前言

Linux Kernel Debugging 是目前主要探討關於 kernel 和 kernel module 以 debug 的最新書籍。我們詳細介紹各種強大的開源（open source）工具，以及許多進階技巧，不只是 printk，還有更多可以運用於 debug kernel、kernel module 與 device driver。這是每個專業開發人員都必須學習和掌握的關鍵技能。

本書適用的對象

本書適合 Linux kernel 的開發者、module、device driver 的開發人員，以及想要在 kernel 層進行除錯工作與加強 Linux 系統能力的測試人員；對於想要理解並對 Linux kernel 內部架構除錯的系統管理員來說，這本書也很實用；想要掌握 C 語言程式設計以及 Linux 指令的人也會很有用。總之，學點 kernel（跟 module）的開發經驗，對你肯定是有益而無害的。

本書涵蓋的內容

〈第 1 章：軟體除錯概論〉，引領我們踏上這趟旅程，探討何謂軟體除錯，以及它如何結合科學與藝術。幾個精選的軟體「恐怖故事」將強調精心設計、良好且安全地撰寫程式碼，以及解決問題的除錯能力重要性。在更實際的層面上，你將在自己的 Linux 系統或 VM 上設置所需的工作區（workspace），以便可以跟著後面的範例和作業一起練習；這點非常重要。

〈第 2 章：Debug Kernel 的方法〉，涵蓋各種以 kernel 程式碼層級的角度來進行對 kernel 除錯的方法。這將讓你建立觀念，了解如何根據特定情況和系統限制，選擇最佳或最可行的方法。

〈第 3 章：透過檢測除錯：使用 printk 與其族類〉，重新回顧常用的 printk() kernel API 基本知識。接下來，將深入探討如何利用它來實現 kernel / device driver 的除錯方法。本章的重點在於「了解 kernel 強大的動態除錯框架（dynamic debug framework），以及如何在產品化的軟體使用」。

〈第 4 章：透過 Kprobes 儀器進行 debug〉，本章會說明 kernel 的強大 Kprobes 框架，這是不同於其他方法操控 kernel 和 module 的方式，幾乎可以掛入（hook）到任何 kernel 或 module 的函式，即使是工廠製造過程中的產品，這個實用方法可以在生產過程中用來除錯系統。

〈第 5 章：Kernel 記憶體除錯問題初探〉。探討記憶體（memory）的 bug 與毀損（corruption），在 C 這類型的語言，像這種記憶體類型的問題很常發生。本章會先讓你了解發生這種事的原因，然後切入重點：這類系統中容易出現的典型記憶體問題。接下來，你將學習如何運用強大的、基於編譯器的（compiler-based）KASAN 技術，以及 kernel 之基於編譯器的 UBSAN 技術，來迎頭痛擊這些記憶體問題。

〈第 6 章：再論 Kernel 記憶體除錯問題〉。繼續介紹 kernel 的記憶體除錯問題，深入探討常見的 slab（SLUB）記憶體除錯問題細節，接著使用 kmemleak 來檢測困難的 kernel 記憶體洩漏（memory leakage）bug。在第 5 章與第 6 章的結尾會詳細比較各種記憶體損毀問題與相對應的檢測工具。

〈第 7 章：Oops！解讀 kernel 的 bug 診斷〉，本章的關鍵主題為：何謂 kernel 的「Oops」診斷訊息，以及更重要的，如何深入解讀？在這趟有趣的旅程中，會讓你產生一個簡單的 kernel Oops，並用最完整的方式解讀。此外，還將示範（demo）幾個有助於完成此任務的工具和技巧。深入了解 Oops 通常是有助於找出 kernel bug 的根本原因（root cause）！本章也會顯示一些實際的 Oops 訊息。

〈第 8 章：鎖的除錯〉。本章探討的重點是上鎖（lock），上鎖是設計穩健與可靠的 kernel 或 device driver 程式碼的關鍵。不幸的是，上鎖很容易出錯，如造成死結（dead lock）等問題，而且事後很難除錯。本章會簡介對上鎖問

題除錯的基本知識，大部分的篇幅則用來說明一個極為強大的現代化工具，**KCSAN（Kernel Concurrent Sanitizer）**，這個工具有助於揭露深層的上鎖問題，即資料競爭（data race）。你在本章將學習如何為（debug）kernel 設定 KCSAN 組態，以及使用方法。在深入探討幾個實際的 kernel bug 案例後，會歸納出這些 bug 的根本原因都是上鎖問題的結論。

〈第 9 章：追蹤 Kernel 流程〉。本章介紹強大技術，讓你可以詳細追蹤 kernel 程式碼流程，精細到如何使用每個函式的呼叫！首先介紹主要的 kernel 追蹤基礎工具 ftrace 以及它的使用方式。接下來，你將學習如何使用 ftrace 的強大前端工具：trace-cmd、KernelShark GUI（Graphic User Interface），以及 perf-tools 工具集。最後介紹如何使用 LLTng 以及視覺化工具 TraceCompass GUI，來追蹤與分析 kernel 層。

〈第 10 章：Kernel Panic、Lockup 以及 Hang 〉。本章會詳細解釋 kernel panic 的意思，以及發生 kernel panic 時，kernel 裡的程式碼執行路徑。最重要的是，你將學會如何編寫自定的 kernel panic handler，以便在 kernel panic 時，讓你的程式碼還能夠保持運作。此外，這章的相關主題還有：檢測 kernel 中的鎖死和 CPU / 工作佇列（work queue）停滯，以及 hang（卡住不動）。

〈第 11 章：使用 Kernel GDB（KGDB）〉，本章介紹強大的 KGDB，這是一個 kernel 原始碼層的除錯框架。你將學會如何配置和設定 KGDB，以及如何在程式碼的層面實際利用 KGDB 對 kernel / module 的程式碼除錯、設置 breakpoint、硬體觀察點（hardware watchpoint），以及使用 GDB Python 腳本等。

