O'REILLY®

Debug Hacks中文版

——深入调试的技术和工具

吉岡弘隆 大和一洋 著
大岩尚宏 安部東洋 吉田俊輔

马晶慧 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是Miracle Linux的创始人吉冈弘隆和几位工程师们多年从事内核开发的经验积累。从调试器的基本使用方法、汇编的基础知识开始，到内核错误信息捕捉、应用程序调试、内核调试，本书深入浅出地讲解了Linux下应用程序和内核的调试技巧。

虽然本书的出发点是Linux内核调试，但书中的绝大部分知识在许多领域都能派上用场。如Linux应用程序开发，嵌入式Linux开发，甚至时下流行的iOS应用程序开发，只要从事应用程序开发的工作，就会涉及调试，那么读一读本书也不无裨益。

图书在版编目（CIP）数据

Debug Hacks中文版：深入调试的技术和工具/（日）吉冈弘隆等著；马晶慧译.—北京：电子工业出版社，2011.9

ISBN 978-7-121-14048-8

Ⅰ.① D… Ⅱ.① 吉… ② 马… Ⅲ.① 程序开发工具Ⅳ.① TP311.52

中国版本图书馆CIP数据核字（2011）第134202号

策划编辑：张春雨
责任编辑：付 睿
封面设计：O'Reilly Japan 张健
印 刷：
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036
开 本：720×1000 1/16 印张：26.5 字数：572千字
印 次：2011年9月第1次印刷
印 数：4000册 定价：69.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

Debug Hacks 推荐序

凡是程序就有 bug。bug 总是出现在预料之外的地方。世界上第一个 bug 据说是继电器式计算机中飞进的一只蛾子，倒霉的飞蛾被夹在继电器之间，导致了计算机故障。由于这个小插曲，程序中的错误就被称为了 bug。传闻，后来的 COBOL 创始人 Grace Hopper 这位巾帼英雄为纪念这只飞蛾，在日记里记下了这段故事。

从那时起到现在已经过了半个世纪，而 bug 还在不断地产生。程序员之间流传的“格言”之一就是“程序的执行，是根据编写而不是想象”。而且人总是会出错的，程序中必然会混入 bug。从某种意义上来说，也许 bug 昭示了人类的极限。绝大部分 bug 都是给程序员带来小小烦恼的可爱的小东西，但最近 bug 引起的“事故”在报纸上引发热议的事情也时有发生。

程序员的工作就是写程序，然而随着计算机融入社会、程序越来越复杂，要想写出完美的程序简直难于上青天，因此任何程序都必须面对 bug。个人感觉，程序员的一大半时间都花在了寻找 bug、修改 bug 上。

但是，仅仅在黑暗中寻找 bug 是不会有任何效果的。bug 也有 bug 的寻找方法和修改方法。特别是在寻找并确定 bug 位置方面，有着各种各样的技术。“调试”这个词给人的印象是“改正 bug”，但实际上，确定了位置的 bug 完全不可怕，基本上都能立即改正。调试的精髓就是发现并确定 bug 的位置。本书汇集了身经百战的程序员们从经验中得来的 bug 寻找方法和修改方法，特别是平时看不到的有关 Linux 本身的 bug 的 hack，更是无价之宝。可能某些 hack 并非大多数程序员日常用得到的，但即便如此，其思路也有参考价值。特别感激的是本书详细地解释了 GDB、

Valgrind、oprofile 等方便的工具。此外，本书还涉及 Ruby 的两个“bug”，对此我也十分感激。

希望本书能成为诸位程序员的“启明灯”，让我们在每天与 bug 的斗争中更顺利。

2009 年 3 月于羽田空港

松本行弘[注1]

注 1：松本行弘（Yukihiro Matsumoto），Ruby 语言创始人。——译者注

鸣　谢

关于作者

吉冈弘隆（Hiro Yoshioka）

Miracle Linux 的程序员，毕业于庆应义塾大学研究生院。从 1999 年开始主持名为“内核读书会”的 Linux 技术交流会，曾就职于日本数字设备公司（DEC）研发中心、日本甲骨文，后创立 Miracle Linux。大学毕业后参与了多款软件产品（COBOL 日文版、DEC Rdb、Oracle 8、MIRACLE LINUX、Asianux 等）的开发。

吉冈弘隆还担任了 JIS X0208:1990/X0212:1990 的标准化，U-20 编程大赛的评委，以及安全编程夏令营的部门主审。

2008 年，吉冈弘隆获得了经济产业省商务情报政策局长的感谢信和乐天技术奖 2008 年金奖。

博客“ユメのチカラ”（梦的力量）：http://blog.miraclelinux.com/yume/。

“未来のいつか/hyoshiok の日記”（有朝一日/hyoshiok 的日记）：http://d.hatena.ne.jp/hyoshiok/。

大和一洋（Kazuhiro Yamato）

任职于 Miracle Linux 的软件工程师。之前主要工作内容为 Linux 内核和 GLIBC 周边，最近也开始着力于 gstreamer 等媒体相关 hack。不善使用 GUI 工具，开发环境完全使用 VIM + GCC + GDB。

大岩尚宏（Naohiro Ooiwa）

任职于 Miracle Linux 的软件工程师。从事开发业务，但更多的是调查并分析内核，负责解决驱动程序、网络、文件系统等大范围的许多 bug，号称任何 bug 都能解决，近期在验证 Linux 的节能功能。擅长绘画，但不善日语和英语。经常被上司催促给社区出补丁，与日本的 Linux 开发者代表人物见面的机会很多。

安部东洋（Toyo Abe）

任职于 Miracle Linux 的软件工程师。初入 Linux 内核界时连 Hello World 程序都写不好，那段经历真是让人几欲落泪。到目前只涉及了 x86 架构，写完本书之后打算挑战 ARM。

吉田俊辅（Shunsuke Yoshida）

任职于 Miracle Linux 的系统工程师，不论在哪里都自称为普通人。曾先后就职于地方性软件企业和制造业系统集成企业，后加盟 Miracle Linux。主要负责操作系统、数据库、网络、虚拟化等基础设施。参加了小江户 LUG [注1]、YLUG（横滨 Linux User Group）、USAGI 补完计划[注2]等关东附近的开源社区。参加过各种活动、展出，以及写书。

博客“第三のペンギン”（第三只企鹅）：http://blog.miraclelinux.com/thethird/。

关于贡献者

岛本裕志（Hiroshi Shimamoto）

软件工程师。主要通过解决问题来学习 Linux，特长为分析 core、crash。活跃在 x86、任务调度（scheduler）、实时系统等领域的 Linux 内核社区中。

注 1：小江户らぐ，全称小江户 Linux User Group，以日本崎玉县川越市为中心的 Linux 用户组。此处译为小江户 LUG。——译者注

注 2：USAGI（UniverSAl playGround for Ipv6）项目致力于开发 Linux 上的高质量 IPv6 和 IPsec（用于 IPv4 和 IPv6）的协议栈，而 USAGI 补完计划是改善该项目的俱乐部。——译者注

美田晃伸（Akinobu Mita）

Fixstars 公司程序员。不太明白调试器的用法，调试时使用 printf[注3]。

致谢

本书的灵感来源于传世之作《BINARY HACKS——ハッカー秘伝のテクニック 100 選》[注4]一书。执笔本书的不仅有 Miracle Linux 的员工，贡献者岛本裕志、美田晃伸也提供了很有意思的 hack。在此表示衷心的感谢。

此外，本书的推荐序来自于 Ruby 编程语言的缔造者松本行弘。在此表示真挚的感谢。

——吉冈弘隆

注 3： Linux 内核的 Fault Injection 的作者和维护者。

注 4： 译者注：本书的中文版《Binary hacks——黑客秘笈 100 选》已由中国电力出版社出版，ISBN：9787508387932。

前 言

本书主要内容是程序员在编程过程中无法避免的调试过程。虽然调试技术不依赖于任何编程语言和开发环境，任何编程都无法避免，但调试领域却很难总结，也几乎没有任何参考书。

编程入门读物多如牛毛，为何调试的入门读物却屈指可数？

思考一下编程中的设计、编码、测试和调试等过程，就会发现在调试上花费大量时间的情况并不罕见。实际上，软件开发成本中的大部分并没有用在设计新软件上，而是用在扩展、改变软件及修改软件错误上。

与设计新软件相比，有时会感到调试更加困难。尽管设计、编码和测试相关的最佳实践以书籍的形式广为流传，但软件开发的重要一环——调试，相关的入门书籍却寥寥无几，真是有点奇怪。

