fb_sys_fops crct10di_
_pclmul cec crc32_pclmul ghash_clmulni_intel rc_core aesni_intel glue_helper ttm crypto_simd ps
mouse cryptd drm ahci libahci i2c_piix4 e1000 pata acpi
[448.050937] CR2: 0000000000000030
[448.051593] ---[end trace cc44ad6c5fd2bc79]---

```

The output is annotated with three green circles numbered 7, 8, and 9:

- Circle 7 points to the number 7 at the end of the call trace line.
- Circle 8 points to the number 8 at the end of the modules linked in line.
- Circle 9 points to the number 9 at the end of the CR2 value line.

*Figure 7.14 – Line(s) 7, 8, 9 of the Oops output from our buggy oops\_tryv2 module, case 3, showing the call (stack) trace, modules and CR2 value*

The really key thing to see and interpret here is the call (or stack) trace. Let's do

The really key thing to see and interpret here is the call (or stack) trace. Let's get to it!

#### Interpreting the call stack within the Oops

The lines that are below the string `Call Trace:` and slightly indented to the right represent the call or stack trace. This, of course, is the kernel mode stack of the process context (though it could also be an interrupt! More on this follows shortly...) that caused the Oops.

This call trace is very valuable to the development team, to you, debugging an Oops; why? It literally shows you the history – *how we got to where we did*, to the bug. So how does one interpret it? A few key points follow:

- Firstly, each (slightly right indented) line below `Call Trace:` represents a function in the call path, abstracted by a call frame
- Each frame has the same notation as we saw earlier: `funcname+x/y`; where `x` is the distance from the start of the function in bytes, and `y` is the size of the function
- Read it bottom-up, of course (the vertical arrow in *Figure 7.14* points up to show this); recall, pretty much all modern processors follow the *stack-grows-down* semantic
- Ignore any line that begins with a question mark symbol `?`; this implies the call frame is probably *invalid*, a stale leftover from an earlier trace, perhaps. The kernel's stack unwind algorithm (there are several, they're even configurable!) is smart enough to pretty much guarantee getting this right; so, trust it.

Putting together what we learned, this is the call sequence that led to our Oops:

```
ret_from_fork --> kthread --> worker_thread --> process_one_work
```

Now, of course, understanding this properly requires at least a basic understanding of what our code was doing when the Oops occurred. We do know that the Oops was actually triggered in the function `do_the_work()`; refer back to our notes in the section *Finding the code where the Oops occurred*. This function was our custom workqueue routine. Now how did it get invoked? Ah, indirectly, when our module invoked the `schedule_work()` API passing the pointer to our work structure (glance back at the code, `ch7/oops_tryv2/oops_tryv2.c` if you need to). This work function was

serviced by the kernel's default events workqueue. The servicing – which really means the consuming or execution of our work function – is done by spinning up (or using an existing) kernel worker thread (that belongs to the kernel default workqueue).

The `kthread()` routine – that you see in the call sequence just above – is an internal kernel interface that performs this task, spinning up a kernel thread. It invokes the kernel workqueue function `worker_thread()` whose job it is to process all work items on the workqueue. This is in turn done by calling each work item in a loop, by within it calling the function `process_one_work()`, whose sole job is to process the one work item it's given – ours!

So there we are – the kernel stack does indeed reveal exactly how we got to our worker routine `do_the_work()`. Whew, it can be painstaking work! But, then again, you're the forensic detective at work; no one said it's easy!

### **Tip – seeing the kernel mode stack of all CPU cores**

The kernel has a tunable, `/proc/sys/kernel/oops_all_cpu_backtrace`, whose default value is `0`, off. Turning it on (by writing `1` to it as root), will have the call stacks for all CPU cores on the system displayed as part of the Oops diagnostic. This can be very useful in a deep debug scenario.  
(Internally, the call tracing on other CPU cores is done via the **Non Maskable Interrupt (NMI)** backtrace facility).

Don't forget: the line following the call trace – the one starting with the string `Modules linked in:` - displays all modules loaded in the kernel at the time of the Oops. Why? Modules are very often third-party code (for device drivers, custom network firewall rules, custom filesystems, and so on); hence, they are highly suspect when the kernel encounters a bug! Hence the list of all modules. Indeed, our very own buggy module, `oops_trv2`, stands proudly first in this list (as it's the last loaded one. Also, the tainted flags `OE` denote that it's an out-of-tree unsigned module).

Finally, literally the last line in the Oops diagnostic is what the kernel developers call the *executive summary* – the value of the `CR2` register! By now you should realize why: it's the faulting (virtual) address, access to which caused the Oops.

### **Exercise**

Run this same Oops-ing test case ( `ch7/try_oopsv2` ) with passing the module parameter `bug_in_workq=yes` on your custom *debug* kernel and see what happens. (On mine, with KASAN enabled, the kernel does Oops but not before KASAN catches the bug as well!).

Finally, we're done interpreting our Oops in great detail, literally line-by-line, learning aplenty along the journey (hey, it's the journey that matters, not the destination).

We'd like to also mention (in passing) that there exist several frameworks to help capture a kernel Oops and report them back to the vendor (or distributor). Among these is the `kerneloops(8)` program (man page: <https://linux.die.net/man/8/kerneloops>). Many modern distributions use the **Kdump** feature to collect the entire kernel memory image on an Oops or panic occurring, for later analysis (typically via the `crash` app).

Great; now, it's important to continue onto the following section, where we cover how exactly you can use various tools and techniques to uncover the buggy line(s) of code!

## Tools and techniques to help determine the location of the Oops

While analyzing a kernel Oops, we can certainly use all the help we can get, right. There are several tools and helper scripts that can be leveraged; among them, and part of the (cross) toolchain, are the `objdump`, GDB and `addr2line` programs. Besides them, a few kernel helper scripts (found within the kernel source tree) can prove very useful as well.

In this section, we'll start learning how to exploit these tools to help interpret an Oops.

### **Tip – getting the unstripped vmlinux kernel image with debug symbols**

Many, if not most, of the tools and techniques to help debug kernel issues do **depend on your having an unstripped uncompressed vmlinux kernel image with debug symbols**. Now, if you've built both a debug and

production kernel – as we've recommended from literally the outset of this book! - you'll of course have the debug `vmlinux` kernel image file (which fulfills this requirement).

If not? Well, pretty much all enterprise (and desktop) Linux distros provide a package – separate ones for the commonly used kernel versions that get integrated into the distro – which will provide it (and more). Often, the package is named `linux-devel*` or `linux-headers*`; it's essentially just a compressed archive that contains the kernel headers, the unstripped `vmlinux` with debug symbols, and possibly more goodies within it. Download, extract and see for yourself.

For example, a note on the `kernel-*` RPM's for RedHat RHEL 8 can be found here ([https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/8/html/managing\\_monitoring\\_and\\_updating\\_the\\_linux-kernel-rpm/managing-monitoring-and-updating-the-kernel#the-linux-kernel-rpm-package-overview\\_the-linux-kernel-rpm](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_monitoring_and_updating_the_linux-kernel-rpm/managing-monitoring-and-updating-the-kernel#the-linux-kernel-rpm-package-overview_the-linux-kernel-rpm)). As well, here's a link to download RPM packages for the `kernel-devel` Linux packages for the following Linux distros – AlmaLinux, ALT Linux, CentOS, Fedora, Mageia, OpenMandriva, openSUSE, PCLinuxOS and Rocky Linux: <https://pkgs.org/download/kernel-devel>.

Next, it's important to realize that, when debugging an Oops on a non-native arch (for example, arm, arm64, powerpc, and so on), **you'll need to run the cross-toolchain version** of the tools we examine and use below. (As a concrete example, if you're debugging an Oops that occurred on an ARM-32, compiled with the `arm-linux-gnueabihf-` toolchain prefix (this would be the value of the environment variable `CROSS_COMPILE`), then you'd need to run not `objdump` but  `${CROSS_COMPILE}objdump`, I.e., `arm-linux-gnueabihf-objdump`. The same goes for the other tools in the toolchain (like GDB, `readelf`, `addr2line`, and so on)).

Okay, let's begin using these tools!

## Using objdump to help pinpoint the Oops code location

First off, you'll gain the maximum benefit from these tools by (re)building your buggy kernel, or kernel module, with `CONFIG_DEBUG_INFO=y`. In other words, boot from a debug kernel and build your module(s) there.

In the so-called better `Makefile`, set the variable

```
MYDEBUG := y
```

(it's left as `n` by default). Now rebuild the module; due to debug symbols and information embedded within it, its size is typically (much) larger than for the non-debug or production variant.

The `objdump` utility has the intelligence to deeply examine and interpret an **Executable and Linker Format (ELF)** object file (which includes the kernel `vmlinux` uncompressed image as well as a kernel module image, the `.ko` file). We'll typically run `objdump` with the `-dS` option switches to disassemble the module and intermix source code with assembly wherever possible (which is possible when it's compiled with `-g`)! If you're running live (I.e., the module is still loaded in kernel memory), you can try this:

```
$ grep oops_tryv2 /proc/modules
oops_tryv2 16384 0 - Live 0x0000000000000000 (0E)
```

Due to security reasons, to prevent info-leaks, the address isn't shown. Run as root to see it:

```
$ sudo grep oops_tryv2 /proc/modules
oops_tryv2 16384 0 - Live 0xfffffffffc0604000 (0E)
$ objdump -dS --adjust-vma=0xfffffffffc0604000 ./oops_tryv2.ko > oops
```

With the correct VMA object address of our module specified, `objdump` is able to display the full kernel virtual address on the left extreme.

The `objdump` option switches we use are:

- `-d` or `--disassemble` : Display assembler content of executable sections
- `-S` or `--source` : display a source code-assembly intermix wherever possible (implies `-d` )

Of course, there are many other options; do have a look at the man page of `objdump`. Here's some sample output:

```
static void do_the_work(struct work_struct *work)
{
ffffffffffc0604000: e8 00 00 00 00 callq ffffffff0604005
ffffffffffc0604005: 55 push %rbp
r_
```

L . . . ]

Analysis procedure:

1. Above, you can see that the start address of our `do_the_work()` function – the function where the Oops occurred (via the `RIP` value, see *Figure 7.5*) – is `0xfffffffffc0604000`
2. Add the offset as shown by the first `RIP` line in the Oops diagnostic message: `0xfffffffffc0604000 + 0x124 = 0xfffffffffc0604124`
3. Look for the closest match to this address within `objdump`'s output:

```
fffffffffc0604103: 74 27 je ffffffc060412c <do_the_work+0x12c>
 pr_info("Generating Oops by attempting to write to an invalid kernel memory pointer\n");
fffffffffc0604105: b9 3d 00 00 00 mov $0x3d,%ecx
fffffffffc060410a: 48 c7 c2 00 00 00 00 mov $0x0,%rdx
fffffffffc0604111: 48 c7 c6 00 00 00 00 mov $0x0,%rsi
fffffffffc0604118: 48 c7 c7 00 00 00 00 mov $0x0,%rdi
fffffffffc060411f: e8 00 00 00 00 callq ffffffc0604124 <do_the_work+0x124>
 oopsie->data = 'x';
fffffffffc0604124: c6 04 25 30 00 00 00 movb $0x78,0x30
fffffffffc060412b: 78
}
kfree(gctx);
```

*Figure 7.15 – Screenshot showing the output of `objdump -dS -adjust-vma=...` for our buggy module*

We've annotated the same screenshot below, highlighting it so that you can see the relevant address – `0xfffffffffc0604124` – highlighted by the blue rectangle and the relevant code region by the red rectangle:

```
fffffffffc0604103: 74 27 je ffffffc060412c <do_the_work+0x12c>
 pr_info("Generating Oops by attempting to write to an invalid kernel memory pointer\n");
fffffffffc0604105: b9 3d 00 00 00 mov $0x3d,%ecx
fffffffffc060410a: 48 c7 c2 00 00 00 00 mov $0x0,%rdx
fffffffffc0604111: 48 c7 c6 00 00 00 00 mov $0x0,%rsi
fffffffffc0604118: 48 c7 c7 00 00 00 00 mov $0x0,%rdi
fffffffffc060411f: e8 00 00 00 00 callq ffffffc0604124 <do_the_work+0x124>
 oopsie->data = 'x';
fffffffffc0604124: c6 04 25 30 00 00 00 movb $0x78,0x30
fffffffffc060412b: 78
}
kfree(gctx);
```

*Figure 7.16 – Same screenshot annotated to show the portion where the code caused the Oops!*

The C source line is just before the machine and assembly, showing that this is precisely where the fault – and thus the Oops – occurred!

If you aren't doing a live run, no matter, just run `objdump` in the same way leaving out the `--adjust-vma=` parameter (see an example of this in the upcoming section *On ARM with objdump*).

Also, `objdump` can be very useful when analyzing an Oops where the culprit is likely in-kernel code (as opposed to module code). In these cases, you'll require the disassembled `vmlinux` with source code intermixed:

```
 ${CROSS_COMPILE}objdump -dS <path/to/kernel-src/>/vmlinux > vmlinux
```

We assume the `vmlinux` used has been compiled with debug symbols. In fact, this is a one-time thing to do (unless the kernel itself is updated of course).

## Using GDB to help debug the Oops

The powerful GDB debugger can also be exploited to help with pinpointing the line of source code that triggered the Oops. GDB's `list` command can really help here. To use it though, you'll need to (re)build your module with debug symbols (I.e., with the `-g` and other useful option switches). You can see, within our better `Makefile`, if we set the variable `MYDEBUG` to `y` (the default being `n`), how it employs compiler option switches that are important for debug purposes:

```
MYDEBUG := y
ifeq (${MYDEBUG}, y)
EXTRA_CFLAGS deprecated; use ccflags-y
 ccflags-y += -DDEBUG -g -ggdb -gdwarf-4 -Og -Wall -fno-omit-frame-pointer
```

After building it for debug, let's have GDB have a go:

```
$ gdb -q ./oops_tryv2.ko
Reading symbols from ./oops_tryv2.ko...
(gdb) list *do_the_work+0x124
0x160 is in do_the_work (<...>/ch8/oops_tryv2/oops_tryv2.c:62).
[...]
61 pr_info("Generating Oops by attempting to write to an invalid
62 oopsie->data = 'x';
63 }
64 kfree(gctx);
(gdb)
```

Check it out; line 62 in the source is precisely where our bug is; GDB is bang

on target!

More on using GDB for kernel/module debug is available at the official kernel documentation site: <https://www.kernel.org/doc/html/latest/admin-guide/bug-hunting.html#gdb>.

## Using addr2line to help pinpoint the Oops code location

The `addr2line` utility has the ability to translate (virtual) addresses into their corresponding pathname(s) and line number(s)! It can be immensely useful to quickly pinpoint the place in the source code where the bug triggered.

The address (or even multiple addresses) can be specified to `addr2line` via the key `-e` (executable) option switch. (The utility takes several other optional parameters as well; do a quick `addr2line -h` to see them).

For our module, let's invoke `addr2line` as follows, passing the offset from the start of the function reported by the Oops diagnostic as the `RIP` register value (the value `0x124`; see *Figure 7.5*):

```
$ addr2line -e ./oops_tryv2.o -p -f 0x124
do_the_work at <...>/ch8/oops_tryv2/oops_tryv2.c:62
```

Again, perfect, and so easy to use! The optional parameter `-p` pretty prints the output, while `-f` displays the function name as well.

The `addr2line` utility is also useful when you have a kernel (not module) crash and the kernel virtual address. In these cases, supply the uncompressed `vmlinux` file (which has all debug symbols) to `addr2line` via the `-e` option switch:

```
addr2line -e </path/to/>vmlinux -p -f <faulting_kernel_address>
```

Note that `addr2line` will *not* work correctly on addresses generated on a system with **Kernel Address Space Layout Randomization (KASLR)**, a kernel security / hardening feature, configured (from kernel version 3.14). In this case (and it's usually the case due to security), use the `faddr2line` script instead (we cover it in the following section). Alternatively, you can disable KASLR at boot by passing kernel command line option `nokaslr` via the bootloader. (For more on KASLR, see the *Further reading* section).

Let's now check out some kernel helper scripts

Let's now check out some kernel helper scripts.

## Taking advantage of kernel scripts to help debug kernel issues

The modern Linux kernel has many helper scripts, helping you to debug kernel bugs. Here's a quick table summarizing them; a bit more detailed take on how to practically use them follows:

| Script                           | Purpose                                                                                                                                                                                                                                                                            |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| scripts/<br>checkstack.pl        | Estimates the stack size used by functions within the kernel (or module), in descending order by size                                                                                                                                                                              |
| scripts/<br>decode_stacktrace.sh | A script that tries to convert all kernel (virtual) addresses passed to it (usually by redirecting stdin from the dmesg output or piped to it), to source filenames with line numbers                                                                                              |
| scripts/<br>decodecode           | A script that attempts to add useful information by parsing the Code: <...machine code bytes...> line in a typical Oops report; figures out and specifies the instruction where the fault actually occurred and shows it as ' --- trapping instruction ' to the right of that line |
| scripts/<br>faddr2line           | Same as addr2line but appropriate for use on systems using KASLR (for security) and for interpreting stack dumps from kernel modules                                                                                                                                               |
| tools/debugging/kernel-chktaint  | Interprets the kernel tainted flags; can pass the tainted bitmask as a parameter; if not, it looks up the current system's taint state and prints its report                                                                                                                       |
| scripts/<br>get_maintainer.pl    | With the -f file directory parameter, this script identifies and prints details on the maintainer(s), the mailing list, and so on. Useful to                                                                                                                                       |

quickly find who maintains a given piece of code in the kernel!

Table 7.4 – A summary table of several useful kernel helper scripts

The list isn't exhaustive but is plenty to work with. Let's get going!

### Using the checkstack.pl script

As you should be aware, every user mode thread alive has two stacks – a user space stack and a kernel-mode stack. The former is dynamic and can grow pretty large (8 MB by default on your typical Linux); the latter is used when execution enters kernel-space. The **kernel mode stack is fixed in size and small**, typically two pages on 32-bit and four pages on 64-bit systems. In effect, just 8 KB or 16 KB, with a typical 4 KB page size (hey, don't assume the page size the MMU's using; the kernel macro `PAGE_SIZE` has the correct value).

Overflowing the kernel-mode stack is thus relatively easy to do; thus, the kernel maintainers consider it worthwhile to have a Perl script handy. It parses the kernel functions and reports the largest possible stack size required by it. This script outputs the stack size used by functions passed to it (often via `objdump` across a pipe, as seen below), in descending order by size:

```
$ objdump -d <...>/linux-5.10.60/vmlinux | <...>/linux-5.10.60/scripts/checkstack.pl
0xffffffff810002100 sev_verify_cbit [vmlinux]: 4096
0xffffffff81a554300 od_set_powersave_bias [vmlinux]: 2064
0xffffffff817b24100 update_balloon_stats [vmlinux]: 1776
[...]
```

No reason you can't run it on a kernel module; for example:

```
$ objdump -d /lib/modules/5.10.60-dbg02-gcc/kernel/drivers/net/netconsole.ko | <...>/linux-5.10.60/scripts/checkstack.pl
0x00000000000013800 enabled_store [netconsole]: 224
0x0000000000000000 init_module [netconsole]: 224
0x000000000000c300 remote_ip_store [netconsole]: 208
[...]
```

Overflowing the stack, especially a kernel mode stack, is no joking matter. The resulting stack corruption *can result in an abrupt and dramatic system hang*, with no real way to debug what exactly occurred. Better to be prepared than sorry!

A quick kernel internals note: precisely because kernel stack overflows can simply and immediately hang the system, recent kernels have shifted to enabling an arch-specific kernel config called `CONFIG_VMAP_STACK`, turning it on (the `x86_64` has it enabled from the 4.9 kernel, and `arm64` from 4.14). Very briefly, the `vmalloc()` interface is used to allocate (task and IRQ) stack memory; this ensures that a later bad page fault (perhaps from an overflow) on these pages can be handled by the kernel's fault/Oops handling code.

## Leveraging the `decode_stacktrace.sh` script

A raw (kernel) stack dump is of limited usefulness when the text – the function names – is all that's seen. The `decode_stacktrace.sh` script attempts to remedy this situation by showing, for each function name on the stack call trace, its source code location and line number within the kernel and/or module!

In effect, it's a kind of combination of the raw stack trace along with the information that the `addr2line` utility provides (just that it's done for *every* call frame on the stack. In fact, this script is at some level a wrapper over the `addr2line` utility, internally invoking  `${CROSS_COMPILE}addr2line`). It's usage is as follows:

```
$ </path/to/>/linux-5.10.60/scripts/decode_stacktrace.sh
Usage:
<...>/linux-5.10.60/scripts/decode_stacktrace.sh -r <release> | <vm:
```

You can see that, to be effective, this script requires the pathname of the uncompressed kernel `vmlinux` image with debug symbols. Next, the `base-path` parameter is the pathname of the directory where this file is available; we provide the path to our `vmlinux` within the 5.10.60 kernel source tree (where we built it). Alternately, you can specify the kernel release via the `-r` option switch; the script will attempt to retrieve the `vmlinux` image (with debug symbols) based on this value). The `modules_path` parameter is the location where the defective kernel module lives; here, it's the current directory (as we're working from there), so we specify it as `./`.

Again using our usual buggy-in-workqueue module example (on an `x86_64` guest system), we've saved the `dmesg` output into a file named `dmesg_oops_buginworkq.txt`. Just as the `vmlinux` with debug symbols must be passed, you should ensure you (re)compile the module with debug flags on

(set the `MYDEBUG` variable in the `Makefile` to `y` and rebuild it).

Here's what the `decode_stacktrace.sh` script buys us when we run it through the saved `dmesg` output:

```
$ ~/lkd_kernels/productionk/linux-5.10.60/scripts/decode_stacktrace
[...]
[448.049414] BUG: kernel NULL pointer dereference, address: 000000
[...]
[448.049547] Workqueue: events do_the_work [oops_tryv2]
[448.049562] RIP: 0010:do_the_work (/home/letsdebug/Linux-Kernel-I
<< ... output of the decodecode script ... >>
[...]
[448.049934] Call Trace:
[448.049949] process_one_work (kernel/workqueue.c:1031 (discriminat
[448.049967] worker_thread (./arch/x86/include/asm/current.h:15 ke
[448.049984] ? process_one_work (kernel/workqueue.c:2222)
[448.050002] kthread (kernel/kthread.c:277)
[...]
[448.050937] CR2: 0000000000000030
[448.051593] ---[end trace cc44ad6c5fd2bc79]---
```

This script also interprets the machine code running on the processor at the time of the fault – this work is actually performed by yet another helper script, the `scripts/decodecode` one (which we show next). Below, you can see the bash function which invokes it:

```
$ cat <...>/linux-5.10.60/scripts/decode_stacktrace.sh
[...]
decode_code() {
local scripts=`dirname "${BASH_SOURCE[0]}"`
echo "$1" | $scripts/decodecode
}
```

The original commit of this script into the kernel source tree (in version 3.16), is interesting to browse through; check it out here:

<https://github.com/torvalds/linux/commit/dbd1abb209715544bf37ffa0a3798108e>

## Interpreting machine code with the `decodecode` script

As with the `decode_stacktrace.sh` script, this script reads from standard input; thus, passing it the `dmesg` Oops output via a file or across a pipe is typical. It attempts to decode the machine code that was running on the

processor core at the time of the bug (or crash) and displays useful output. It's even able to identify the particular instruction that caused the fault (or trap) to be raised by the MMU and shows it by printing `<-- trapping instruction` to the right of that line. Check out the example screenshot below showing running this script on our `oops_tryv2` module's `printk`'s when it Oops'ed:

```
$ ~/lkd_kernels/productionk/linux-5.10.60/scripts/decodecode < dmesg_oops_buginworkq.txt
[53.695794] Code: c0 e8 d0 2d 47 c6 f6 c3 01 74 27 b9 3d 00 00 00 48 c7 c2 c0 53 60 c0 48
c7 c6 3c 50 60 c0 48 c7 c7 18 51 60 c0 e8 61 35 a8 c6 <c6> 04 25 30 00 00 00 78 48 8b 3d cd
23 00 00 e8 a8 ba 13 c6 5b 41
All code
=====
0: c0 e8 d0 shr $0xd0,%al
3: 2d 47 c6 f6 c3 sub $0xc3f6c647,%eax
8: 01 74 27 b9 add %esi,-0x47(%rdi,%r1z,1)
c: 3d 00 00 00 48 cmp $0x48000000,%eax
11: c7 c2 c0 53 60 c0 mov $0xc06053c0,%edx
17: 48 c7 c6 3c 50 60 c0 mov $0xfffffffffc060503c,%rsi
1e: 48 c7 c7 18 51 60 c0 mov $0xfffffffffc0605118,%rdi
25: e8 61 35 a8 c6 callq 0xfffffffffc6a8358b
2a:* c6 04 25 30 00 00 00 movb $0x78,0x30 <-- trapping instruction
31: 78
32: 48 8b 3d cd 23 00 00 mov 0x23cd(%rip),%rdi # 0x2406
39: e8 a8 ba 13 c6 callq 0xfffffffffc613bae6
3e: 5b pop %rbx
3f: 41 rex.B

Code starting with the faulting instruction
=====
0: c6 04 25 30 00 00 00 movb $0x78,0x30
7: 78
8: 48 8b 3d cd 23 00 00 mov 0x23cd(%rip),%rdi # 0x23dc
f: e8 a8 ba 13 c6 callq 0xfffffffffc613babc
14: 5b pop %rbx
15: 41 rex.B
$
```

*Figure 7.17 – Screenshot showing the output of the decodecode script*

It has indeed shown the trap at the precise place in the machine code / assembly line of code where it occurred (notice the operand of `0x30` – the offset we're working with – to the `movb` machine instruction where the trap occurred).

The original commit of this script into the kernel source tree happened many years back (in July 2007, kernel version 2.6.23) is here:

<https://github.com/torvalds/linux/commit/dcecc6c70013e3a5fa81b3081480c03e1>

As mentioned already, this script is itself invoked by the `decode_stacktrace.sh` script to better interpret the machine code bytes; thus,

the `decode_stacktrace.sh` script is a superset of this one.

## Exploiting the `faddr2line` script on KASLR systems

Are you running on a KASLR-enabled kernel? Let's check:

```
$ grep CONFIG_RANDOMIZE_BASE /boot/config-$(uname -r)
CONFIG_RANDOMIZE_BASE=y
```

Yes, here, we are. In cases like this (as mentioned already), the `addr2line` script may not work as expected; in such cases, use the `faddr2line` script instead (as you'll guess, it is a wrapper over the `addr2line` utility):

```
<...>/linux-5.10.60/scripts/faddr2line
usage: faddr2line [--list] <object file> <func+offset> <func+offset>
```

For example, in a portion of the call trace seen in our `oops_tryv2` module (for the bug triggered in the `workqueue` function), you can see:

```
$ ~/lkd_kernels/productionk/linux-5.10.60/scripts/faddr2line ./oops_
bad symbol size: base: 0x0000000000000000 end: 0x0000000000000000
<< TODO! ?? FAILS on the module ?? >>
Works well with vmlinux ..