〈第 12 章：再談談一些 kernel debug 方法〉。這個龐大的 kernel 除錯主題會在本章劃上休止符。建議你可以、而且你有時應該自己也會想要使用一些其他方法，包括：什麼是功能強大且資源密集的 Kdump / crash 工具，有時它會是你的救急神器。接著，介紹靜態分析（static analysis）之所以非常重要的原因，以及用於分析 Linux kernel / module / device driver 程式碼的合適工具，也會介紹程式碼覆蓋率（code coverage）和 kernel 測試框架。章節最後還會透過 journalctl 介紹日誌、kernel assertion（核心斷言）和警告巨集（warning macro）。

CHAPTER 7

Oops！解讀 kernel 的 bug 診斷

Kernel code 應該要是完美的，不應該當機（crash）。但是，當然有時候還是會發生……，歡迎來到現實世界。

當 userspace code 遇到 bug，比如典型的無效記憶體存取時，處理器的**記憶體管理單元（memory management unit, MMU）**會無法透過行程上下文（process context）的分頁表，將無效的 userspace 虛擬位址轉換為實體位址時，就會引發錯誤。然後，kernel 中的錯誤處理常式會接管控制。最終常導致發送致命訊號給故障的 process 或 thread，通常是 SIGSEGV。當然，這可能會使 process 處理訊號並終止。

現在考慮完全相同的情況，只是這一次，無效的記憶體存取是發生在 kernel mode 的 kernel space！嘿，這不應該發生，對吧？確實如此，但是 kernel space 中也會發生 bug，且這次，kernel 的錯誤處理常式在察覺到觸發 bug 的

是 kernel-mode code 時，會執行程式碼來產生 **Oops**，一個詳細說明發生什麼事的 *kernel* 診斷；不幸的 process context 可能也會死掉，這是副作用。

你在這裡將學習到的關鍵主題是「kernel Oops 診斷訊息」，以及更重要的：如何詳細解讀。在這整段過程中，你將產生一個簡單的 kernel Oops，並了解如何準確解讀；此外，也會看到能幫助完成此任務的幾種工具和技術。深入了解 Oops 往往有助於找出 kernel bug 的根本原因！為了幫助你更理解且更能發現典型問題，我們還會討論指出一些實際的 kernel Oops。

本章將重點討論並涵蓋以下主題：

- 產生一個簡單的 kernel bug 和 Oops
- kernel Oops 及其意義
- 魔鬼藏在細節裡：解碼 Oops
- 協助判斷 Oops 位置的工具和技術
- ARM Linux 系統上的 Oops 及使用 Netconsole
- 幾個實際的 Oops

7.1 技術需求

技術需求和工作空間如同第 1 章〈軟體除錯概論〉，程式碼範例也可以在本書的 GitHub repository[1] 找到。唯一的新鮮事是，我們將向你展示如何 clone 和使用有效的 procmap 工具。

7.2 產生一個簡單的 kernel bug 和 Oops

你一定聽過這句話：擒賊先擒王，因此，這裡先來學習如何自己生成 kernel bug，這應該不是多大的挑戰。

1 *https://github.com/PacktPublishing/Linux-Kernel-Debugging*

正如你所知，經典的 bug 教學是惡名昭彰的 NULL pointer dereference；緊接而來的章節「NULL 陷阱分頁到底是什麼？」會詳細說明。因此，計畫如下：

- 先寫一個非常簡單的 kernel module，執行 dereference NULL pointer，位址 `0x0`，我們會在 version 1 的 `oops_tryv1` 模組呼叫它。
- 一旦你嘗試過了，就可以進入稍微複雜一些的 version 2 `oops_tryv2` 模組。這裡將提供 3 種不同的方法來生成 Oops !

在開始 Oops 的生成任務之前，先來好好了解 procmap 工具的功能，以及何謂 NULL trap 分頁。首先就來看看這個工具。

procmap 工具

它能夠視覺化（visualize）kernel 虛擬位址空間（virtual address space, VAS）的完整記憶體映射（memory map），以及任何給定的 process 的使用者 VAS，這就是 procmap 工具的設計目的。先自首：我就是原始作者。

它的 GitHub 頁面 [2] 上的描述總結其功能：

> *procmap* 設計為一個 console / CLI 工具，可用於視覺化 Linux process 的整個記憶體映射，實際上是視覺化 kernel 與 user mode 虛擬位址空間（VAS）的記憶體映射。
>
> 它以垂直平鋪的格式，將虛擬位址以降冪的方式，將給定 process 的整個記憶體映射以簡易視覺化的方式輸出，請見後續螢幕截圖。該腳本具有顯示 kernel 和 userspace 映射的智慧型功能，以及計算並顯示出之後會出現的稀疏記憶體區域（sparse memory region）。此外，每個區段（segment）或映射非常近似地依照相對大小縮放，並以上色編碼以方便閱讀。在 64-bit 的系統上，還會顯示所謂的非規範稀疏區域（non-canonical sparse region）或是空洞（hole），通常在 x86_64 上是接近於驚人的 16,384 PB。

2 *https://github.com/kaiwan/procmap*

此工具包括只查看 kernel space 或 user space、詳細模式以及 debug 模式，將其輸出以方便的 CSV 格式導至指定文件以及其他選項。它還有一個 kernel 元件（一個模組），目前可以在自動偵測的 x86_64、AArch32 和 AArch64 CPU 上運行。

請注意，它在任何實質意義上都不完整，目前仍在開發；有幾個注意事項，歡迎提供回饋與貢獻！

可以從這裡下載或 clone：`https://github.com/kaiwan/procmap`。

NULL 陷阱分頁到底是什麼？

在所有基於 Linux 的系統上，實際上，是在所有現代基於虛擬記憶體的作業系統上，kernel 將可用於一個 process 的虛擬記憶體區域分為兩個部分：「user」和「kernel VAS」，稱之為 VM 分割，《Linux Kernel Programming》的第 7 章〈記憶體的內部管理：基本知識〉有詳細討論。