调试是种极端个性化的工作，10 个人就会有 10 种调试方法。而且，调试中有高手，也有菜鸟，也有像拥有魔法一样，哼着歌就能找到 bug 并改正的 hacker。

执笔本书之际，心底的想法之一就是以我们遇到过的事情为中心，明确地介绍具体的调试方法。

我们写下我们自己的调试方法，同时也加深了对自己的调试方法的理解。为什么使用这条命令调试？最初是怎样找到这个 bug 的？经过这种反反复复的自问自答，调试的过程也经历了考验。

每条 hack 都是实际工作中遇到的事情。有的例子中为了简化说明，重新书写了测试程序等，但那些也都是在实际遇到的 bug 的基础上进行的说明。举出实际例子，就可以写出经验之谈，避免纸上谈兵了。

在写出的调试过程中，哪怕我们的调试方法还有很多改善余地，也要明确地写出来，起到抛砖引玉的作用，这种写法尽管质朴却十分重要。倒不如说，我们相信正是不断积累这种拙劣的做法，调试方法才会进步。

比我们的方法更好的方法一定存在。讨论好的调试方法需要一个舞台作为基础，这正是本书永恒不变的愿望。特别是希望诸位 hacker 们（大牛程序员）与自己的方式对比着阅读，针对我们的方法的适用范围、优缺点等各种观点进行讨论，这才是我们专业程序员对调试更深入的理解。

调试方法，与其说是本书中明确写出的东西，不如说是从各自的经验中得来的、某种秘籍一样的存在。作为程序员，在不断的钻研和积累中掌握调试技术，才是我们专业程序员的基本能力。

此外，像我们这种把自己的调试方法广泛公开的行为，如果能普遍流传的话，就能与更多的人共享最佳实践，而这些实践是我们程序员的无价之宝。

随着工具和开发环境的进步，今后调试方法也会发生变化，而且调试方法也会随着我们自身对于调试和编程风格的理解而逐步进化。为了主动地学习这些知识，我们希望通过本书，和诸位程序员一起学习进步。

阅读本书的必要知识和读者对象

本书的读者对象是主要用 C/C++等编程语言进行开发的应用程序程序员和 Linux 内核开发者。本书并没有限定语言和开发环境，但在示例中均使用 Linux。要进行底层调试，计算机体系的基础知识、编程语言的基础知识是必要的。此外，开发环境方面，UNIX 编程环境的基础知识也是必要的。除此之外的知识并没有特别要求。

本书读者对象为自己进行编程设计、实现、测试和调试工作的初级到中级程序员。本书面向那些希望加强自己的编程技术的人。不仅是 C/C++程序员，对于使用 Perl/PHP/Python/Ruby 等脚本语言进行编程的人们来说，哪怕语言和工具各不相同，本书中的许多方法也具有参考价值。此外，对于使用 Windows、Mac 等不同平台的人来说，本书的思考方式同样可以参考。

我们特别希望诸位学生们阅读本书。如果在读过一遍编程语言入门书之后，希望更深入地理解编程的话，本书中的 hack 可以用来参考。我在编辑本书时就在想，要是在学生时代就有这样一本书该多好啊。

使用脚本语言编程的人们，平常几乎不会意识到计算机体系和机器语言等，但是，如果 Ruby 处理程序发生 segmentation fault 而突然崩溃，为改正错误，就要用到本书中的知识和技术。对于想把程序员之路走得更宽的人来说，本书是个好的起点。

此外，我们也希望拥有各自调试风格的熟练的 hacker 们阅读本书，并提出宝贵的意见。特别是，市面上几乎没有正面讨论 Linux 内核调试的参考书，因此在我们给本书设置的范围、读者对象及书籍结构等方面，如果能获得您的意见反馈，真是不胜荣幸。

本书内容范围

我们选择的例子主要是 Linux 上的应用程序和 Linux 内核本身，其原因只是我们在从事这方面的工作。

说起编程，还有 Web 应用程序、嵌入式、游戏、中间件等各种各样的应用领域。尽管各个领域都有特殊的调试方法，但本书并不涉及。虽然包罗万象的全能调试方法并不存在，但本书主要讨论更为通用的调试方法。因此，许多情况下这种思路可以灵活运用。

本书包含的内容

本书介绍调试的基本思路和方法。不仅包括对应用程序的调试，也包括对操作系统（Linux 内核）的调试。此外也会涉及 GDB 等调试器的使用方法、转储文件（dump）的查看方法、crash 的使用方法，以及 kprobes 和 oprofile 等调试工具。

本书涉及的工具之外，还有许多优秀工具，例如 ftrace、LTTng、dmalloc、blktrace、lockdep、kgdb、KDB、utrace、lockmeter、mpatrol、e1000_dump、git-bisect、kmemcheck 等。关于这些工具，我们期待看到诸位读者的 Debug hacks。

本书不包含的内容

本书不包含编程的一般方法，例如软件设计、容易调试的编码方法和测试方法论等。TDD（Test-driven development，测试驱动开发）是集测试和调试于一身的开发过程，但并不在本书的讨论范围。

此外，本书也不涉及一般疑难解答（troubleshooting）中众所周知的故障发生时的问题区分、迂回策略（workaround）的提出等。

本书主要着眼于发现 bug 之后的修改过程，这个狭义的调试。

本书结构

“第 1 章 热身准备” 概要地讲述调试是怎样的过程，并介绍本书的整体情况。

“第 2 章 调试前的必知必会” 介绍调试的基本知识，包括调试器（GDB）的用法、Intel 架构的基础知识、栈的基础知识、函数调用时的参数传递方法、汇编语言的学习方法等。

“第 3 章 内核调试的准备” 介绍 Linux 内核调试方法的基础，包括 Oops 信息的阅读方法、串口控制台的使用方法、通过网络获取内核信息的方法、SysRq 键、各种转储的获取方法、crash 命令的使用方法、用 IPMI 和 NMI watchdog 获取崩溃转储、内核特有的汇编语言等内核调试的基础知识。

“第 4 章 应用程序调试实践” 讲述用户应用程序的实践性的调试方法。通过实例介绍各种调试方法，包括栈溢出导致的 segmentation fault（SIGSEGV）、backtrace 无法正确显示、数组非法访问导致的栈破坏、灵活应用监测点检测内存非法访问、malloc()/free()中发生的故障、应用程序停止响应等。

“第 5 章 实践内核调试” 介绍内核故障的基本调试方法，包括 kernel panic（NULL 指针访问、链表破坏、竞态条件）、内核停止响应（死循环、自旋锁、信号量、实时进程）、运行缓慢、CPU 负载过高等各种异常情况的调试方法。

“第 6 章 高手们的调试技术” 汇集了广泛的内容，如调试时的各种工具、小技巧等。介绍的工具、技巧有 strace、objdump、Valgrind、kprobes、jprobes、KAHO、systemtap、proc 文件系统、oprofile、VMware Vprobe、错误注入、Xen 等。其他还有 OOM Killer 的运行和原理、通过 GOT/PLT 进行函数调用的原理和理解、initramfs、用 RT Watchdog 检测实时进程停止响应的方法、调查手头的 x86 机器是否支持 64 位的方法等。

“附录 Debug hacks 术语的基础知识” 介绍本书出现的术语。阅读各篇 hack 时如果遇到不理解的术语，可以参照该附录。

本书的使用方法

本书除了第 1 章之外，并没有假设阅读顺序。有一定基础的读者可以随意翻阅感兴趣的章节。如果想掌握基础知识，可以认真阅读第 1 章和第 2 章，并阅读参考文献等。而对于玩转内核的高手们，阅读本书所介绍的工具的使用方法等，也可能会感到甘之如饴。如果读者能告诉我们一些 hack 技巧，我们会感到十分幸运。

本书的体例

等宽字体（sample）

表示文件名、源代码、输出、命令等。

粗体等宽字体（sample）

表示用户输入的内容。

表示提示、建议等。

表示注意和警告等。

各 hack 左侧的温度计图标表示各 hack 的难易度。

初级

中级

高级

中文版书中订口处的“ ⟨ ”表示原书页码，便于读者与原日文版图书对照阅读，本书的索引所列页码为原日文版页码。

意见和疑问

我们尽最大努力对本书的内容进行验证和确认，但难免出现错误或容易导致误解的说法，也可能有印刷错误。阅读本书时如发现问题，请联系我们，以便再版时改正。

日本：

株式会社 O'Reilly Japan

东京都新宿区坂町 26 番地 27 Intelligent Plaza Building 1F 邮编 160-0002

电话：03-3356-5227

FAX：03-3356-5261

电子邮件 japan@oreilly.co.jp

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

有关本书的技术问题和意见，请发送至以下电子邮件：

info@mail.oreilly.com.cn

japan@oreilly.co.jp

本书的网页、示例代码[注1]、勘误表和追加信息位于：

http://www.oreilly.co.jp/books/9784873114040/

关于 O'Reilly 的其他信息请访问下面的网站。

http://www.oreilly.com.cn/（中文）

http://www.oreilly.co.jp/（日语）

http://www.oreilly.com/（英语）

注 1： 这些示例代码是笔者在执笔时使用的程序，并不保证能在任何环境下正常运行，而且可能会在没有预先通知的情况下做出修改。另外，本书不为示例代码提供任何支持。

目 录

第1章

热身准备

hack #1~#3

调试是什么

本书从基础到应用逐步说明调试过程。

调试究竟是什么过程呢？尽管是个程序员都会调试，但却没有任何详细记载调试过程的文档，真是匪夷所思。10 个程序员就有 10 种迥异的方法，然而这些方法却从未被明确记载过。