...
[53.693615] Call Trace:
[53.693627] process_one_work+0x1b8/0x3b0
[...]
[53.693688] ret_from_fork+0x22/0x30
```

Okay; let's find out where exactly the `ret_from_fork()` function is in the codebase:

```
$ ~/lkd_kernels/debugk/linux-5.10.60/scripts/faddr2line ~/lkd_kerne
ret_from_fork+0x22/0x30:
ret_from_fork at arch/x86/entry/entry_64.S:302
```

Note that I invoked `faddr2line` with the *debug* kernel's `vmlinux` as a parameter; when I do the same thing, but now with the *production* kernel's `vmlinux` (where `CONFIG_DEBUG_INFO` is off), see the difference:

```
$ ~/lkd_kernels/productionk/linux-5.10.60/scripts/faddr2line ~/lkd_
ret_from_fork+0x22/0x30:
ret_from_fork at ???:
```

It simply can't help us; the moral here: even if the bug occurred in your production system, use the (same version and arch) debug kernel `vmlinux` to help track it down!

## Are we clean? The `kernel-chktaint` script

We've covered the interpretation of the kernel's *tainted flags* in the preceding section *Interpreting the kernel tainted flags*.

The `kernel-chktaint` script is a simple helper script that interprets the kernel's tainted bitmask and prints its report; here's an example when I ran it on my x86\_64 Ubuntu guest that our `oops_tryv2` buggy module caused an Oops upon:

```
$ tools/debugging/kernel-chktaint $(cat /proc/sys/kernel/tainted)
Kernel is "tainted" for the following reasons:
 * kernel died recently, i.e. there was an OOPS or BUG (#7)
 * externally-built ('out-of-tree') module was loaded (#12)
 * unsigned module was loaded (#13)
For a more detailed explanation of the various taint flags see
Documentation/admin-guide/tainted-kernels.rst in the the Linux kernel sources
or https://kernel.org/doc/html/latest/admin-guide/tainted-kernels.html
Raw taint value as int/string: 12416/'G D OE '
$
```

Figure 7.17 – Screenshot showing output from the `kernel-chktaint` helper script

If you don't pass a parameter, the script looks up the proc pseudo file `/proc/sys/kernel/tainted`, and interprets its value. Note that this script lives in the `tools/debugging` directory under the kernel source tree, not the `scripts/` one.

## Locating our saviors - the `get_maintainer.pl` script

Heard this one? When you fail, try, try again; if you fail non-stop, deny you ever tried. Ha ha, very funny. We prefer this: when all else fails, contact the maintainer(s)!

It's easy to do with the `scripts/get_maintainer.pl` script; typically, provide the file or directory via the `-f` option switch and the details are revealed (do see its pretty verbose help screen though for more options).

An example: say you're having trouble with the kernel's **Kernel GDB (KGDB)** feature and want to ask someone pertinent questions regarding your troubles.

Who do you ask? The maintainers of course; well, who maintains it? The screenshot below shows how this question is easily answered, via the `get_maintainer.pl` Perl script:

```
$ cd ~/lkd_kernels/productionk/linux-5.10.60/
$ scripts/get_maintainer.pl
scripts/get_maintainer.pl: missing patchfile or -f file - use --help if necessary
$ scripts/get_maintainer.pl -f kernel/debug/
scripts/get_maintainer.pl: No supported VCS found. Add --nogit to options?
Using a git repository produces better results.
Try Linus Torvalds' latest git repository using:
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
Jason Wessel <jason.wessel@windriver.com> (maintainer:KGDB / KDB /debug_core)
Daniel Thompson <daniel.thompson@linaro.org> (maintainer:KGDB / KDB /debug_core)
Douglas Anderson <dianders@chromium.org> (reviewer:KGDB / KDB /debug_core)
kgdb-bugreport@lists.sourceforge.net (open list:KGDB / KDB /debug_core)
linux-kernel@vger.kernel.org (open list)
$
```

*Figure 7.18 – Screenshot showing output from the kernel `get_maintainer.pl` helper script*

The last few lines provide the key portion of the answer – the KGDB maintainers, their email, and more importantly, the email address of the KGDB mailing list. Note that you need to run it from the root of the kernel source tree. Note that running this script within a git-based kernel source tree can produce superior results.

Do realize that this script searches the `MAINTAINERS` file in the root of the kernel source tree to provide its results; no reason you can't do the same with a simple `grep`:

```
$ grep -A15 -w "KGDB" MAINTAINERS
KGDB / KDB /debug_core
M: Jason Wessel <jason.wessel@windriver.com>
M: Daniel Thompson <daniel.thompson@linaro.org>
R: Douglas Anderson <dianders@chromium.org>
L: kgdb-bugreport@lists.sourceforge.net
S: Maintained
W: http://kgdb.wiki.kernel.org/
T: git git://git.kernel.org/pub/scm/linux/kernel/git/jwessel/kgdb
F: Documentation/dev-tools/kgdb.rst
[...]
```

```
F: - kernel/debug/
```

Above, `L` denotes the mailing list. So, fire off your well thought out email to the mailing list!

Well, then, you ask, what happens when you (and/or your team) are the maintainer(s)? You got it, keep reading this book and learning (tongue-in-cheek grin).

That has us close this section. Any other kernel helper scripts? Oh, there're plenty, here's a sampling of the `check*` ones:

```
\ls scripts/check*
scripts/check-sysctl-docs scripts/checkkconfigsymbols.py scripts/
```

We've seen a couple of these already (recall, the `checkpatch.pl` Perl script is invoked by our better `Makefile`!).

On this note, there's a helper script, `scripts/extract-vmlinux` : it's used to extract an uncompressed `vmlinux` from an existing kernel image file. In a similar vein, the `kdress` utility attempts to extract an uncompressed debug-symbols `vmlinux` from an existing `vmlinuz` image (and `/proc/kcore`); see `kdress` here: <https://github.com/elfmaster/kdress>. Of course, with these, it's often a case of **YMMV (Your Mileage May Vary)**, though!

We'll leave it to you to now, intrepid explorer, to check them out!

## Leveraging the console device to get the kernel log after Oops'ing in IRQ context

When you tried out the first two versions of our Oops-generating buggy modules (`ch7/oops_tryv1` and `ch7/oops_tryv2`), you'd have typically found that, though the kernel has a bug and generated an Oops diagnostic, the system is still usable. (Of course, no guarantees on this!).

These two modules generated the Oops while running kernel (module) code in process context (often, it's the `insmod` process, but our workqueue test case had the Oops occur in the context of a kernel worker thread). Now, what if we do the same thing – generate an Oops by, say, attempting to read an address within the NULL trap page (as we did earlier), **but this time while running in interrupt**

## context!

Well, our `ch7/oops_inirqv3` module does precisely this: it sets up a function that will run in (hard) interrupt context by leveraging the kernel's `irq_work*` functionality (we in fact used this same feature for running one of the memory leakage test cases in interrupt context in the previous chapter). Here's the relevant code snippet for this code – the interrupt context work function that generates a simple Oops:

```
// ch7/oops_inirqv3/oops_inirqv3.c
void irq_work(struct irq_work *irqwk)
{
 int want_oops = 1;
 PRINT_CTX();
 if (!want_oops) // okay, let's Oops in irq context!
 // a fatal hang can happen here!
 *(int *)0x100 = 'x';
}
```

Simple enough. Try running this module on your system. In my case at least, running my Ubuntu 20.04 guest VM (with our custom production 5.10.60-prod01 kernel) on Oracle VirtualBox 6.1, it has the VM simply freeze! No `printk`'s are seen, the login shell is unusable, the system appears to be hung.

*Now what? How does one debug when even a `dmesg` can't be run?*

## Setting up Oracle VirtualBox with a virtual serial port

Ah, welcome to the real world. For now, here's what we'll do: leveraging the kernel console device concept, we can **set up an additional serial console (pseudo) device on our guest VM which actually backs onto a log file on our host system**. (Do realize that this case is very specific to using an x86\_64 guest with Oracle VirtualBox as the hypervisor app; the general concept, though, is applicable pretty much everywhere).

Follow these steps to setup your hypervisor and guest system to log all kernel `printk`'s from guest to host into a file on the host:

1. If already running, shutdown your x86\_64 guest Linux VM
2. On your host system, go to your Oracle VirtualBox app GUI, select your guest VM (typically seen in the column on the left side) and open your

guest VM **Settings** (by clicking on the settings gearwheel)

3. The **Settings** dialog box opens; here's a screenshot as it appears on my host system:

 *Figure 7.19 – Screenshot of the Oracle VirtualBox Settings dialog for our guest VM*

4. Now navigate to the **Serial Ports** option; enable the serial **Port 1** (by clicking its toggle button); set the **Port Number** to **COM1** (equivalent to `ttyS0` on Linux), **Port Mode** to **Raw File** (via the dropdown menu) and within the **Path/Address** text box enter the pathname to the console log file (the path is with respect to your host system):

 *Figure 7.20 – Screenshot of the Oracle VirtualBox Settings / Serial Port setup dialog for our guest VM, setting up a serial console to log to a file on the host*

(My host system too is Linux; hence the pathname follows the usual Unix/Linux conventions; on a Windows/Mac host, provide the pathname to the file on your host in accordance with its naming conventions). Click the **OK** button when done.

5. Start the guest system; press a key (usually *Shift*) to interrupt the GRUB bootloader and enter its menu interface screen. Navigate to the **Advanced options for Ubuntu** menu; within it, highlight the appropriate kernel (I'm selecting our custom 5.10.60-prod01 kernel) via the up/down arrow keys
6. Type `e` to edit this kernel entry; scroll down to the line beginning with `linux /boot/vmlinuz-5.10.60-prod01 root=UUID=... ro quiet splash`

Most importantly, edit the kernel command line adding the new serial console(s): take the cursor to the end of this entry and type: `console=ttyS0 console=tty0 ignore_loglevel`:

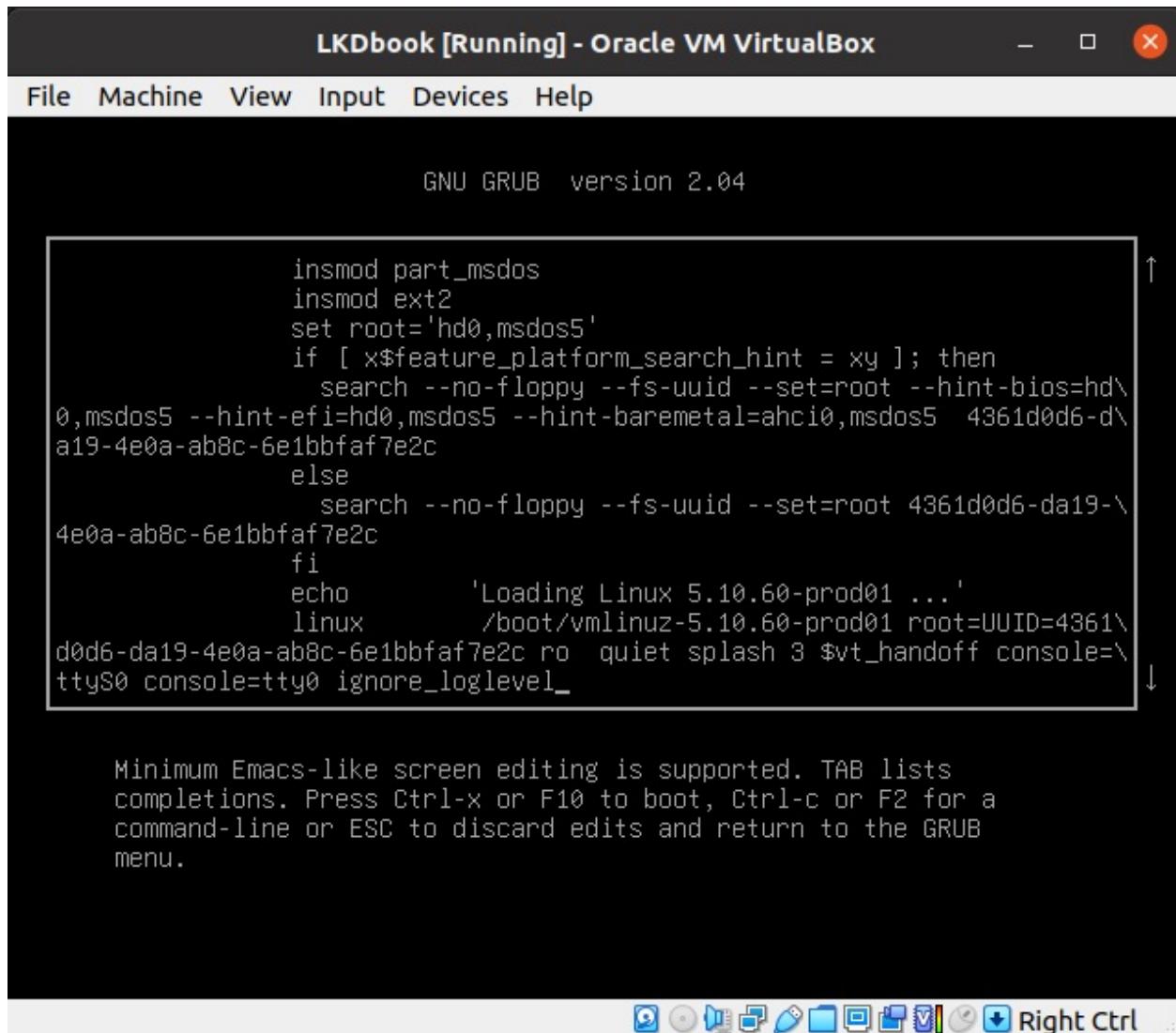


Figure 7.21 – Screenshot of the GRUB CLI, showing how we've edited the kernel command line to add additional serial console(s)

On boot, this has the target kernel understand there's additional serial console devices to send the kernel printk's to; the `ignore_loglevel` directive (as we saw before), has the kernel send all printk's to the consoles, irrespective of their log level (good for debug scenarios). When done, type `Ctrl-x` to boot.

Retrying `oops_inirqv3` with the serial console enabled

All right; by now I assume you've followed the detailed steps above, and are logged into your guest VM with the new serial console(s) enabled. Check this with:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.10.60-prod01 root=UUID=<...> ro quiet sp
```

As mentioned back in *Chapter 3, Debug via Instrumentation - printk and friends* section *Emitting a printk from userspace*, a quick test to see if the (pseudo) serial console works as expected is to do this; on the guest, as root:

```
echo "testing serial console 1 2 3" > /dev/kmsg
```

On the host, lookup your console file (by the pathname you provided earlier); the message sent above should be appended to it; on my Linux host:

```
$ tail -n1 ~/console_lkd.txt
[646.403129] testing serial console 1 2 3
```

It's fine; let's proceed, running our buggy-in-irq module again:

```
cd <...>/ch7/oops_inirqv3
make
sudo insmod ./oops_inirqv3.ko ; sudo dmesg
[... <hangs> ...]
```

Again, the system seems to be hung. But this time, the Oops diagnostic printk's would have made it across the (pseudo) serial line to the file on the host! Check; on my host, this is what I see within the serial console file:

```

[770.407919] BUG: kernel NULL pointer dereference, address: 00000000000000100
[770.408580] #PF: supervisor write access in kernel mode
[770.409050] #PF: error_code(0x0002) - not-present page
[770.409521] PGD 0 P4D 0
[770.409757] Oops: 0002 [#1] PREEMPT SMP PTI
[770.410143] CPU: 1 PID: 1699 Comm: insmod Tainted: G OE 5.10.60-prod01 #6
[770.410869] Hardware name: innoteck GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[770.411615] RIP: 0010:irq_work+0x36/0x150 [oops_inirqv3]
[770.412100] Code: 05 6f fb 9a 3f a9 00 01 ff 00 74 19 a9 00 00 0f 00 0f 84 f5 00 00 00 ba 48 00 00 00 f6
c4 ff of 84 e7 00 00 0f 1f 44 00 00 <c7> 04 25 00 01 00 00 78 00 00 00 c3 55 65 4c 8b 04 25 c0 7b 01 00
[770.413793] RSP: 0018:ffff9cc4800f0f78 EFLAGS: 00010006
[770.414274] RAX: 0000000080010001 RBX: ffffffff066b480 RCX: 0000000000000080b
[770.414922] RDX: 0000000000000068 RSI: ffffffff066b480 RDI: ffffffff066b480
[770.415546] RBP: fffff9cc4800f0f98 R08: 0000000000000000 R09: 0000000000000000
[770.416168] R10: 0000000000000000 R11: 0000000000000000 R12: 0000000000000022
[770.416787] R13: 0000000000000020 R14: 0000000000000000 R15: 0000000000000000
[770.417397] FS: 00007f514a975540(0000) GS:ffff903c7dc40000(0000) knlGS:0000000000000000
[770.418136] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[770.418662] CR2: 0000000000000100 CR3: 00000000265fe006 CR4: 00000000000706e0
[770.419283] Call Trace:
[770.419500] <IRQ>
[770.419684] ? irq_work_single+0x34/0x50
[770.420032] irq_work_run_list+0x31/0x50
[770.420398] irq_work_run+0x5a/0xf0
[770.420711] __sysvec_irq_work+0x30/0xd0
[770.421085] asm_call_irq_on_stack+0x12/0x20
[770.421479] </IRQ>
[770.421677] sysvec_irq_work+0x9f/0xc0
[770.422478] asm_sysvec_irq_work+0x12/0x20
[770.423215] RIP: 0010:native_write_msr+0x6/0x30
[770.423990] Code: 0f 1f 40 00 0f 1f 44 00 00 55 48 89 e5 0f 0b 48 c7 c7 60 07 07 b5 e8 51 70 c1 00 66 0f
1f 84 00 00 00 00 89 f9 89 f0 0f 30 <0f> 1f 44 00 00 c3 55 48 c1 e2 20 89 f6 48 09 d6 31 d2 48 89 e5 e8
[770.426876] RSP: 0018:ffff9cc482603bb0 EFLAGS: 00000206
[770.427732] RAX: 00000000000000f6 RBX: 0000000000000010 RCX: 0000000000000083f
[770.429039] RDX: 0000000000000000 RSI: 00000000000000f6 RDI: 0000000000000083f
[770.430134] RBP: fffff9cc482603bb8 R08: 0000000000000010 R09: fffff903c137550a0
[770.431195] R10: fffff903c01ae5410 R11: 0000000000000000 R12: ffffffff066b480
[770.432229] R13: 0000000000288a8 R14: 0000000000000001 R15: ffffffff066b0d8
[770.433264] ? native_apic_msr_write+0x2b/0x30
[770.434066] x2apic_send_IPI_self+0x20/0x30
[770.434859] arch_irq_work_raise+0x2a/0x40
[770.435620] __irq_work_queue_local+0xbff/0x130
[770.436398] irq_work_queue+0x32/0x50
[770.437091] ? 0xffffffff0662000
[770.437751] try_oops_init+0x2a/0x1000 [oops_inirqv3]
[770.438559] do_one_initcall+0x48/0x210
[770.439353] ? kmem_cache_alloc_trace+0x3ae/0x450
[770.440431] do_init_module+0x62/0x240
[770.441121] load_module+0x2a04/0x3080
[770.441814] ? security_kernel_post_read_file+0x5c/0x70
[770.442651] __do_sys_finit_module+0xc2/0x120
[770.443390] ? __do_sys_finit_module+0xc2/0x120
[770.444141] x64_sys_finit_module+0x1a/0x20

```

*Figure 7.21 – Partial screenshot showing the content of the serial console raw file on the host; we can clearly see (and thus analyze) the Oops diagnostic!*

Fantastic; this time we can see the Oops diagnostic in all its glory and therefore analyze it.