在 x86_64 平台上，每個 process 的整個 VAS 當然有 2^64 個位元組。就目前而言，這是一個極大的數字。有 16 EB（exabytes），1 exabyte = 1,024 petabytes = 100 萬 terabytes = 十億 gigabytes ！ VAS 實在太大了。因此，在 x86_64 上，kernel 預設的設計是這樣拆分的：

- 大小為 128 TB 的 kernel VAS，錨定在 VAS 的頂部，從 **kernel virtual address（KVA）**`0xffffffffffffffff`， 即 VAS 的 最 頂 部， 到 KVA `0xffff800000000000`
- 大小為 128 TB 的 user VAS，錨定在 VAS 的底部，從 **user virtual address（UVA）**`0x00007fffffffffff` 到 UVA `0x0`，即 VAS 底部

想一想

64-bit 的 VAS 太大了，在這種情況下，最終只使用可位址空間的一小部分。16 EB 是 16,384 PB，在 x86_64 上使用了 128 TB + 128 TB = 256 TB，即 256/1024 = 0.25 PB。這意味著大約使用了可用 VAS 的 0.0015%。

現在，有趣的點在於：user VAS 的最底端，第一個虛擬分頁（virtual page），即從第 0 個位元組到第 4095 個位元組，稱為 **NULL trap page**。我們假設你現在已經安裝了 `procmap` 公用程式，趕快對我們的 shell process 執行這個公用程式，我們這裡 shell process 的 PID 是 1076，接下來可以看到它會顯示 NULL trap page（分頁），查看它顯示的 NULL trap page：

```
$ </path/to/>procmap
 --pid=1076
[...]
```

可以在下面的螢幕截圖看到 NULL trap page：

```
+-----------------------------------------------------------------------+ 000057826c590000
|       /usr/bin/bash  [  36 KB,rw-,p,0x118000]                         |
+-----------------------------------------------------------------------+ 000057826c587000
|       /usr/bin/bash  [  16 KB,r--,p,0x114000]                         |
+-----------------------------------------------------------------------+ 000057826c583000
|       /usr/bin/bash  [ 220 KB,r--,p,0xde000]                          |
|                                                                       |
+-----------------------------------------------------------------------+ 000057826c54c000
|       /usr/bin/bash  [ 708 KB,r-x,p,0x2d000]                          |
|                                                                       |
+-----------------------------------------------------------------------+ 000057826c49b000
|       /usr/bin/bash  [ 180 KB,r--,p,0x0]                              |
|                                                                       |
+-----------------------------------------------------------------------+ 000057826c46e000
|<... Sparse Region ...> [  87.50 TB,---,-,0x0]                         |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
~ .       .       .       .       .       .       .       .       .   ~
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
+-----------------------------------------------------------------------+ 0000000000001000
|       < NULL trap >  [   4 KB,---,-,0x0]                              |
+------------------      U S E R   V A S  start uva  ------------------+ 0000000000000000
VAS mappings:  name    [ size,perms,u:maptype,u:0xfile-offset]
```

圖 7.1　從 procmap 工具的 user VAS 下方的部分螢幕截圖

你在前面的螢幕截圖底部可以看到 NULL trap page，bash process 的某些映射在較高的位置可以看到。NULL trap page 的全部權限 - rwx - 都設置為 ---，因此沒有任何 process 或 thread 可以在其中讀取、寫入或執行任何內容！這就是為什麼當 process 嘗試對位址 0x0 讀取或寫入 NULL byte 時，不會發揮作用。簡而言之，實際發生的情況是：

- 一個 process 試圖存取（讀取 / 寫入 / 執行）或解參考 NULL byte。
- 實際上，存取此分頁中的任何位元組都會導致相同的事件序列，因為 --- 模式會套用在分頁中的每一個位元組資料，這就是為什麼會稱它為 NULL trap page！因為它會捕捉對其中任何位元組資料的存取。
- 分頁中全部位元組資料的權限都是 0：不能讀取、不能寫入、不能執行。除非有快取，否則現在所有虛擬位址都會到達 MMU。MMU 會做出檢查，然後執行執行期的位址轉換，將虛擬位址轉換為實體位址。在這裡，MMU 檢測到分頁中全部位元組的資料都沒有權限，因此引發錯誤。通常在 x86 上是一般的保護錯誤。
- OS 預先安裝了錯誤和陷阱 / 例外處理常式。控制權會傳遞給適當的錯誤處理函式。
- 這個錯誤處理常式函式，在導致錯誤的 process 之 process context 中執行。它透過一個相當複雜的演算法，找出問題所在。
- 在這裡，錯誤處理常式會得出結論，正在執行 user mode 的 process 嘗試進行錯誤的存取。因此，它會發送致命訊號（SIGSEGV）給它，這最終可能導致 process 死亡和區段錯誤（segmentation fault）。[(core dumped)] 訊息會顯示在 console 上。當然，該 process 可以安裝一個 signal handler 來處理，然而，在清理之後它必須終止。

現在，你已經了解 NULL trap page 及其工作原理，以下就來試試根本不應該做的事情：嘗試在 kernel mode 讀取 / 寫入 NULL 位址，進而引發 kernel bug！

簡單的 Oops v1 範例：dereference NULL pointer

這裡是第 1 個簡單版有 bug 的 kernel module，只是單純讀取或寫入 NULL 位址。正如上一節所學，對 NULL trap page 中任何位置的存取「包含讀取、寫入或執行」，都會導致 MMU 跳躍並觸發錯誤。在 kernel mode 下，這件事當然也成立。

以下是充滿 bug 的模組相關程式碼片段，請 clone 本書的 GitHub repo. 瀏覽並自行嘗試！

```
// 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 by writing to
NULL");
```