在本书中，我们根据自己的调试体会来介绍调试方法。我们不敢，也不会声称这些方法是最佳方法、除此之外别无他路。即便如此，把调试方法明确地写下来，也是有某种价值的。那就是，明确地写下来，经验较少的程序员就能直接学到别人无法传授的调试方法，而经验丰富的程序员也会再次思考自己的调试方法。特别是经验丰富的程序员们，希望大家阅读本书，找出和自己方法之间的差别。此外，明确这种差别，不断积累，我们程序员才能提高对编程和调试的认识。

编程的流程

程序员书写代码直到完成的过程中，会经历需求定义、设计、编码、测试和调试等阶段（如图 1-1 所示）。这里详细讨论编码、测试和调试阶段。

编写代码时，感到完成到某种程度时，就应该进行编译和构建，修改并消灭编译错误和构建错误。

2

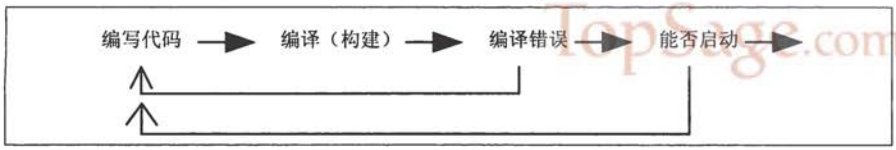


图 1-1 编程的流程

消灭编译错误后，虽然功能尚未完成，但也能运行，因此先跑起来看看行为是否与预期一致。这并不是正式的测试，而是先看看能否启动、能否运行。不能启动或崩溃等 bug，应该在进入测试阶段之前全部改掉。

调试和测试

非正式测试（确认能否启动等最基本的功能）之后，就是测试和调试工作。

这里先定义一下测试和调试。

测试就是确认程序是否与规格一致的工作。程序行为与期待行为（规格）不一致，就称为 bug。测试就是寻找程序中的 bug 的过程。

bug 不仅会在正式的测试阶段发现，也会在平常随意使用时发现，被自己以外的其他人发现，其发现方式多种多样。

不论哪种情况，调试总是开始于 bug 被确认之后。

一些人认为，即使 bug 没有被正式确认，某些工作也应当包括进来成为广义的“调试”，如觉得程序行为有些奇怪，或是针对用户提问进行确认等工作。但本书中调试仅仅指对通过某种方法发现的 bug 进行修正的过程。

寻找 bug 这件事

好的测试能发现大量 bug，而测试的初衷就是找出更多的 bug。

从测试的立场来看，发现 bug 的测试是成功的测试。这是因为，行为与规格（specification)不一致的程序是“失败”的程序，但作为测试而言，找到了 bug 就是“成功”（参考文献）。

3

也有观点说，并不是“bug 被发现了”这种被动形式，而应当使用“发现 bug”这种主动的说法，是通过测试（主动）发现了 bug。从这种观点中能感受到积极寻找 bug 的意志。

bug 并非自然发生，而是程序员自己写出来的，因此不应该被动地被发现，而是要主动地找出 bug 并改正。

作为程序员，谁都希望 bug 在被别人发现之前自己发现，哪怕只早一点点也好。

此外，有的方法论主张在编写程序代码之前先编写测试程序。这种方法称为 TDD（测试驱动开发）。它主张，不要去测试程序，而是每个程序都应事先准备好测试。可以说它是集测试、编程和调试于一身的方法论，不过本书不打算详细讲解。

bug 的分类

程序的行为可以分为以下几种。

① 执行期待的行为并结束。

② 不执行期待的行为并结束。

③ 不结束。

如果把编程简单地看做提供输入并获得输出的过程，那么测试就是针对几组输入检验其输出的过程。

①为与输入相应的输出等于期待的输出（规格），并且可以结束。这种情况并没有发现 bug，因此不需要调试。但是，仅仅是测试中没发现 bug，并不能保证 bug 不存在。

②是对于给定的输入，没有输出预期的值的情况。这时就发现了 bug。

③为死循环或死锁等情况，对于给定的输入，无法输出期待的值，程序也不结束。

②和③的情况下就需要调试。本书说明这些情况应该如何调试。

调试的流程

4

调试的流程请见图 1-2。

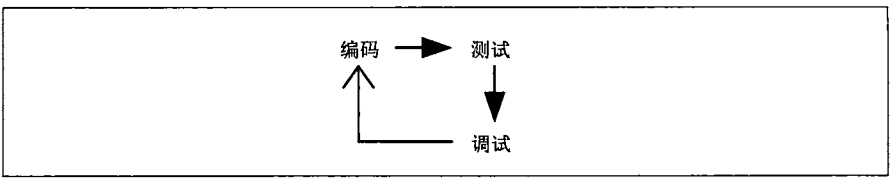


图 1-2 调试的流程

有的 bug 是通过前述的测试发现的，有的 bug 是被其他人发现的。调试的流程如下。

① bug 复现。
② 调试。
③ 确认执行过程（测试）。
④ 程序行为与预期相同，则停止；程序行为与预期不同，转到②。

本书将讲述各种各样的 bug 调试方法。

参考文献

《ソフトウェア・テストの技法第 2 版》（マイヤーズ等著，近代科学社出版，ISBN 978-4-7649-0329-6）。[注1]

——吉冈弘隆

Debug hacks 的地图

一开始很难判断 bug 的种类。

本 hack 将对 bug 进行分类，以判断使用本书的哪一条 hack 来解决问题。

图 1-3 和图 1-4 将故障分类为“异常停止”、“不停止”和“其他现象”，并分别说明原因区分方法和有关的 hack 编号。内核故障的区分方法参见表 1-1。

注 1：英文原书为《The Art of Software Testing, Second Edition》，Glenford J. Myers 等著，Wiley 出版，ISBN 978-0471469124。中文译本为《软件测试的艺术（原书第 2 版）》，机械工业出版社，ISBN 978-7111173199。——译者注