Take a closer look, it's interesting: here, we know that the bug occurred in an interrupt context. The `RIP` register points to the correct IRQ work function (`irq_work()`). Further verifying this, the call (stack) trace now has two distinct parts – the IRQ stack (delimited by the `<IRQ> ... </IRQ>` tokens) and, further

down, the non-IRQ process-mode kernel stack trace. Together, reading them bottom-up (ignoring the frames prefixed with ?), they paint a pretty clear picture of what happened.

## Several stacks

The reality is that there can be several stacks in existence simultaneously; it depends on the arch (CPU) and on what happened, on the code paths taken. On modern arch's, for example, interrupt processing occurs on a separate stack, the IRQ stack. On the x86\_64, there can be a regular stacks – the user space and the kernel mode stack (what are called the task stacks), an IRQ stack, a hardware exception stack (for handling double faults, NMI, debug, mce) and an entry stack. Starting with the current stack pointer, the stacks frames are unwound; each stack has a pointer to the next one.

Of course, the line at the top:

```
BUG: kernel NULL pointer dereference, address: 0000000000000100
```

also clearly indicates the NULL pointer page dereference that is the root cause of this Oops; where it occurred is what the call traces reveal.

The Oops output contains the process context in play at the time of the Oops:

```
CPU: 1 PID: 1699 Comm: insmod Tainted: G OE 5.10.60-pi
```

Now, this might have you think that the `insmod` process was the one executing kernel code at the time of the crash and that it's the culprit. Well, not this time! This is as – as you'll recall – this time, the bug occurred while the kernel was executing our IRQ work function, *in interrupt context*. The presence of the IRQ call stack gives us that information as well...

So, an important thing to realize: the `insmod` process was merely the value that the `current` macro happened to point to at the time the Oops occurred; in other words, the `insmod` process was actually interrupted by the interrupt (a work IRQ here)! That's why it shows up; it doesn't necessarily mean it was running the code that Oops'ed.

Quite often, when analyzing Oops'es, you might find the process context is swapper (PID 0). That, of course, is the kernel thread that runs on the

processor when that core is idle (it's the so-called *idle thread*, there's one per CPU core, of the form `swapper/n`, where `n` is the core number starting with `0`). So, the point here, is that when this shows up as the process context, it's perhaps more likely that it was interrupted by some interrupt (or softirq) that is actually the one that ran the buggy code.

The last line of output (not seen in the above truncated screenshot) shows why the system hung – it's considered a fatal error, the kernel did actually panic:

```
[770.483105] ---[end Kernel panic - not syncing: Fatal exception
```

A word of caution: unfortunately, the serial console log file seemed to get truncated. (This seems to be a known issue, see the link in the *Further reading* section).

This general approach here has us specifying a serial console device to log kernel printk's to. Extending this approach, the **netconsole** facility allows one to log kernel printk's across a network! In effect, *it's a remote printk facility*; we shall shortly cover the basic usage of netconsole.

## An Oops on an ARM Linux system and using netconsole

To get the most out of this section, I'd definitely recommend you be at least a little conversant with the processor ABI conventions (especially stuff like function call, parameter passing, return value) for the processor your code's running upon. So, here, it's for the ARM-32; again, do review the basics that we covered in *Chapter 6, Debug via Instrumentation – Kprobes* in the section *Understanding the basics of the Application Binary Interface (ABI)*.

Here, our test environment is: a Raspberry Pi 0W running Raspbian 10 (buster) with the standard 5.10.17+ kernel. This popular prototyping (and product) board has the Broadcom BCM2835 **System on Chip (SoC)** which internally sports a single ARM-32 CPU core for this board.

We cross compile and run our usual test case on the device: our `oops_tryv2` module case 3 (passing the `bug_in_workq=yes` parameter). It does cause an Oops, of course, but the bug was severe enough to hang the board entirely! So how do we get to see the kernel log?

Ah, the kernel's netconsole facility turns out to be the answer (well, at least one way. Another way is to use the Raspberry Pi's serial UART along with a USB-to-serial converter and to see the console output on a terminal – minicom / Hyperterminal - window)! Of course, we assume you have the target kernel – in our case, the (embedded ARM) target system kernel configured for netconsole; the kernel config `CONFIG_NETCONSOLE` should be set to either `y` or `m` (here, we assume it's built as a module, the typical case).

To load the netconsole driver as a module, this is the format of the key parameter, named `netconsole`:

```
netconsole=[+][src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-ri
```

(Please read the official kernel doc for completeness here:  
<https://www.kernel.org/doc/Documentation/networking/netconsole.txt>. We leave the source and destination ports as defaults).

Netconsole works by configuring a sender, and sending all kernel printk's to a receiver system across a network. Quite obviously, the sender is the one that sends data – the kernel printk content – across the network (over UDP) to the receiver system. Very briefly, I setup netconsole as follows:

- Set the Raspberry Pi 0W as the sender system: strictly speaking, the sender should have a static IP address so that the receiver can reliably specify it; here, to test, we don't bother setting it up, it does work (of course, the IP addresses below are an example, change it to suit your systems addresses):

```
sudo modprobe netconsole netconsole=@192.168.1.24/wlan0,@192.168.1.100,6666
```

- Set the Linux host system (it could be your x86\_64 guest VM) as the receiver (you can also setup a Win/Mac host as the receiver system; I won't delve into that here). On the receiver system, you don't actually require the `netconsole` module installed. Simply run `netcat` as follows to capture the incoming network stream (from the sender embedded system) and log the data it receives to a file:

```
netcat -d -u -l 6666 | tee -a dmesg_arm.txt
```

Run the buggy module on the embedded board, while running `netcat` on the host receiver system (to capture the kernel log being sent by netconsole from the

embedded system).

## A practical consideration – ARM (cross) compiler fails

I find that, quite often, when building the module for ARM (using the x86\_64-to-ARM32 cross compiler, `arm-linux-gnueabihf-gcc`), it fails with an error of the sort:

```
ERROR: modpost: "__aeabi_ldivmod" [<...>/ch8/oops_tryv2/oops_tryv:
```

This seems to be an issue with the way division is carried out for ARM; a silly workaround: just comment out code that performs division; here, our `SHOW_DELTA()` macro. Once removed, it compiles correctly.

The screenshot below captures these (the window with the light background is the embedded sender system, where we run our buggy module; the window with the darker background is the receiver host system where `netcat` is running):

```

rpi oops_tryv2 #
rpi oops_tryv2 # modprobe netconsole netconsole=@192.168.1.24/wlan0,@192.168.1.101/
rpi oops_tryv2 # echo test123 > /dev/kmsg
rpi oops_tryv2 #
rpi oops_tryv2 # insmod ./oops_tryv2.ko bug_in_workq=yes
]

root@k7550:/home... ~ Terminal Terminal Terminal Terminal
[6964.642063] test123
[6982.109243] oops_tryv2: loading out-of-tree module taints kernel.
[6982.115208] oops_tryv2:try_oops_init():87: Generating Oops via kernel bug in workqueue function
[6982.127430] oops_tryv2:do_the_work():57: In our workq function: data=67
[6982.131918] oops_tryv2:do_the_work():61: Generating Oops by attempting to write to an invalid kernel memory pointer
[6982.140055] 8<---- cut here ---
[6982.144180] Unable to handle kernel NULL pointer dereference at virtual address 0000001c
[6982.152208] pgd = 01cf7cd3
[6982.156062] [0000001c] *pgd=00000000
[6982.159842] Internal error: Oops: 817 [#1] ARM
[6982.163651] Modules linked in: oops_tryv2(0) netconsole aes arm aes_generic cmac bnef hc_i_uart btbcm bluetooth ecdh_generic ecc libaes 8021q garp stp llc brcmfmac brcmutil sha256 generic libsha256 cfg80211 rfkill raspberrypi_hwmon bcm2835_codec(C) bcm2835_isp(C) snd_bcm2835(C) bcm2835_v4l2(C) v4l2_mem2mem bcm2835_mmhal_vchiq(C) videobuf2_vmalloc videobuf2_dma_contig videobuf2_memops videobuf2_v4l2 snd_pcm videobuf2_common vc_sm_cma(C) snd_timer snd_videodev mc uio_pdrv_genirq uio_fixed i2c_dev ip_tables x_tables ipv6 [last unloaded: netconsole]
[6982.189999] CPU: 0 PID: 994 Comm: kworker/0:1 Tainted: G WC 0 5.10.17+ #1414
[6982.197569] Hardware name: BCM2835
[6982.201486] Workqueue: events do_the_work [oops_tryv2]
[6982.205388] PC is at do_the_work+0x68/0x94 [oops_tryv2]
[6982.209269] LR is at 0x0
[6982.213225] pc : [<bf1a0068>] lr : [<00000000>] psr: 60000013

```

Figure 7.22 – Screenshot showing netconsole in action; the left lighter window is the embedded sender; the right darker window is the host running netcat

Voila. We capture the kernel log on the receiver system, and can now analyze it at leisure.