我們保留一個名為 `try_reading` 的布林模組參數。在預設情況下，它為 `0`（或 off）。如果設置為 `1`（或值為 `yes`），則模組程式碼將嘗試讀取 NULL 位址的內容；如果保持為 `0`，程式碼將檢測到這一點，並嘗試將一個位元組（`'x'`）寫入 NULL 位址。這是初始化函式的程式碼：

```
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 actually optimizes
it away, as we're not working with the result of the read. This makes it appear that
the read does NOT cause an Oops; this ISN'T the case, it does, of course. So, to prove
it, we try and printk the variable, thus forcing the compiler to generate the code, and
voila, we're rewarded with a nice Oops ! */
        pr_info("val = 0x%lx\n", val);
    } else // try writing to NULL
        *(int *)val = 'x';
```

```
    return 0;       /* success */
}
```

這很直觀。請閱讀上面的詳細評論，關於在讀取情況下的編譯器最佳化，以及如何避開這個問題。

當然，關鍵點在於讀取和寫入存取都有問題，如前一節「NULL 陷阱分頁到底是什麼？」詳細描述的。任何嘗試讀取 / 寫入 / 執行 NULL trap page 中的任何位置都不被允許，並會引發錯誤！現在，kernel module 的程式碼，在 insmod 這個 process 的 process context 中執行，將執行到有 bug 的存取。

現在可以想一想：kernel 並不是一個 process，當錯誤處理常式的程式碼偵測到在 **kernel mode** 有一個存在 bug 的存取動作；是的，它可以，而且還真的發生了！並理解「kernel 存在著 bug」，因此觸發了 Oops！我們還提到，其中一部分就是處理 vmalloc 錯誤以及中斷處理。

什麼是 !!<boolean> 語法？

這是使用 C 寫程式的特色：用 `!!<boolean_expression>` 可以保證表示式的值是 `0` 或 `1`，無論傳遞什麼值，例如，傳遞 `5` 變成 `!!(5)`，現在 `!5` 就是 `0` 而 `!0` 就是 `1`，超聰明！

以下部分螢幕截圖顯示寫入 kernel 日誌的 Oops 訊息前面一部分。不用擔心，我們肯定會涵蓋其餘的部分並學習如何詳細解讀，現在只是看看而已。在這個範例中，嘗試寫入 NULL byte 並觸發 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]
```

圖 7-2　部分的螢幕截圖顯示，透過 oops_trymodule 試著寫入 NULL 位址時，會導致經典的 Oops

在 kernel 日誌中的一行開頭是 Oops，可見黑色區塊；這表示在 Oops 之後 kernel printk 印的都是 Oops 的診斷訊息。

重新開機：有用但愚蠢的權宜之計（workaround）

你有沒有注意到，一旦出現 bug，kernel module 就無法透過 rmmod 卸載，這是因為引用計數器（reference count）不為 0 嗎？ lsmod 驗證了這一點：

```
$ lsmod |grep oops
oops_tryv1             16384  1
```

這通常是因為 Oops 發生時優先權高於 process context 執行 exit，若以我們的情況來說則是 insmod，因此 module 的 reference count 會減到變成 0。在之前的輸出內容中，最右邊的那個 1 表示目前的 module reference count，所以要避免卸載這個 module。

現在，如果你無法卸載 module，就無法再次掛載它（在編輯原始碼後重試）。正確方法是重新開機重頭開始。這個煩人的問題有一個非常愚蠢的解決方法，就是簡單地清理（make clean），重新命名程式碼，編輯 Makefile 以使用新名稱，然後再編譯。這樣，它就可以用新的名字掛載！雖然很蠢，但開發中和急著驗證時非常方便。

再做一些 Oops：充滿 bug 的 module v2

如本章開頭所述，在 v2 錯誤模組中，我們將做一些稍顯實際的事情，並希望觸發 kernel Oops。此模組有 3 種不同的觸發 Oops 方式：

- 第 1 種：透過寫入在 NULL trap page 中隨機生成的 KVA。
- 第 2 種：透過允許使用者傳遞隨機且無效的 KVA，並嘗試將某些內容寫入其中。可以利用 procmap 工具找到無效的 KVA。
- 第 3 種：啟動一個簡單的工作佇列函式（workqueue function）。將有一個 kernel worker thread 在排班到時會執行其程式碼。在工作佇列函式中，嘗

試將某些內容寫入 structure 的成員，其中 structure pointer 為 NULL。由於這種情況可能非常真實，因此本章幾乎都會以它作為使用案例。

先使用上面的第 1 種方法觸發 Oops ！

案例 1：透過在 NULL trap page 寫入隨機位置而引起的 Oops

由於非常類似於第一個 v1 module，所以我不會深入探討這個問題，就是使用 kernel 介面（`get_random_bytes()` API）生成一個隨機數，並透過使用 modulo 運算符將其縮小為 0 到 4,095 之間的數字。模組的 init 函式的相關程式碼如下：

```
// 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 invalid kernel
address in NULL trap page\n");
           get_random_bytes(&page0_randptr, sizeof(unsigned int));
      bad_kva = (page0_randptr %= PAGE_SIZE);
    }
    pr_info("bad_kva = 0x%lx; now writing to it...\n", bad_kva);
    *(unsigned long *)bad_kva = 0xdead;
[...]
```

以上的最後一行是寫入這個 bad KVA 的嘗試，當然，這會觸發一個 Oops。想嘗試的話，只需在不傳遞任何參數的情況下 `insmod` 模組。這將使程式碼進入這個使用案例，可以自行嘗試並查看 kernel 日誌。

案例 2：透過寫入到 kernel VAS 中的無效未映射位置來引發 Oops

第 2 個使用案例有一個名為 `mp_randaddr` 的模組參數。若要執行，需要用一般方式將參數傳遞給模組，並將其設置為無效的 kernel 位址或 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; deliberately
invalid, to cause an Oops!");
```

當模組的 init 函式檢測到你傳遞了一個非零值給這個參數時，它會執行以下程式碼：

```
} else if (mp_randaddr) {
        pr_info("Generating Oops by attempting to write to the invalid kernel address
passed\n");
        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;
```