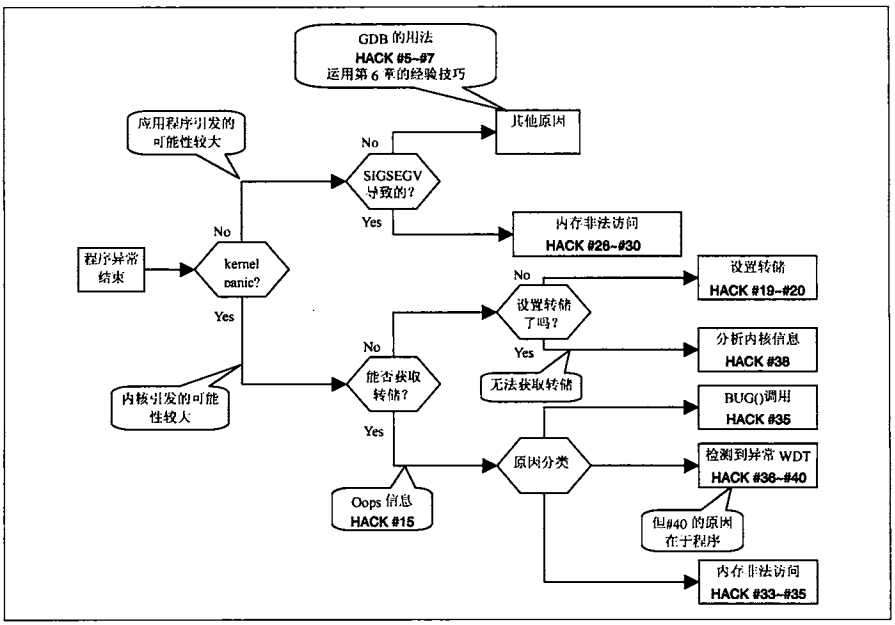


图 1-3 程序异常结束时的应对方法

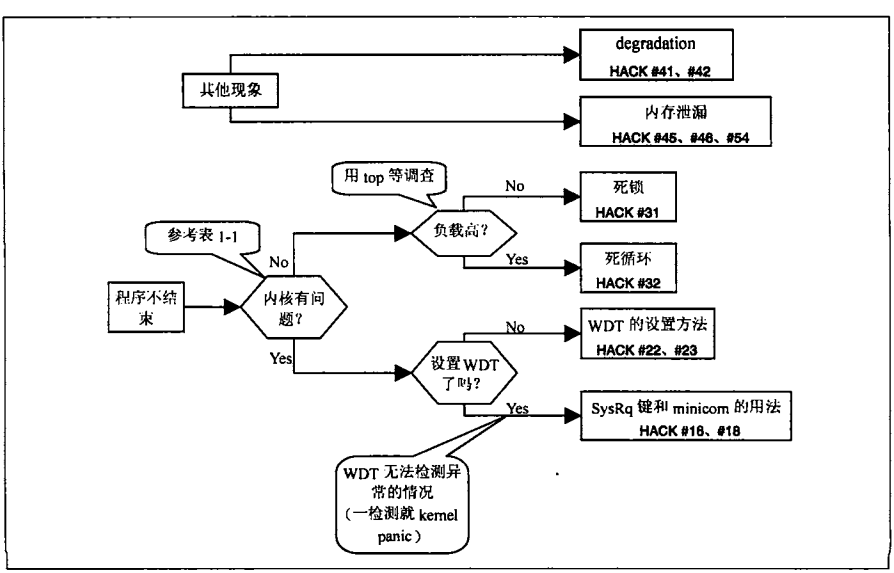


图 1-4 程序不结束时的应对方法

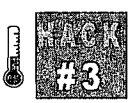
6 表 1-1 内核有问题的现象

区分方法	结果
ps	显示中途停止
	状态为 D
ping	不返回响应
键盘	键盘无法输入
kill -9	无法结束程序
strace	无法附加（attach）到进程（无响应）
gdb	无法附加（attach）到进程（无响应）
查看内核信息	softlockup 等有输出结果

总结

本 hack 说明了典型 bug 的区分方法和与本书的 hack 的对应关系。希望本 hack 能对阅读本书有所帮助。

——大和一洋、安部东洋、大岩尚宏



调试的心得

介绍调试前及调试时的心得体会。

发现 bug 之后，为确定调试方向，必须收集信息并掌握当前状况。

发现 bug 有几种形式。

① 通过测试发现。
② 在测试之外发现。

①是按照测试项目进行测试，“确认”的意思更多一些。能够复现的 bug 或是自己开发的程序都能较快地改正。

②是最麻烦的情况。②中有自己发现的情况，也有别人发现的情况。因此，在与测试环境不同的环境中发现的 bug、在多个功能的综合测试或压力测试中发现的 bug 也有很多。

不管怎么说，bug 总是要修改的，本 hack 针对难度较高的②总结了以下心得。

- 复现之前的心得。
- 复现之后的心得。
- 分析时的心得。
- 找不出问题原因时的心得。

1. 复现之前的心得

发现 bug 之后的重要工作是以 dump（转储文件）为提示复现 bug。

这里说明一下复现之前的准备工作和心得（复现用的测试程序（TP）的制作方法请参见**“HACK#34 kernel panic（链表破坏篇）”**。dump 的采集方法请参见**“HACK#19 使用 diskdump 获取内核崩溃转储”**和**“HACK#20 使用 kdump 获取内核崩溃转储”**）。

配置环境

发现 bug 之后就该进行调试工作，但在不同的硬件或软件版本上确认 bug 的话，其行为有可能不一样。

在 bug 发生的机器上复现是最好的，如果无法做到，应当尽可能让复现环境与 bug 的发生环境一致。硬件和软件都相同的环境是最理想的。首先，要使用同样的硬件，即使硬件不同，也要保证网卡和 CPU 数量等尽可能接近。软件方面，要保证操作系统、版本、分区状况和交换区大小等各个细节的一致。

根据笔者经验，下述环境一致的情况能够复现 bug。

- 挂载（Mount）、内核模块等参数、选项。
- 网络通信的对方的硬件（网卡）。
- 磁盘的硬件制造商。
- 配置文件的内容。

曾遇到过一个问题，在测试用的内部网络上无法复现的 bug，在其他环境中却能复现。其原因是：内部网络无法访问 NTP 服务器。
另一个例子是，/etc/sysconfig/network-script/ifcfg-eth*的内容完全一致，却无法再现。其原因是：/etc/sysconfig/network 的设置内容不足。

8 **听取**

试图复现他人发现的bug时，如果有疑问或需要更多信息，应当总结之后一起问。反复询问细小的问题会让人疲倦，所以应当先操作一遍，将问题、疑难事项总结之后再提问。此外，对方可能不告诉你重要信息，因为他觉得跟bug没关系，所以在听取时应当考虑到这一点。

曾有一次，我试图用别人写的测试程序复现bug，但怎么也无法复现。连续一周，每几分钟执行一次，都无法复现，但最后无意间几秒执行一次就复现了。我也曾问过测试程序的执行时机，但回答的人和实际编写测试程序并执行的人并不是同一人，因此没告诉我要几秒钟执行一次。此外，bug若是I/O缓存相关的问题，必须在第1次进行I/O处理时执行测试程序才能复现。

重新审视配置

在忽忙中解决bug时，经常会犯下简单的错误，如配置错误等。无法复现时，应当确认网线、配置文件内容正常，并进一步用命令的输出确认是否有问题。

2. 复现之后的心得

下面讲讲复现之后的心得。

确认现象

有时看似复现了bug，其实却是完全不同的现象。要确认bug现象是否真的复现了。

确认复现率和时间

bug是100%复现，还是仅在特定时机才会复现，使用的调试方法和时间是完全不同的。

此外，有些bug执行30分钟不能复现，而执行10小时才能复现。因此要弄清楚复现的时间和概率。

3. 分析时的心得

下面讲述bug分析的切入点，以及寻找问题原因时的心得。

9 **目视确认现象**

比如进行某种操作后就重启的例子。应当确认，是执行特定操作后出现panic就重启的，还是几十秒以后由watchdog重启的？重启之前键盘能否操作、ping是否畅通等，确认这些对于调试很有帮助。如果每次bug复现所用的时间都一样，就应该

确认这段时间的长短，这个值对于调试也经常有所帮助。

曾遇到过在 shell 中执行 sleep 1，其睡眠时间明显超过 1 秒的现象。可能是由于内核计时器本身的 bug，因此用 time 命令测量的结果没有参考价值。实际确认一下现象，发现实际睡眠时间约为 5 秒（多 4 秒），而且每次都睡眠相同的秒数。进一步，如果 sleep 10 的话，会睡眠 10 倍，即 50 秒，还是睡眠 10+4 秒，即 14 秒呢？确认结果是 14 秒，这就是解决问题的切入点。

再举个不同的例子。我们经常接到“控制台消息无法显示到画面上”这种问题反馈，这时请执行 echo 7 > /proc/sys/kernel/printk。比这个文件设置的控制台级别还要低的消息都不会显示，这一点应当注意。

尽量缩小范围

如果执行 3 个测试程序会发生 bug，那么应该试着逐个执行。如果范围能缩小到一个程序，调试就会更加简单；如果必须同时执行 3 个程序 bug 才会发生，那么可以从其他方向进行调试。通过选项缩小范围的例子请见 **“HACK#32 应用程序停止响应（死循环篇）”**。此外，还要调整相关参数。限制范围、调整参数后，应该能缩短问题复现的时间。通过调整参数提高复现概率的例子请参见 **“HACK#35 kernel panic”**。

如果是文件系统日志（Journal）方面的 bug，可以通过 mount -o commit=1 将提交（commit）时间从默认的 5 秒缩短到 1 秒，bug 可能更容易复现。如果是 e1000 驱动程序的缓冲区方面的 bug，可以用 ethtool -G ethX rx 64 tx 64 缩小收发缓冲区。诸如此类问题，根据 bug 情况可以用各种方法改变参数。

根据内核配置、内核启动参数划分问题

10

如果是内核的 bug，可以通过修改启动参数、内核配置的方法寻找原因。在启动参数中加上 nosmp，可以确认是否为 SMP 环境下发生的 bug。如果是 e1000 驱动程序相关 bug，禁用 NAPI 也许就能做出判断。

根据版本划分问题

开源代码随时都在修改、更新。有些代码的版本要比现行系统中的代码新很多。如果在高版本中不发生，那么只需调查代码差异即可。用版本判断问题原因的例子请见 **“HACK#32 应用程序停止响应（死循环篇）”**、**“HACK#36 内核停止响应（死循环篇）”**。相反，版本升级后产生 bug，要确认低版本是否发生。

通过其他途径确认

不要受限于一种信息，而应当结合多种信息确认。网络问题的话，不应仅查看ifconfig命令，还要综合考虑ip命令、route命令及/proc/net的信息等。有时命令的显示也有bug。

根据事实做出判断

根据现象很容易想到“很可能是某个原因”，但不要仅凭表象做出没有根据的判断，而应当认真地确认，否则，就会与真正的原因擦肩而过。解决问题时经常会有这种感慨：“没想到原因在这儿啊！”