Though we don't delve into the details here, I'll point this out: the Oops bitmask here, the value 817 (it's in the line: Internal error: Oops: 817 [#1] ARM ; it's in hexadecimal), is definitely *not* interpreted the same way as we did for the x86. So how does one interpret it? You'll need to refer to the **Technical Reference Manual (TRM)** of the particular ARM core in question – here it's the core on the BCM2835, the ARM1176JZF-S; interpreting its **Fault Status Register (FSR)** encoding is what needs to be done to see what it means.

## Figuring out the actual buggy code location (on ARM)

The key lines in the Oops diagnostic (from the ARM system's kernel, sent over netconsole to our receiver system) are:

```
Workqueue: events do_the_work [oops_tryv2]
PC is at do_the_work+0x68/0x94 [oops_tryv2]
LR is at irq_work_queue+0x6c/0x90
```

In this particular case, the system seems to have hard-hung before the call stack can even be shown! No matter: the line with **Program Counter (PC)** – the equivalent of the RIP register on the x86\_64 – clearly tells us what occurred: the bug seems to have been triggered at the function `do_the_work()`, within the `oops_tryv2` module, at an offset of 0x68 (decimal 104) bytes from the start of this function; the length of this function is 0x94 (decimal 148) bytes.

Interestingly, the **Link Register (LR)** register on ARM *specifies the return address*; in effect, we can tell that the function `irq_work_queue()` called our workqueue routine `do_the_work()`! (And, as with the x86\_64, the `Workqueue:` line tells us that the kernel-default `events` workqueue functionality was invoked to run our work routine).

Okay, let's make good use of what we learned in the previous section and use some of the tooling at our disposal to actually identify the buggy line of code that triggered this Oops! We shall do so using three of these tools – `addr2line`, `GDB` and `objdump`; read on to see the magic!

### On ARM with `addr2line`

On the device, let's run the powerful `addr2line` utility on our module, providing the start offset that the Oops reported (the value `0x68`, as you saw above; and hey, the `rpi oops_tryv2 $` is our shell prompt, as the environment variable `PS1=rpi \w $`):

```
rpi oops_tryv2 $ addr2line -e ./oops_tryv2.ko 0x68
</path/to/>Linux-Kernel-Debugging/ch8/oops_tryv2/oops_tryv2.c:62
```

Here's a relevant snippet of the source file with line numbers prefixed:

```
61 pr_info("Generating Oops by attempting to write to an in-
62 oopsie->data = 'x';
63 }
```

The offending buggy line is highlighted. The `addr2line` utility has nailed it!

## On ARM with GDB

On the device, let's now run GDB on our module, using its powerful `list` command to get the job done:

```
rpi oops_tryv2 $ gdb -q ./oops_tryv2.ko
Reading symbols from ./oops_tryv2.ko...done.
(gdb) list *do_the_work+0x68
0x68 is in try_oops_init (/home/pi/Linux-Kernel-Debugging/ch8/oops_tryv2/oops_tryv2.c:62).
57 pr_info("In our workq function: data=%d\n", priv->data);
58 t2 = ktime_get_real_ns();
59 // SHOW_DELTA(t2, t1);
60 if (!bug_in_workq) {
61 pr_info("Generating Oops by attempting to write to an invalid kernel memory pointer\n");
62 oopsie->data = 'x';
63 }
64 kfree(gctx);
65 }
66
(gdb) █
```

*Figure 7.23 - Screenshot showing how ARM GDB's `list` command caught the offending source line*

Look carefully at the output of the `list *do_the_work+0x68` command in the screenshot above; again, exactly right!

## On ARM with objdump

On the device, let's now run the `objdump` utility on our module (I've removed some empty lines from the output below):

```
rpi oops_tryv2 $ objdump -dS ./oops_tryv2.ko
./oops_tryv2.ko: file format elf32-littlearm
Disassembly of section .text:
00000000 <do_the_work>:
/* Our workqueue callback function */
static void do_the_work(struct work_struct *work)
{
 0: e1a0c00d mov ip, sp
 4: e92dd800 push {fp, ip, lr, pc}
```

```
8: e24cb004 sub r1, 1p, #4
[...]
```

Scroll down to the nearest match to the start offset (as seen in the usual funcname+x/y format; here, the x value is 0x68):

```
5c: ebfffffe bl 0 <printk>
oopsie->data = 'x';
60: e3a03000 mov r3, #0
64: e3a02078 mov r2, #120 ; 0x78
68: e5c3201c strb r2, [r3, #28]
}
kfree(gctx);
```

Clearly, the offending source line that generated this (ARM) assembler is highlighted above. Again, bang on target!

As mentioned earlier, a practical consideration: *what if you can't run these tools on the embedded device?* Then, you'll need to have:

- The unstripped debug kernel (`vmlinux` image) with debug symbols available
- The cross toolchain with all tools/utils used to generate the bootloader, kernel and root filesystem images for your target.

Again, this is why we highly recommend you always build and keep a debug kernel in addition to the production one.

When running other kernel helper scripts on an Oops generated on a non-x86 platform, don't forget to set the `ARCH` and `CROSS_COMPILE` environment variables appropriately (just as we do when cross compiling). For example, to run the `decode_stacktrace.sh` script forOops output generated on an ARM machine, do (something like this):

```
ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- scripts/decode_stacktrac
```

One last thing: for a bit of variety, here's the same bug (and Oops) being triggered on another popular ARM development / prototyping board – the BeagleBone Black (from Texas Instruments)!

```

[20178.051346] oops_tryv2:try_oops_init():87: Generating Oops via kernel bug in workqueue function
[20178.064694] oops_tryv2:do_the_work():57: In our workq function: data=67
[20178.075333] oops_tryv2:do_the_work():61: Generating Oops by attempting to write to an invalid kernel memory pointer
[20178.097847] Unable to handle kernel NULL pointer dereference at virtual address 00000001c
[20178.107986] pgd = 7f31d0d1
[20178.110727] [00000001c] *pgd=00000000
[20178.116429] Internal error: Oops: 805 [#2] PREEMPT SMP ARM
[20178.121959] Modules linked in: oops_tryv2(0+) oops_tryv1(0+) usb_f_acm u_serial usb_f_ecm usb_f_mass_storage usb_f_rndis u_ether libcomposite wkup_m3_rproc pm33xx wkup_m3_ipc uio_pdrv_genirq uio_pruss_soc_bus pru_rproc pruss_irq_pruss_intc remoteproc virtio_ring spidev
[20178.146455] CPU: 0 PID: 3912 Comm: kworker/0:1 Tainted: G D O 4.19.94-ti-r42 #1buster
[20178.155452] Hardware name: Generic AM33XX (Flattened Device Tree)
[20178.161596] Workqueue: events do_the_work [oops_tryv2]
[20178.166763] PC is at do_the_work+0x84/0xa0 [oops_tryv2]
[20178.172025] LR is at wake_up_klogd+0x7c/0xa8
[20178.176313] pc : [<bf10e084>] lr : [<c01ac370>] psr: 600f0013
[20178.182606] sp : dae09ee8 ip : dae09e10 fp : dae09efc
[20178.187853] r10: 00000000 r9 : dc761b10 r8 : 00000000
[20178.193100] r7 : df900a00 r6 : df8fd700 r5 : dc121200 r4 : dc761b0c
[20178.199654] r3 : 00000000 r2 : 00000078 r1 : c10ed348 r0 : 00000067
[20178.206212] Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment none
[20178.213379] Control: 10c5387d Table: 9c4cc019 DAC: 00000051
[20178.219155] Process kworker/0:1 (pid: 3912, stack limit = 0x77671b58)
[20178.225625] Stack: (0xdcae09ee8 to 0xdcae0a000)
[20178.230005] 9ee0: 00000043 c0169a40 dae09f34 dae09f00 c0159b20 bf10e00c
[20178.238223] 9f00: df8fd700 df8fd700 df8fd700 dc121200 dc121214 df8fd700 00000008 df8fd718
[20178.246440] 9f20: c1504d00 df8fd700 dae09f74 dae09f38 c015aa84 c0159978 c0d3d4c8 c10e1598
[20178.254658] 9f40: c15dd636 ffffffe000 c015ff00 d9ed6cc0 d9ed65c0 00000000 dae08000 dc121200
[20178.262874] 9f60: c015aa24 d9a09e74 dae09fac dae09f78 c016040c c015aa30 d9ed6cdc d9ed6cdc
[20178.271091] 9f80: 00000000 d9ed65c0 c0160354 00000000 00000000 00000000 00000000 00000000
[20178.279308] 9fa0: 00000000 dae09fb0 c01010e8 c0160360 00000000 00000000 00000000 00000000
[20178.287524] 9fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
[20178.295741] 9fe0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
[20178.303997] [<bf10e084>] (do_the_work [oops_tryv2]) from [<c0159b20>] (process_one_work+0x1b4/0x504)
[20178.313180] [<c0159b20>] (process_one_work) from [<c015aa84>] (worker_thread+0x60/0x508)
[20178.321312] [<c015aa84>] (worker_thread) from [<c01604c0>] (kthread+0x16c/0x174)
[20178.328747] [<c01604c0>] (kthread) from [<c01010e8>] (ret_from_fork+0x14/0x2c)
[20178.335998] Exception stack(0xdcae09fb0 to 0xdcae09ff8)
[20178.341072] 9fa0: 00000000 00000000 00000000 00000000
[20178.349289] 9fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
[20178.357504] 9fe0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
[20178.364154] Code: e34b0f10 eb427ace e3a03000 e3a02078 (e5c3201c)
[20178.387196] ---[end trace 1218b813e308db06]---

```

*Figure 7.23 – Screenshot showing the same Oops on a TI BeagleBone Black running Debian Linux with a custom 4.19.94 kernel*

I'll leave it an exercise to you to carefully browse this screenshot and spot all the key points that we've spent so many pages discussing!