這幾乎與第 1 種情況相同；有趣的是：我怎麼知道要傳遞哪個 kernel 位址或 KVA？我怎麼知道它在 kernel VAS 中無效或未映射的位置？

啊，這就是 procmap 工具發揮作用的地方！只需執行 `procmap`：傳遞任何 PID 並指定 `--only-kernel` 選項開關，現在不用關心 user VAS。以下是我在 x86_64 guest VM 上叫用它的方式，你需要更新 `PATH` 環境變數以引入安裝 procmap 的目錄：

```
$ procmap --pid=1 --only-kernel
...
```

這裡是它顯示的輸出部分截圖，重點是 kernel VAS 的上半部：

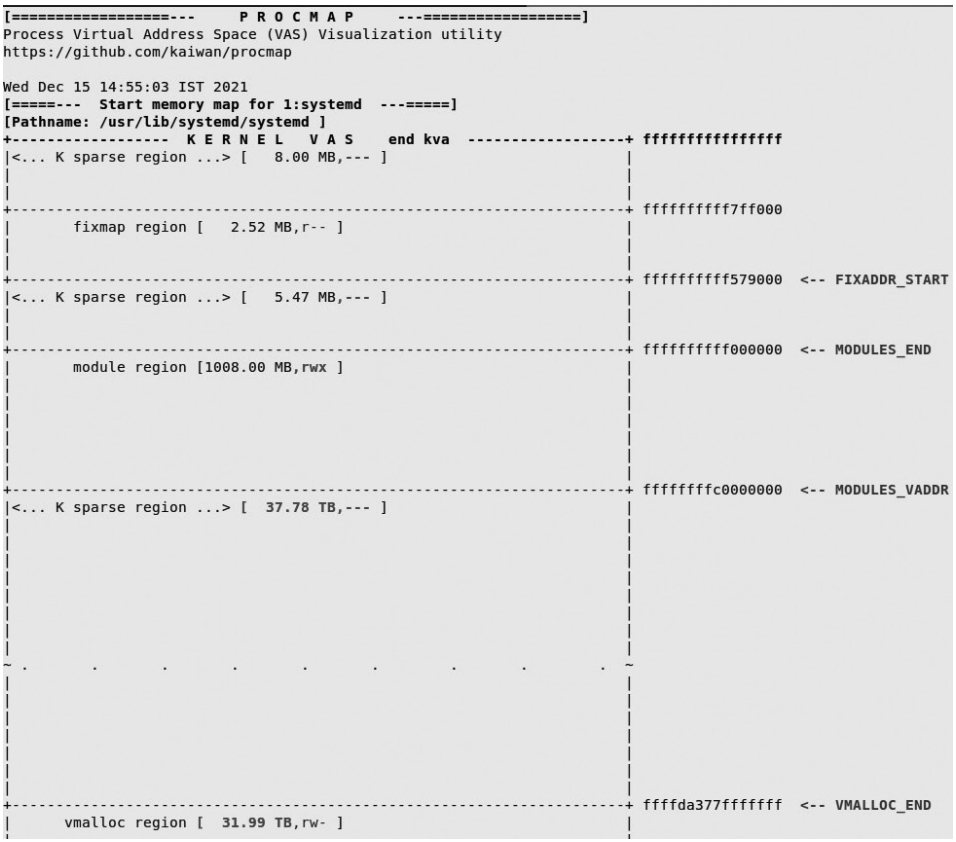

圖 7.3　部分螢幕截圖顯示 procmap 工具的輸出，重點是 kernel VAS 的上半部，有一些稀疏（未映射）區域明顯可見

好的，仔細看前面的螢幕截圖。有標記為 `<... K sparse region ...>` 的區域是 kernel VAS 中的空洞。這裡沒有映射任何內容，這很常見，這樣的記憶體通常被稱為稀疏區域或位址空間中的空洞。

重點是：稀疏區域是未映射的區域，因此，如果你嘗試以任何方式存取這些位置，不管是讀取、寫入或執行，那就會是一個 bug！因此，在稀疏區域內選擇一個 KVA，我會選擇一個在模組區域，既 kernel 模組所存在之處；和 kernel vmalloc 區域，即 vmalloc() 從中分配記憶體之處，這兩者即 `0xffffffffc0000000` 和 `0xffffda377fffffff` 之間的任何位址。因此，我將使用 KVA `0xffffffffc000dead` 作為無效 kernel 位址的值並執行。

好的，確保你已經編譯了 oops_tryv2 模組，並搭配剛剛討論的參數，載入這個模組。

```
$ modinfo -p ./oops_tryv2.ko
mp_randaddr:Random non-zero kernel virtual address; deliberately invalid, to cause an
Oops! (ulong)
bug_in_workq:Trigger an Oops-generating bug in our workqueue function (bool)
$
```

使用 modinfo 工具可以顯示我們的模組能接受兩個參數，現在請先忽略第二個參數，那是下一個主題。讓我們開始吧，終於！

```
$ sudo insmod ./oops_tryv2.ko  mp_randaddr=0xffffffffc000dead
Killed
$
```

啊哈！這個模組藉由模組參數傳遞，故意試圖寫入無效的 kernel 位址 0xffffffffc000dead，卻遇到一個有 bug 的結果。如我們所願，出現一個 Oops，以下螢幕截圖顯示其中一部分：

```
[49132.584848] oops_tryv2:try_oops_init():92: Generating Oops by attempting to write to the invalid kernel a
ddress 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: ffffffffc000dead
[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: innotek 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:ffffba3783dffc20 EFLAGS: 00010246
[49132.591423] RAX: ffffffffc000dead RBX: 0000000000000000 RCX: 0000000000000000
[49132.591954] RDX: 0000000000000000 RSI: 0000000000000027 RDI: 00000000ffffffff
[49132.592398] RBP: ffffba3783dffc38 R08: 0000000000000000 R09: ffffba3780e9f020
[49132.592864] R10: 0000000000000001 R11: 00000000ffffffff R12: ffffffffc0604000
[49132.593322] R13: ffff8f90766f6530 R14: ffffba3783dffe70 R15: ffffffffc06da158
[49132.593769] FS:  0000785ef7e11540(0000) GS:ffff8f90bdd40000(0000) knlGS:0000000000000000
[49132.594258] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[49132.594711] CR2: ffffffffc000dead CR3: 000000005a5e4001 CR4: 0000000000706e0
[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 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 f0 ff ff 73 01 c3 48 8b 0d c3 f5 0c 00 f7 d8 64 89 01 48
```