4. 问题原因不明时的心得

有时会调查不出问题原因而束手无策，有时会有说不清楚的现象。下面是这些情况下的心得。

怀疑硬件问题

调试软件时会遇到无法解释的现象，或是同样条件下每次行为不一致的情况，此时可以怀疑是硬件问题。虽说一开始就怀疑硬件的话，本来能改好的问题也改不好了，但产生不自然的行为时，很有可能是硬件故障。虽说是硬件故障，但像没开电源这种很容易发现的问题，也可能会被当做软件故障。

11

以前曾遇到过分区突然看不见的现象。由于一直在更新软件，因此最初以为是软件问题，但有时候又能看见分区。想起以前在旁边捡到个电子元件，完全不知道是哪儿掉下来的，于是怀着侥幸心理看了一眼磁盘内部，发现是磁盘里的电子元件掉了。最后换了块硬盘就完全没问题了。

不仅是笔者，其他人也有同样的经验。比如看上去像个bug，于是反复调试软件，最后发现是硬件连接线断了一根。

EDAC（Error Detection And Correction）（bluesmoke）

Linux的校验错误检测和通知功能为EDAC（合并到标准内核[注2]之前称为bluesmoke）。EDAC能检测出内存的ECC校验错误和PCI总线的校验错误。

内存发生ECC校验错误时，硬件会在MCH（Memory Controller Hub）寄存器

注2：标准内核（stock kernel），指由kernel.org提供的官方内核。而大多数发行版都会对内核做修改，标准内核就是相对于这些而言的。——译者注

上显示错误详情，并产生 NMI 中断。EDAC 利用该中断检测错误。EDAC 检查 MCH 寄存器，如果是能修正的 1 比特错误就产生警告，多于 2 比特的无法修复的错误，就故意让内核 panic。至于 PCI 总线的校验错误，EDAC 会轮询寄存器，如果产生错误就引发 panic。

EDAC 还能从 sysfs 文件系统中获取设置和错误的统计信息，还能获得产生错误的内存的 DIMM 编号。

详情请见内核代码的 Documentation/drivers/edac.txt 或 Documentation/edac.txt。

找找以前改正的同类错误

原因不明、无从下手时，可以找找以前是否有同类的 bug。在 git 或 Bugzilla 中用关键字搜索一下，也许能找到。

无法复现、原因不明时

有时信息不足导致无法复现 bug，进而无法分析，而时间又不允许继续分析下去了。此时应当在程序中加入输出调试信息的处理。下次发生同一现象时，就能根据这些信息复现 bug，也可以弄清楚其原因。

12

为 bug 发生做准备

bug 突然产生时，如果不清楚 bug 产生之前做了什么处理，bug 就很难解决。因此，应当为 bug 的发生做好充分准备。如果是自动化测试程序，就应当输出日志，这样即使在深夜执行，之后也能确认其行为。定期获取内存、网络、I/O、CPU 使用率等的日志，有助于调试的进行。可根据情况选择 `sar`、`top`、`free`、`/proc/meminfo`、`/proc/slabinfo` 等日志。

跟同事讨论

无论如何找不到原因时，可以试着跟同事讨论一下。为把现象讲清楚，必须自己先总结好。很多情况下用不着问，在总结的过程中就自己把问题解决了。而且，跟同事讨论时，可能会得到意想不到的线索，或得到一些提示。但是，同事也是很忙的，所以不要什么问题都问，仅限于那些实在无法解决的问题。

咨询社区

像 Linux 等开源软件都有社区。有不明白的问题，可以在开发者的邮件列表里提问。

世界上最了解源代码的当然是代码作者，直接询问他们，能得到最可信的信息。可以有效地使用这个方法。如果是提案或 bug 修正案，开发者们也会很高兴的。有关社区修正的话题在“**HACK#42 CPU 负载过高的故障**”中介绍。

总结

本 hack 介绍了调试前的准备工作，包括注意点、心得等，以便顺利进行调试。虽然是很基本的东西，但繁忙时也会疏忽。这些内容很重要，一定要认真学习。

参考文献

- EDAC Project
  http://bluesmoke.sourceforge.net/

——大岩尚宏

第2章

调试前的必知必会

hack #4~#14

本章介绍调试的基本知识，包括调试器（GDB）的使用方法、Intel 架构的基础、栈的基础知识、函数调用时的参数传递方式、汇编语言的学习方法等。

13

获取进程的内核转储

本 hack 介绍如何获取用户空间的进程的内核转储。

获取内核转储（core dump）的最大好处是，它能保存问题发生时的状态。只要有问题发生时程序的可执行文件和内核转储，就可以知道进程当时的状态。比如，在不清楚 bug 复现方法的情况下，或是 bug 极其罕见，又或是只在特定的机器上发生的情况等，只要获取内核转储，那么即使手头没有复现环境，也能够调试。

启用内核转储

大多数 Linux 发行版默认情况下关闭了内核转储功能。用 ulimit 命令可以查看当前的内核转储功能是否有效。

```
$ ulimit -c
0
```

-c 选项表示内核转储文件的大小限制。上例中限制为 0，表示内核转储无效。按照以下方式执行 ulimit 命令，即可开启内核转储。

```
$ ulimit -c unlimited
```

14

这个命令的意思是不限制内核转储文件的大小。设为无限制之后，发生问题时进程

的内存就可以全部转储到内核转储文件中。在调试大量消耗内存的进程时，可能会希望设置内核转储文件的上限，这时直接在参数中指定大小上限即可。例如，下面的命令设置上限为1GB。

```
$ ulimit -c 1073741824
```

开启该功能之后，下面试着执行一个程序，确认是否能够生成内核转储。这里所用的程序是个只访问0地址的程序。

```
$ ./a.out
Segmentation fault (core dumped)
```

当前目录下就会生成内核转储文件。

```
$ file core*
core.7561: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from './a.out'
```

要用GDB调试生成的内核转储文件，应当使用以下方式启动GDB。

```
$ gdb -c core.7561 ./a.out
GNU gdb Fedora (6.8-17.fc9)
...
Core was generated by './a.out'.
Program terminated with signal 11, Segmentation fault.
[New process 7577]
#0 0x000000000040048c in main () at segfault.c:6
6                       *a = 0x1;
```

segfault.c的第6行收到了11号信号。用GDB的list命令可以查看附近的源代码。

```
(gdb) l 5
1     #include <stdio.h>
2
3     int main(void)
4     {
5               int *a = NULL
6               *a = 0x1;
7               return 0;
8       }
```


由于指针a的值是NULL，可以看出在访问NULL指针的时候收到了信号。这是个最简单的例子，但调试复杂程序时，从内核转储入手也十分有效。程序越复杂，就

越难判断接收到信号时程序做了什么操作。此外，如果 bug 很难复现，那么单靠跟踪源代码会很难确定原因。这些情况下，内核转储能将问题发生时的状态原原本本地保存下来，有助于确定 bug 原因。

在专用目录中生成内核转储

使用大型文件系统时，会希望将内核转储放在固定的位置。默认情况下会在当前目录中生成，但可能难以弄清文件到底在哪里生成。此外，大量生成的内核转储文件可能会给系统的磁盘空间造成压力。这种情况下可以准备内核转储专用分区，并在该分区中设置内核转储文件，这样就方便很多。转储保存位置的完整路径可以通过 sysctl 变量 kernel.core_pattern 设置。假设在/etc/sysctl.conf 中这样设置。

```
# cat /etc/sysctl.conf
kernel.core_pattern = /var/core/%t-%e-%p-%c.core
kernel.core_uses_pid = 0
# sysctl -p
```

在该状态下执行刚才的 a.out 程序，就会在/var/core/下生成内核转储文件。

```
$ ls /var/core/
1223267175-a.out-2820-18446744073709551615.core
```

文件名的格式如下所示。

生成内核转储的时刻-进程名-PID-内核转储最大大小.core

kernel.core_pattern 中可以设置的格式符如表 2-1 所示。

表 2-1 kernel.core_pattern 中可以设置的格式符

格式符	说明
%%	%字符本身
%p	被转储进程的进程 ID（PID）
%u	被转储进程的真实用户 ID（real UID）
%g	被转储进程的真实组 ID（real GID）
%s	引发转储的信号编号
%t	转储时刻（从 1970 年 1 月 1 日 0:00 开始的秒数）
%h	主机名（同 uname(2)返回的 nodename）
%e	可执行文件名
%c	转储文件的大小上限（内核版本 2.6.24 以后可以使用）