A tip: as mentioned earlier, interpreting the Oops bitmask on system other than the x86 implies looking up the TRM for that processor. Here, the Oops bitmask shows up in this line (highlighted):

```
Internal error: Oops: 805 [#2] PREEMPT SMP ARM
```

So, we need to look up the TRM for the TI Sitara AM335x SoC (it's a Cortex-A8 core; that's the one this board's running upon). Within it, the **Memory Protection Fault Status Register (MPFSR)** holds the error code, what shows

up as the Oops bitmask! The relevant TRM document is available here as a PDF document (<https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf>). The details on the internal encoding of the MPFSR register is available in section 11.4.1.87 (as of this writing, the relevant MPFSR register details can be found on page 1735 of this PDF document).

## A few actual Oops'es

A few actual Oops'es (that I've very arbitrarily searched for via the keyword `oops`, from the **Linux Kernel Mailing List (LKML)** archives):

- A kernel NULL pointer dereference (on 4.14-rc2):  
<https://groups.google.com/g/linux.kernel/c/rG2uYWdoteo/m/6RacvsJ6BwA hl=en>
- An Oops on 4.9.33 (read the email):  
<https://groups.google.com/g/linux.kernel/c/t4IRjnxo2Kc/m/7Me5AEVIBwA>
- An Oops flagged by Intel's superb kernel test robot! (this has been reproduced on a Qemu-based x86-32; so, look for the `EIP` register, not `RIP`!): <https://lkml.org/lkml/2020/8/10/1390>
- A recent one (as of this writing) – an Oops on 5.14.19:  
<https://lkml.org/lkml/2021/11/18/1116>
- An Oops on arm64 while booting 5.8.0-rc5 (read the analysis):  
<https://lkml.org/lkml/2020/7/20/139>

Then there's the interesting **Linux Driver Verification (LDV)** project. They have a set of rules that are validated via their static and dynamic analysis frameworks, as well as other tooling. As far as kernel bugs go, this project has found several; they're documented here, under the heading *Problems in Linux Kernel*: <http://linuxtesting.org/results/ldv>. Do take a look!

Of course, you can always simply search the kernel Bugzilla site (<https://bugzilla.kernel.org/query.cgi>) for Oops'es. (Do note, though, that the kernel community really wants you to write your bug report direct to the appropriate mailing list (recall the `get_maintainer.pl` script) and copied to the LKML, not this site).

A key concern, of course, is *being able to obtain the kernel log* in the first place; else, how can one analyze and interpret the Oops or panic that possibly occurred. With a severe-enough bug, the saving of the kernel log to disk (or flash memory)

may be compromised. For this reason, there are alternatives; here's a short list:

- **Serial console**: the kernel printk's are saved onto another system across a physical serial console; it can also be a virtual serial console, as we saw in the previous section
- **Netconsole**: a facility to enable the transfer of kernel printk's across a network
- Employing persistent RAM to save the kernel log buffer; for example, the kernel Ramoops framework has the kernel continually save kernel printk's into a circular buffer in a persistent memory region (allowing the content to be accessed after a reboot). See details in the official kernel doc here:  
<https://www.kernel.org/doc/html/latest/admin-guide/ramoops.html>
- The kernel's elaborate **Kdump** framework to capture the entire kernel memory image; this, along with the `crash` app to analyze it can be very powerful. We'll provide an introduction in a later chapter

Many similar (to the ones mentioned above) independent implementations have been made by both individuals and organizations; you may come across some while working on projects or products.

## Summary

Awesome! Great job on completing this really important chapter!

Here, you first learned what a kernel Oops is; you can perhaps think of it as the equivalent to a user mode segfault, but as it's the kernel that's buggy, all guarantees are off. We began by showing you how to generate a simple NULL pointer dereference bug, triggering an Oops (though it may sound silly and obvious, these bugs still do occur; the last portion of this chapter points you to some actual Oops'es, some of which are NULL pointer dereference bugs). We then went a bit further, triggering bus in the NULL trap page and then in a random sparse region of kernel VAS (recall the useful `procmap` utility which allows you to see the entire memory map of any process). Still further, more realistically, we used the kernel's default events workqueue to have a kernel worker thread illegally access an invalid pointer, causing an Oops (our case 3)! We used this as a useful test case throughout the remainder of the chapter.

The meat of this topic – actually interpreting the detailed Oops diagnostic – was then covered in a lot of detail, with many screenshots to show you how it looks.

Of course, being arch-specific, we covered it mainly from the viewpoint of the x86\_64. We also did cover generating and interpreting an Oops on ARM (32bit) systems, using the Raspberry Pi 0W (and a quick look at the BeagleBone Black Oops screenshot) as test boards. Learning to use various toolchain utilities and kernel helper scripts to help you debug the Oops was a critical learning here. We even covered using the powerful netconsole facility, along the way.

The chapter closed with pointing you to a few actual Oops'es that are interesting to see; importantly, we mentioned a few techniques to help capture the kernel log in situations where it can get problematic.

Needless to say (but I'll say it!), please do take the trouble to look at (and even generate!) actual Oops'es, and practice using the various available tools and techniques to try and interpret them.

With this behind you, I'll see you in the following chapter where we'll look at another key topic – figuring out locking bugs.

## Further reading

- My earlier book: *Linux Kernel Programming, Part 2 - Char Device Drivers and Kernel Synchronization* is freely downloadable as an e-book here:  
[https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Linux-Kernel-Programming-\(Part-2\)/Linux%20Kernel%20Programming%20Part%202%20-%20Char%20Device%20Drivers%20and%20Kernel%20Synchronization\\_e](https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Linux-Kernel-Programming-(Part-2)/Linux%20Kernel%20Programming%20Part%202%20-%20Char%20Device%20Drivers%20and%20Kernel%20Synchronization_e)
- VirtualBox serial console log file getting truncated? This Q&A seems related: [Solved] How to create unique path for serial port log file:  
<https://forums.virtualbox.org/viewtopic.php?f=1&t=86254>
- CPU Registers on the x86\_64: [https://wiki.osdev.org/CPU\\_Registers\\_x86\\_64](https://wiki.osdev.org/CPU_Registers_x86_64)
- Official kernel documentation: Bug hunting:  
<https://www.kernel.org/doc/html/latest/admin-guide/bug-hunting.html>
- KASLR
  - Kernel address space layout randomization, LWN, Oct 2013:  
<https://lwn.net/Articles/569635/>
  - A brief description of ASLR and KASLR, Sep 2019:  
<https://dev.to/satorutakeuchi/a-brief-description-of-aslr-and-kaslr-2bbp>

- The Meltdown/Spectre hardware bugs
  - Meltdown and Spectre: <https://meltdownattack.com/>
  - Spectre and Meltdown explained: A comprehensive guide for professionals, Tech Republic, May 2019: <https://www.techrepublic.com/article/spectre-and-meltdown-explained-a-comprehensive-guide-for-professionals/>
  - CVE details: [https://www.cvedetails.com/cve-details.php?t=1&cve\\_id=cve-2017-5753](https://www.cvedetails.com/cve-details.php?t=1&cve_id=cve-2017-5753)
  - Here's how, and why, the Spectre and Meltdown patches will hurt performance, Ars Technica, Jan 2018: <https://arstechnica.com/gadgets/2018/01/heres-how-and-why-the-spectre-and-meltdown-patches-will-hurt-performance/>
  - KPTI/KAISER Meltdown Initial Performance Regressions, B Gregg, Feb 2018: <https://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html>
- Netconsole
  - Official kernel doc: <https://www.kernel.org/doc/Documentation/networking/netconsole.txt>
  - Also briefly covered here: Debugging by printing, eLinux: [https://elinux.org/Debugging\\_by\\_printing#NetConsole](https://elinux.org/Debugging_by_printing#NetConsole)
- Article: Much ado about NULL: Exploiting a kernel NULL dereference, Oracle Linux Blog, Apr 2010: <https://blogs.oracle.com/linux/post/much-ado-about-null-exploiting-a-kernel-null-dereference>