圖 7.4　螢幕截圖顯示嘗試寫入無效 / 未映射核心位址所生成的 Oops

即使 Oops 不是 kernel panic，根據情況和 bug 的嚴重程度，kernel 也可能變得沒有反應、不穩定或兩者兼具。或者它可能會繼續運作，就好像沒有發生任何警報一樣！無論如何，Oops 最終都是一個 kernel-level bug，要能夠檢測、解譯與修復它！

現在就來看看如何詳細解釋 Oops kernel 的輸出，開始吧！

7.4 魔鬼藏在細節裡：解碼 Oops

以下將使用「案例 3：當結構指標為 NULL 時，透過寫入結構成員觸發 Oops」這節的使用 / 測試案例，討論第三種情境。簡單回顧一下，這是我們觸發此特定 kernel Oops（案例 # 3）的方法：

```
cd ch7/oops_tryv2
make
sudo insmod ./oops_tryv2.ko bug_in_workq=yes
```

正如之前所看到的，它會觸發 Oops。有趣的部分在這：一步步逐行解讀 Oops。

開始之前要先了解，下面的詳細討論當然是依架構而定的，這裡和目前與 x86_64 平台有關，因為 Oops 輸出部分當然非常取決於架構，後面還會展示典型 ARM 平台上出現 Oops 的方法。

Oops 的逐行解釋

Oops 的初始化與真正關鍵的部分請參考圖 7.5。為了便於逐行參考，這裡是同一個螢幕截圖的註釋圖，只是放大一些以便更清晰：

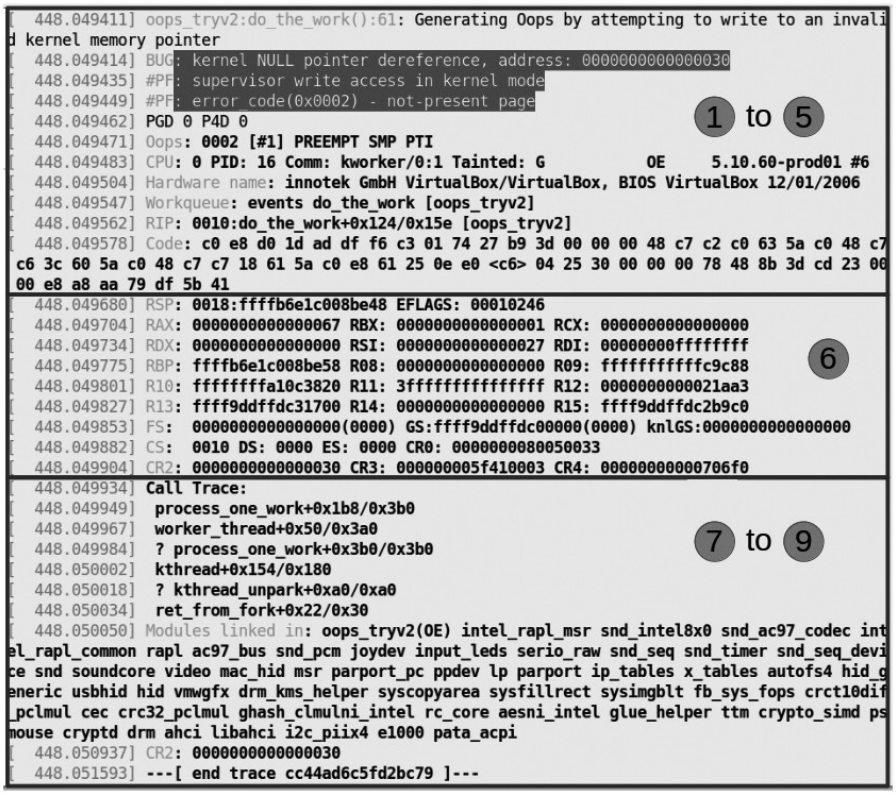

圖 7.6　有註解的完整螢幕截圖：案例 3 的 oops_tryv2 工作佇列功能 bug 所引發的 Oops 輸出

為了更清晰易懂，這個圖表和討論可以分成幾個部分，透過圖 7.6 的矩形會看得更明顯。以下是第一部分：

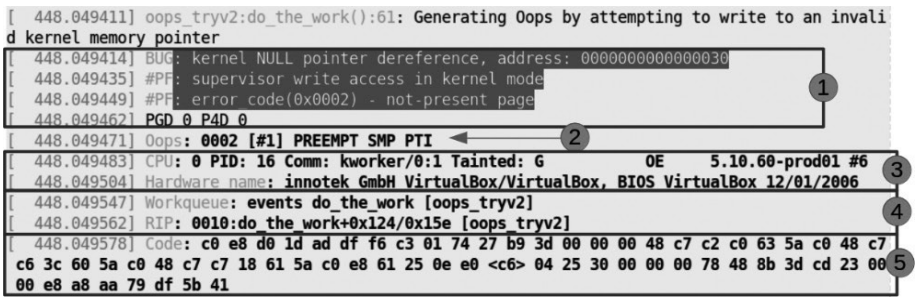

圖 7.7　3 張螢幕截圖註釋圖中的第 1 張：oops_tryv2 工作佇列函式 bug 引發的 Oops 輸出

好了，來深入了解細節！從這裡開始顯示的資料是取決於架構而定的，僅適用於 x86 平台。

解讀 Oops 的第 1 行

在圖 7.7 中標記為 1 的矩形內，有一個深色標示的程式碼，是針對 x86_64 的特定架構。這是作業系統錯誤處理常式的一部分程式碼，當在 kernel 中檢測到異常條件，即 bug 時，就會開始編寫 Oops 診斷訊息。實際 x86 錯誤處理程式碼的一部分為：

```
// 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);
```