16

上例中设置了 kernel.core_uses_pid=0，是因为我们改变了文件名中 PID 的位置。如果设置该值为 1，文件名末尾就会添加.PID。

使用用户模式辅助程序自动压缩内核转储文件

刚才的 kernel.core_pattern 中可以加入管道符，启动用户模式辅助程序。管道符（|）后面可以写程序名。例如下面的格式。

```
# echo "|/usr/local/sbin/core_helper" > /proc/sys/kernel/core_pattern
```

使用该方法可以自动压缩内核转储文件。

```
# cat /proc/sysctl.conf
kernel.core_pattern = |/usr/local/sbin/core_helper %t %e %p %c
kernel.core_uses_pid = 0
# sysctl -p
```

core_helper 的内容很简单。

```
$ cat /usr/local/sbin/core_helper
#!/bin/sh

exec gzip - > /var/core/$1-$2-$3-$4.core.gz
```

这样，发生内核转储时就会在/var/core/下生成压缩的内核转储文件。

17

```
$ ls /var/core/
1223269655-a.out-2834-18446744073709551615.core.gz
```

启用整个系统的内核转储功能

以前只需在/etc/initscript 中写一个执行 ulimit 的脚本即可，但最近的发行版多数无法使用/etc/initscript。下面介绍 Fedora9 中的步骤。首先编辑/etc/profile，开启登录到系统的所有用户的内核转储功能。下面这一行即为默认情况下禁用内核转储的命令。

```
ulimit -S -c 0 > /dev/null 2>&1
```

将其修改为 unlimited。

```
ulimit -S -c unlimited > /dev/null 2>&1
```

接下来要让通过 init 脚本启动的守护进程（daemon process）的内核转储功能有效。需要在/etc/sysconfig/init 文件中添加这样一行命令。

```
DAEMON_COREFILE_LIMIT='unlimited'
```

最后在/etc/sysctl.conf 中加入以下设置。

```
fs.suid_dumpable=1
```

这个设置使得被 SUID 的程序也能内核转储。出于安全性考虑，默认情况下该选项是无效的。启用整个系统的内核转储后，很难判断哪个程序在哪个目录中生成了转储文件，因此要通过“在专用目录中生成内核转储”（15 页）中介绍的方法让内核转储文件在固定的目录中生成。

最后重新启动系统，就可以启用整个系统的内核转储。

利用内核转储掩码排除共享内存

大规模应用程序会使用多个进程，更会使用几个 G 的庞大的共享内存。这种应用程序发生内核转储时，所有进程的共享内存全部转储的话，会对磁盘造成巨大压力，转储过程也会加重系统的负载，甚至由于转储时间过长导致服务停止时间过长等。因此，从版本 2.6.23 开始，内核实现了针对各个进程选择内核转储的内存区段的功能。此外，尽管 RHEL4.7、RHEL5.2 等的基础内核版本号低于上述版本，但也可以使用该功能。由于各个共享内存的进程中，共享内存的内容是相同的，所以没有必要所有进程都转储。因此，这种应用程序应当设置成只在某个进程中转储共享内存，其他进程无须转储。

18

设置方法很简单，可以通过/proc/<PID>/coredump_filter 进行。coredump_filter 使用比特掩码表示内存类型，如表 2-2 所示。

表 2-2 比特掩码对应的内存类型

比特掩码	内存类型
比特 0	匿名专用内存
比特 1	匿名共享内存
比特 2	file-backed 专用内存
比特 3	file-backed 共享内存
比特 4	ELF 文件映射（内核版本 2.6.24 以后的版本可以使用）

笔者的环境中默认值为 3，即转储所有的匿名内存区段。通过 coredump_filter 的内容可以查看设置情况。

```
# cat /proc/<PID>/coredump_filter
00000003
```

要跳过所有的共享内存区段，应将值改为 1。

```
# echo 1 > /proc/<PID>/coredump_filter
```

比特 4 用于转储共享库或可执行文件等 ELF 格式文件映射到的内存区段的第 1 页（x86 中为 4KB）。这样就可以通过内核转储文件查出，该区域映射了哪个 ELF 文件。

总结

本 hack 说明了获取内核转储的基本设置，以及转储专用目录、用户模式辅助程序等较为特别的使用方法。此外，还说明了在新版 Linux 内核中实现的内核转储掩码功能。

参考文献

- Manpage of CORE
  http://www.linux.or.jp/JM/html/LDP_man-pages/man5/core.5.html

——安部东洋

调试器（GDB）的基本使用方法（之一）

本 hack 说明 GDB 的基本使用方法，包括断点设置、继续运行等。

本 hack 说明 Linux 环境下的标准调试器 GDB。示例中使用的编译器为 GCC。GDB 的功能极其丰富，我们按照调试的流程进行说明。基本用法很简单。

流程如下所述。

(1) 带着调试选项编译、构建调试对象。
(2) 启动调试器（GDB）。

(2-1) 设置断点。
(2-2) 显示栈帧。
(2-3) 显示值。
(2-4) 继续执行。

准备

通过 gcc 的-g 选项生成调试信息。

```
$ gcc -Wall -O2 -g 源文件
```

如果使用 Makefile 构建，一般要给 CFLAGS 中指定-g 选项。

```
CFLAGS = -Wall -O2 -g
```

如果用 configure 脚本生成 Makefile，可以这样用。

```
$ ./configure CFLAGS="-Wall -O2 -g"
```

构建方法通常会写在 INSTALL、README 等文件中，参考即可。

编译器含有针对源代码中的各种各样的错误输出信息的功能，称为警告选项（Warning Option）。这些信息并不一定是错误，但却指出了容易引发 bug 的编码方式。

要用心写出整洁的代码，编译时不要出现任何警告甚至错误信息。编译错误是 bug 最大的根源。

-Werror 选项可以在警告发生时，将其当做错误来处理。

此外，发生编译错误时不会生成可执行文件。

给编译器（GCC）加上优化选项后，实际的执行顺序可能由于优化而与源代码顺序不同，因此利用调试器跟踪运行时，有时会执行到莫名其妙的地方，从而造成混乱。比如内联（inline）函数优化（去掉函数调用，而将函数代码在调用的位置展开），该函数名上就无法设置断点。这是因为内联优化从目标文件中去掉了该函数的入口点，符号表中也没有该函数的名称。

优化还会将局部变量保存到寄存器中，因此无法显示该局部变量的内容，必须直接查看寄存器的值。由于这些副作用，有些人建议在调试时去掉优化选项进行编译和构建，但我们不推荐这样做。

为什么呢?

使用 C、C++这些过程式编程语言编程，就是利用编译器这个工具把我们期待的

行为告诉计算机。尽管无须详细了解编译器优化选项的方方面面，但至少应当知道，优化选项可能会让执行顺序与源代码顺序不同。而优化选项就是在理解这一点的基础之上添加的，用来加快执行速度。所以没有必要特意去除。

如果只在调试时去掉优化选项，那就必须管理有优化选项和无优化选项的两种可执行文件。管理对象增加会导致管理成本上升，并不是好事。这会消耗大量成本，比如花费大量时间对没有优化选项的可执行文件进行调试，但实际上在优化后的可执行文件中 bug 并不存在；那么怎样管理由同一源代码编译、构建出的不同可执行文件呢？

而且，准备两个可执行文件的话，测试的工作量肯定会变成两倍，管理成本也会上升。

程序员是追求快乐的职业，为什么要把事情变得复杂呢？测试、调试的可执行文件应该只有一个。如果最终发布的代码是带有优化选项的代码，当然只有针对有优化选项的代码进行测试和调试才是正确的。

启动

$ gdb 可执行文件名

通过 emacs 启动的方法是 M-x gdb。

启动后显示下述信息，出现 gdb 提示符。

```
Current directory is /home/hyoshiok/work/coreutils/src/
GNU gdb 6.8-debian
Copyright(c)2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"…
(gdb)
```

设置断点

可以在函数名和行号等上设置断点。程序运行后，到达断点就会自动暂停运行。此时可以查看该时刻的变量值、显示栈帧、重新设置断点或重新运行等。断点命令（break）可以简写为 b。

格式：

break 断点

```
(gdb) b main
Breakpoint 1 at 0x8048e1f: file uname.c, line 184.
```

断点可以通过函数名、当前文件内的行号来设置，也可以先指定文件名再指定行号，还可以指定与暂停位置的偏移量，或者用地址来设置。

格式：

22

```
break 函数名
break 行号
break 文件名:行号
break 文件名:函数名
break +偏移量
break -偏移量
break *地址
```

[例]

```
(gdb) b iseq_compile
Breakpoint 2 at 0x81126f6: file compile.c, line 422.
(gdb) b compile.c:516
Breakpoint 3 at 0x8107421: file compile.c, line 516.
(gdb) b +3
Breakpoint 4 at 0x805bd58: file main.c, line 31.
(gdb) b *0x08116fd6
Breakpoint 5 at 0x8116fd6: file iseq.c, line 360.
```