檢查前面的 if 條件，可以很清楚為什麼會得到這個輸出：

```
BUG: kernel NULL pointer dereference, address: 0000000000000030
```

當出錯的位址是在第一個分頁之內，而且正在 kernel-mode 內執行時，就會發出它。請回想，user VAS 的第一個分頁是 NULL trap page，這個分頁的全部位址都在 PAGE_SIZE 之內，通常為 4,096 個位元組。若條件為 false，則 kernel 會印出一個替代訊息；此外，這裡導致錯誤的位址，也就是在第一個 NULL trap page 內的位址會接著印出來，且一直都是十六進制，它在這裡的值是 0x30。

還有一點很重要：為什麼是 0x30 而不是 0x0 ？回想一下生成此特定 Oops 的程式碼，可以參考「案例 3：當結構指標為 NULL 時，透過寫入結構成員觸發 Oops」那節。程式碼有 bug 的那行在這：

```
ch7/oops_tryv2/oops_tryv2.c:do_the_work():oopsie->data = 'x';
```

現在，oopsie 是這個程式碼中指向 st_ctx 結構的指標，但其值為 NULL；請記住，從未賦值給它。因此，0x30 的值是從結構的起點，到所參考的結構成員之間的偏移量（offset）！這裡學到的一個關鍵點是，*當出錯的位址顯示為小於分頁大小的小整數值時，例如此例，很可能結構或其他指標的值是 NULL，而顯示的數值是從結構，或可能是陣列或諸如此類的起點，到所參考的結構成員之間的偏移量。*

Oops 中的下一行輸出是：

```
#PF: supervisor write access in kernel mode
```

這是從相同的 show_fault_oops() 函式中後續程式碼所生成的；順帶一提，PF 代表 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");
```

花點時間閱讀程式碼，並將其比對所獲得的輸出，可清楚顯示，kernel 已經弄清很多事情：程式碼是在監督模式（supervisor mode）執行的，這表示 kernel mode，並且有一個寫入嘗試，再次在 kernel mode 下執行。

下一行為：

```
#PF: error_code(0x0002) - not-present page
```

很快就會講解這裡的意義，這是來自 show_fault_oops() 函式內的程式碼，該函式緊接著前面的程式碼：

```
        pr_alert("#PF: error_code(0x%04lx) -
             %s\n", error_code,
  !(error_code & X86_PF_PROT) ? "not-present page" :
  (error_code & X86_PF_RSVD)  ? "reserved bit violation" :
  (error_code & X86_PF_PK)    ? "protection keys violation" : "permissions violation");
```

因此，弄清楚這三行輸出中的每一行產生方式。為什麼顏色是深色背景？啊，這很簡單：dmesg 能解釋 pr_alert() 日誌的層級，並給予相對應的顏色。

我們將跳過**分頁全域目錄（page global directory, PGD）**與 *P4D* 的細節。在 4.11 Linux 中，P4D 是位於 PGD 與**分頁上層目錄（page upper directory, PUD）**之間的一個級別。這些是參考正在執行的 process 所在的 process context 分頁表（page table），有興趣的話，請參閱 dump_pagetable() kernel 函式的程式碼。

解讀 Oops 的第 2 行

為了方便起見，此部分截圖複製自圖 7.7，Oops 的第 2 行輸出如下。

圖 7.8　測試案例 3，有 bug 的 oops_tryv2 模組第 2 行 Oops 輸出

顯然，這裡看得出發生了 Oops，grep 字串 Oops 可能很有幫助，不會那麼荒謬。緊接此字串後面的數字，即這裡的 0002 很重要，這是特定架構的 Oops 位元遮罩（bitmask），學習解讀它將獲益良多。

❖ 解讀（特定架構的）Oops bitmask

從這裡開始，負責顯示 Oops 內容的整個函式名為 arch/x86/kernel/dumpstack.c:__die()。它分為兩部分：__die_header() 和 __die_body() 函式。Oops bitmask 和該行上剩餘的令牌（token），會從標頭函式顯示，例如 PREEMPT SMP …。

為了讓你了解實際 Oops 診斷顯示的 kernel 程式碼工作方式，可見 `__die_header()` 函式螢幕截圖（在 5.10.60 kernel），原始碼檔案為：`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%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);
```

圖 7.9　x86_64 上 kernel Oops 輸出功能的部分螢幕截圖

如上所述，取決於 **arch** 的 **Oops bitmask** 實際上非常有意義，進一步提示發生 kernel bug 的原因！

你可以從前面的 `printk()` 看到它，是 printk 格式字串 `%04lx` 的第二部分，對應於 `err & 0xffff` 程式碼。要如何解釋這個位元遮罩？這裡有解釋，僅適用於 x86 平台；但是，請記住：它是架構特定的（arch-specific）。

MMU 使用編碼過的值來設置分頁錯誤。在 x86 平台上，這是分頁錯誤的 error code bits 設定方式：

```
bit 0 == 0: no page f，ound            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
```

這些資訊實際上曾經是程式碼庫（code base）中較早的 kernel 版本註解。

也許更好的方法是以表格格式檢查分頁 error code 的 5 個**最低有效位元（least significant bit, LSB）**，以更輕鬆地視覺化和解釋特定錯誤，即 Oops 發生的原因：

表 7.1：在 x86 平台上分頁 error code 的 LSB 5 bits 含義

Value of bit	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	-na-	-na-	kernel mode	Read attempt	No page found
1	Instruction fetch fault	Reserved bit used	user mode	Write attempt	Protection fault

因此，現在就很簡單了！我們得到了 Oops 位元遮罩為 `0002`，它是十六進制，請見 `printk:` 格式指定符是 `%04lx`；轉為二進位是 `00010`。根據前面的表格，這意味著以下內容；這裡的 bit 3 和 bit 4 為 0，所以不重要：

```
Bit 2 is 0 : kernel mode
Bit 1 is 1 : write attempt
Bit 0 is 0 : no page found
```

嗯，這裡沒有驚喜，這正是 Oops 診斷會說的事，即圖 7.7 的第 1 點：

```
#PF: supervisor write access in kernel mode
#PF: error_code(0x0002) - not-present page
```

該行的其餘部分如下：...

```
...  [#1] PREEMPT SMP PTI
```

這行很容易解讀：

- `[#1]`: 這是在此系統作業階段（session）期間發生的 Oops 次數。
- `[#1]` 告訴我們這是第一個 Oops。它是一個 session 值，重新開機會重置這個值。
- `PREEMPT`: 程式碼執行的 kernel 組態設定為可搶占（preemption），（`CONFIG_PREMPT=y`）。
- `SMP`：kernel 有啟用 **SMP（Symmetric Multi-Processing）**，支援多核心（multicore）。
- `PTI`：**PTI** 是**分頁表隔離（Page Table Isolation）**的簡稱。在 2018 年初的 **Meltdown/Spectre** 硬體漏洞促使 kernel 社群建立一個名為 PTI 的保護機制，以防止這種嚴重漏洞。詳細資訊可參閱「深入閱讀」。

繼續來解讀 Oops 診斷中的以下兩行。

解讀 Oops 的第 3 行

為了方便起見，此部分截圖複製自圖 7.7：

```
[  448.049483] CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G           OE     5.10.60-prod01 #6  3
[  448.049504] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
```

圖 7.10　測試案例 3，有 bug 的 oops_tryv2 模組第 3 行 Oops 輸出

這幾行以及一些細節，基本上說明 process context：在 kernel mode 下，process 或 thread 執行會導致故障且充滿 bug 的程式碼。我們將一次取一個 token：

- `CPU`：表示程式碼在發生 Oops 時，所在的 CPU core；這裡是 CPU 0。
- `PID`：發生 Oops 時正在執行程式碼的 process 或 thread 的 PID。
- `Comm`：發生 Oops 時正在執行程式碼的 process 或 thread 名稱。
- `Tainted`：kernel 汙染旗標的位元遮罩，很快就會介紹這點。
- `uname -r` 的輸出，包含 kernel release 版本，並接著以 # 符號為前綴的數字，是這個 kernel 已經編譯的次數；這裡是 6。

需要注意的是，當 Oops 從中斷的上下文（interrupt context）觸發時，這裡看到的部分資料會變得可疑。之後的章節中會詳細介紹這點，即在 IRQ context 發生 Oops 後，使用 console 裝置取得 kernel 日誌，會檢查 interrupt context 中觸發的 Oops。

❖ 解讀 kernel 汙染旗標

Linux kernel 社群喜歡知道運行的 kernel 乾不乾淨，骯髒或受到汙染的 kernel，指的是不處於原始狀態的 kernel。這個狀態訊息保留於位元遮罩中的位元值。整個位元遮罩，目前總共包含 18 個位元或旗標，即稱為**受汙染的旗標（tainted flags）**。

在此特定範例中，受汙染的旗標會像以下這樣出現，在字串 `Tainted:` 之後，這裡以 highlight 凸顯出來：

```
CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G           OE     5.10.60-prod01 #6
```

可以根據以下顯示的字母，即「受汙染旗標」來解讀：

表 7.2：解譯 kernel 受汙染的旗標

Bit #	記錄為：當該位元是清除（_）或設定（X）時	若位元設定為（1）時的意義
0	G 或 P	載入專屬模組（P），或只載入 GPL 模組（G）。
1	_ 或 F	強制載入一個或多個模組。
2	_ 或 S	超出系統規範。
3	_ 或 R	強制卸載一個或多個模組。
4	_ 或 M	來自一個 CPU core 回報的**機器檢查異常（Machine Check Exception, MCE）**。
5	_ 或 B	錯誤的分頁參考或不預期的分頁旗標。
6	_ 或 U	使用者空間的應用程式要求要設定 taint。
7	_ 或 D	Kernel 最近掛了，因 Oops 或 `BUG()`。

Bit #	記錄為：當該位元是清除（_）或設定（X）時	若位元設定為（1）時的意義
8	_ 或 A	使用者覆蓋了 ACPI 表格。
9	_ 或 W	Kernel 透過某個 WARN*() 巨集發出警告。
10	_ 或 C	載入階段性及實驗性質的驅動程式。
11	_ 或 I	平台韌體的 bug，已經使用權宜之計（workaround）。
12	_ 或 O	載入 kernel tree 之外 / 外部編譯的模組。
13	_ 或 E	載入未經簽章（unsigned）的模組。
14	_ 或 L	發生一個軟式上鎖（soft lockup）。
15	_ 或 K	Kernel 已經套用了 live patch。
16	_ 或 X	發行版（distros）使用這個旗標，稱之為輔助的汙點（auxiliary taint）。
17	_ 或 T	Kernel 在啟用結構隨機化（structure randomization）的條件下編譯。

在第 2 行中的 _ 符號意指空白，表示這個指定的 taint bit 已經清除。請注意，Oops 中的汙染旗標會依照要求輸出空格，以顯示特定位元已取消設置：`Tainted: G            OE`。

因此，此例中，G|O|E 汙染旗標意指已加載所有 GPL 模組：G、已加載一個或多個外部編譯的（*out-of-tree*）模組：O，以及已加載一個或多個未簽章的模組：E。確實，我們的 oops_tryv2 模組具有雙重授權，包括 GPL、是一個外部編譯的模組、而且沒有簽章。

因此，查表並找出汙染旗標的含義並不難，使用輔助 script 會更加容易！這正是 kernel source tree 中，tools/debugging/kernel-chktaint script 的設計目的，「乾淨了嗎？kernel-chktaint script」這一節會介紹這個 script 的使用方法。

官方 kernel 文件也有涵蓋這些旗標以及更深入的細節，可見文件：https://www.kernel.org/doc/html/latest/admin-guide/tainted-kernels.html。