上面的例子中分别对 iseq_compile()函数、compile.c 的 516 行、现在暂停位置往后 3 行、地址（0x08116fd6）设置断点。

如果不指定断点位置，就在下一行代码上设置断点。

```
(gdb) b
Breakpoint 6 at 0x805bd44: file main.c, line 28.
```

设置好的断点可以通过 info break 确认。

```
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y 0x081126f6   in iseq_compile at compile.c:422
3       breakpoint     keep y 0x08107421   in iseq_translate_threaded_code at compile. c:516
4       breakpoint     keep y 0x0805bd58   in main at main.c:31
5       breakpoint     keep y 0x08116fd6   in iseq_load at iseq.c:360
6       breakpoint     keep y 0x0805bd44   in main at main.c:28
```

运行

用 run 命令开始运行。不加参数只执行 run，就会执行到设置了断点的位置后暂停运行。可以简写为 r。

23

格式：

run 参数

```
(gdb) run -a
Starting program: /home/hyoshiok/work/coreutils/src/uname -a

Breakpoint 1, main (argc=2, argv=0xbf9cd714) at uname.c:184
```

经常用到的一个操作是在 main()上设置断点，然后执行到 main()函数。start 命令能达到同样的效果。

格式：

```
start
```

显示栈帧

backtrace 命令可以在遇到断点而暂停执行时显示栈帧。该命令简写为 bt。此外，backtrace 的别名还有 where 和 info stack（简写为 info s）。

格式：

```
backtrace
bt
```

显示所有栈帧。

```
backtrace N
bt N
```

只显示开头 N 个栈帧。

```
backtrace -N
bt -N
```

只显示最后 N 个栈帧。

```
backtrace full
bt full
backtrace full N
bt full N
backtrace full -N
bt full -N
```

24

不仅显示 backtrace，还要显示局部变量。N 与前述意思相同，表示开头 N 个（或最后 N 个）栈帧。

[例]

```
Breakpoint 2, vm_exec_core (th=0x0, initial=0) at vm_exec.c:86
(gdb) bt
#0 vm_exec_core (th=0x0, initial=0) at vm_exec.c:86
#1 0x08107421 in iseq_translate_threaded_code (iseq=0x977dcf0) at compile.c:510
#2 0x08107ac5 in iseq_setup (iseq=0x977dcf0, anchor=0xbfd5f01c) at compile.c:963
#3 0x081127c7 in iseq_compile (self=158469340, node=0x0) at compile.c:501
#4 0x081175f2 in rb_iseq_new_with_bopt_and_opt (node=0x0, name=158469360, filename=
158469360, parent=0, type=3, bopt=0, option=0x81a6ac0) at iseq.c:329
#5 0x081179e5 in rb_iseq_new (node=0x0, name=158469360, filename=158469360, parent=0,
type=3) at iseq.c:306
#6 0x08127148 in Init_VM () at vm.c:1864
#7 0x0806a595 in rb_call_inits () at inits.c:55
#8 0x0805e4d5 in ruby_init () at eval.c:65
#9 0x0805bd77 in main (argc=4, argv=0xbfd5f264) at main.c:34
(gdb)
```

[例：只显示前 3 个栈帧]

```
(gdb) bt 3
#0 vm_exec_core (th=0x0, initial=0) at vm_exec.c:86
```

```
#1 0x08107421 in iseq_translate_threaded_code (iseq=0x977dcf0) at compile.c:510
#2 0x08107ac5 in iseq_setup (iseq=0x977dcf0, anchor=0xbfd5f01c) at compile.c:963
(More stack frames follow...)
```

[例：从外向内显示3个栈帧，及其局部变量]

```
(gdb) bt full -3
#7 0x0806a595 in rb_call_inits () at inits.c:55
No locals.
#8 0x0805e4d5 in ruby_init () at eval.c:65
        _th = (rb_thread_t * const) 0x9711758
        _tag = {buf = {{__jmpbuf = {-1076497952, 135679824, 134593648, -1076498024,
516014545, -1092008770}, __mask_was_saved = 0, __saved_mask = {__val = {134537212,
3086358120, 3218469200, 3086296411, 3086358560, 3086027816, 1, 1, 0, 134561536, 188,
3218469192, 3085631476, 3218469208, 3085102259, 0, 3218469224, 3085102424, 3, 8388608,
8388608, 4294967295, 0, 134593648, 3218469272, 135462041, 8388608, 0, 5, 0, 8388608, 0}}}},
tag = 0, retval = 4294967295, prev = 0x0}
        state = 0
        initialized = 1
#9 0x0805bd77 in main (argc=4, argv=0xbfd5f264) at main.c:34
   variable_in_this_stack_frame = 135679849
```

25

显示栈帧之后，就可以看出程序在何处停止（即断点的位置），以及程序的调用路径。

显示变量

print命令可以显示变量。print可以简写为p。

格式：

print 变量

```
(gdb) p argv
$1 = (char **) 0xbf9cd714
(gdb) p *argv
$2 = 0xbf9cf6a5 "/home/hyoshiok/work/coreutils/src/uname"
(gdb) p argv[0]
$3 = 0xbf9cf6a5 "/home/hyoshiok/work/coreutils/src/uname"
(gdb) p argv[1]
$4 = 0xbf9cf6cd "-a"
(gdb)
```

该例显示 argv[]。可以看出，argv[0]中为可执行文件名（“/home/hyoshiok/work/coreutils/src/uname/”），argv[1]中为第 1 个选项（“-a”）。

显示寄存器

info registers 可以显示寄存器，简写为 info reg。

```
(gdb) info reg
eax            0x61  97
ecx            0x0   0
edx            0xb7f140f8       -1208925960
ebx            0xbf9cd714       -1080240364
esp            0xbf9cd4a0       0xbf9cd4a0
ebp            0xbf9cd678       0xbf9cd678
esi            0x0   0
edi            0x2   2
eip            0x8048ebd        0x8048ebd <main+173>
eflags         0x200213         [ CF AF IF ID ]
cs             0x73  115
ss             0x7b  123
ds             0x7b  123
es             0x7b  123
fs             0x0   0
gs             0x33  51
```

26

在寄存器名之前添加$，即可显示各个寄存器的内容。

```
(gdb) p $eax
$8 = 97
```

显示时可以使用以下格式，如表 2-3 所示。

格式：
p/格式 变量

表 2-3 显示寄存器可使用的格式

格式	说明
x	显示为十六进制数
d	显示为十进制数

（续表）

格式	说明
u	显示为无符号十进制数
o	显示为八进制数
t	显示为二进制数，t 的由来是 two
a	地址
c	显示为字符（ASCII）
f	浮点小数
s	显示为字符串
i	显示为机器语言（仅在显示内存的 x 命令中可用）

```
(gdb) p/c $eax
$7 = 97 'a'
```

十进制数的 97 为 ASCII 字符的'a'。

程序指针可以写为$pc，也可写为$eip，两者都可以显示。这是因为 Intel IA-32 架构中的程序指针名为 eip。

27

```
(gdb) p $pc
$9 = (void (*)()) 0x8048ebd <main+173>
(gdb) p $eip
$10 = (void (*)()) 0x8048ebd <main+173>
```

用 x 命令可以显示内存的内容。x 这个名字的由来是 eXamining。

格式：

x/格式 地址

```
(gdb) x $pc
0x8048ebd <main+173>:    0x0f6ef883
(gdb) x/i $pc
0x8048ebd <main+173>:    cmp    $0x6e,%eax
```

此处 x/i 意为显示汇编指令。

一般使用 x 命令时，格式为 x/NFU ADDR。此处 ADDR 为希望显示的地址，N 为重复次数，F 为前面讲过的格式（x、d、u、o、t、a、c、f、s、i），U 为表 2-4 中所示的单位。

表 2-4 U代表的单位

单位	说明
b	字节
h	半字（2 字节）
w	字（4 字节）（默认值）
g	双字（8 字节）

下面显示从 pc 所指地址开始的 10 条指令（i）。

```
(gdb) x/10i $pc
0x8048ebd <main+173>:    cmp   $0x6e,%eax
0x8048ec0 <main+176>:    je    0x8049048 <main+568>
0x8048ec6 <main+182>:    jg    0x8048f62 <main+338>
0x8048ecc <main+188>:    cmp   $0x61,%eax
0x8048ecf <main+191>:    nop
0x8048ed0 <main+192>:    je    0x8049055 <main+581>
0x8048ed6 <main+198>:    jg    0x8048f90 <main+384>
0x8048edc <main+204>:    cmp   $0xffffff7d,%eax
0x8048ee1 <main+209>:    je    0x8048fe8 <main+472>
0x8048ee7 <main+215>:    cmp   $0xffffff7e,%eax
```

28

也有反汇编的命令 disassemble，简写为 disas。

格式：

① disassemble。

② disassemble 程序计数器。

③ disassemble 开始地址 结束地址。

格式①反汇编当前整个函数，②为反汇编程序计数器所在函数的整个函数，③为反汇编从开始地址到结束地址之前的部分。

```
(gdb) disassem $pc $pc+50
Dump of assembler code from 0x8048ebd to 0x8048eef:
0x08048ebd <main+173>:cmp   $0x6e,%eax
0x08048ec0 <main+176>:je    0x8049048 <main+568>
0x08048ec6 <main+182>:jg    0x8048f62 <main+338>
0x08048ecc <main+188>:cmp   $0x61,%eax
0x08048ecf <main+191>:nop
0x08048ed0 <main+192>:je    0x8049055 <main+581>
```

```
0x08048ed6 <main+198>:jg    0x8048f90 <main+384>
0x08048edc <main+204>:cmp   $0xfffffff7d,%eax
0x08048ee1 <main+209>:je    0x8048fe8 <main+472>
0x08048ee7 <main+215>:cmp   $0xfffffff7e,%eax
0x08048eec <main+220>:lea   0x0(%esi,%eiz,1),%esi
End of assembler dump.
```

首先在任意位置暂停执行程序，即可像上例那样自由显示任意变量和地址。通过确认其值与预期是否相同，以确认是否存在bug。

单步执行

单步执行的意思是根据源代码一行一行地执行。

执行源代码中一行的命令为 next（简写为 n）。执行时如果遇到函数调用，可能想执行到函数内部，此时可以使用 step（简写为 p）命令。

例如，下例中停止在 print_element (name.sysname)中。step 命令可以进入 print_element ()函数内部执行，但 next 命令会在 print_element ()函数执行之后停在下一行（if(toprint & PRINT_NODENAME)）。

```
if (toprint & PRINT_KERNEL_NAME)
  print_element (name.sysname);
if (toprint & PRINT_NODENAME)
```

next 命令和 step 命令都是执行源代码中的一行。如果要逐条执行汇编指令，可以分别使用 nexti 和 stepi 命令。

nexti 命令不会进入函数内部执行，而 stepi 命令会。

继续运行

调试时，可以使用 continue（简写为 c）命令继续运行程序。程序会在遇到断点后再次暂停运行。如果没有遇到断点，就会一直运行到结束。

格式：

```
continue
continue 次数
```

指定次数可以忽略断点。例如，continue 5则5次遇到断点不停止，第6次遇到断点时才暂停执行。

被调试的程序通常为以下几种情况之一：

① 可以正常结束。
② 由于某种原因异常结束（发生内核转储、非法访问等）。
③ 无法结束（死循环等）。
④ 被挂起（停止响应、死锁等）。

除了正常结束之外，其他情况都需要从开头开始继续执行，以寻找问题的原因（调试）。

监视点

大型软件或大量使用指针的程序中，很难弄清变量在什么地方被改变。要想找到变量在何处被改变，可以使用watch命令（监视点，watchpoint）。

格式：

```
watch <表达式>
```

<表达式>发生变化时暂停运行。

此处<表达式>的意思是常量或变量等。

格式：

```
awatch <表达式>
```

<表达式>被访问、改变时暂停运行。

格式：

```
rwatch <表达式>
```

<表达式>被访问时暂停运行。

[例]

```
(gdb) awatch short_output
```

```
Hardware access (read/write) watchpoint 3: short_output
(gdb) c
Continuing.
Hardware access (read/write) watchpoint 3: short_output

Old value = false
New value = true
main (argc=1, argv=0xbfbf8924) at who.c:783
```

变量（short_output）的值发生变化时就会暂停运行。

要注意的是，设置监视点可能会降低运行速度。

删除断点和监视点

用 delete（简写为 d）命令删除断点和监视点。

31

格式：

```
delete <编号>
```

删除<编号>指示的断点或监视点。

[例]

```
(gdb) info b
Num     Type            Disp Enb Address      What
2       watchpoint      keep y               assumptions
3       acc watchpoint  keep y               short_output
          breakpoint already hit 1 time
(gdb) delete 2  ←删除 2 号断点
(gdb) info b
Num     Type            Disp Enb Address      What
3       acc watchpoint  keep y               short_output
          breakpoint already hit 1 time
```

其他断点

硬件断点（hbreak），适用于 ROM 空间等无法修改的内存区域中的程序。在有些架构中无法使用。

临时断点（tbreak）和临时硬件断点（thbreak），与断点（硬件断点）相同，都会在运行到该处时暂停，不同之处就是临时断点（临时硬件断点）会在此时被删除，所以在只需要停止一次时用起来很方便。

遗憾的是没有临时监视点。

改变变量的值

格式：

```
set variable <变量>=<表达式>
```

[例]

```
(gdb) p options
$7 = 1
(gdb) set variable options = 0
(gdb) print options
$8 = 0
```

32

上例将变量（options）的值改成了 0。

该功能可以在运行时随意修改变量的值，因此无须修改源代码就能确认各种值的情况。

生成内核转储文件

使用 generate-core-file 可将调试中的进程生成内核转储文件。

[例]

```
(gdb) generate-core-file
Saved corefile core.13163
```

有了内核转储文件和调试对象，以后就能查看生成转储文件时的运行历史（寄存器值、内存值等）。

此外，gcore 命令可以从命令行直接生成内核转储文件。

```
$ gcore 'pidof emacs'
```

该命令无须停止正在运行的程序以获得内核转储文件，当需要在其他机器上单独分析问题原因，或是分析客户现场发生的问题时十分有用。

总结

这里介绍了 Linux 环境中的标准调试器 GDB 的基本使用方法。内容包括调试器的使用准备、启动、断点设置、栈帧显示、值显示、继续运行等调试的基本过程。

参考文献

- GDB: The GNU Project Debugger
  http://sources.redhat.com/gdb/
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html

——吉冈弘隆

33

调试器（GDB）的基本使用方法（之二）

本 hack 介绍 GDB 的一些使用技巧。

刚才介绍了 Linux 环境下的标准调试器 GDB 的基本用法，接下来介绍一些小技巧。

attach 到进程

要调试守护进程（daemon process）等已经启动的进程，或是调试陷入死循环而无法返回控制台的进程时，可以使用 attach 命令。

格式：

```
attach pid
```

执行这一命令就可以 attach 到进程 ID 为 pid 的进程上。

查看进程 ID 可以使用 ps 命令。下面以 sleep 命令为例介绍调试方法。

```
$ ps aux|grep sleep
hyoshiok 17315  0.0 0.3  8984  5840 pts/4  Ss+ 13:33 0:00   /usr/bin/gdb --annotate=3 sleep
hyoshiok 17606 0.0  0.0  2792  620  pts/2  T+  13:41 0:00   ./sleep 100
```

```
hyoshiok 17895 0.0 0.0 3044 808  pts/1 S+  13:50 0:00   grep sleep
```

左起第 2 列的数字就是进程 ID（pid）。本例中的 pid 为 17606。

用下述方法可以在 GDB 中 attach 该进程。

```
(gdb) attach 17606
Attaching to program: /home/hyoshiok/work/coreutils-6.10/build-tree/ coreutils-6.10/
src/sleep, process 17606
'system-supplied DSO at 0xb801a000' has disappeared; keeping its symbols.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Loaded symbols for /lib/ld-linux.so.2
0xb803d430 in __kernel_vsyscall ()
(gdb) bt
#0 0xb803d430 in __kernel_vsyscall ()
#1 0x410bbdc0 in __nanosleep_nocancel () from /lib/tls/i686/cmov/libc.so.6
#2 0x0804a1ca in xnanosleep (seconds=100) at xnanosleep.c:112
#3 0x08048fd1 in main (argc=2, argv=Cannot access memory at address 0x4) at sleep.c:147
(gdb)
```

34

用 bt 命令显示 backtrace（栈帧），即可看出程序是通过怎样的调用途径陷入等待状态的。在 sleep 命令的例子中，从 backtrace 中可以看出，xnanosleep()调用的__nanosleep_nocancel ()函数调用了系统调用，实现了等待。

查看源代码，发现的确是 xnanosleep()先调用 nanosleep()。

```
xnanosleep (double seconds)
{
...
    errno = 0;
    if (nanosleep (&ts_sleep, NULL) == 0)
    break;
```

本例并不是有 bug，但通过确认 backtrace，可以在程序陷入死循环或由于其他原因导致的等待状况发生时进行调试。

等待的原因可能有调用输入输出或系统调用时发生阻塞，或等待获得锁等。

死循环，就是持续等待某条件变为真，其原因从自旋锁（spin lock）到单纯的逻辑错误（指定了绝不可能变成真的条件而导致的 bug）有很多种。

attach 之后就能使用普通的 gdb 命令，因此可以通过 print 命令显示变量，也可以设置断点。

此外，恢复程序运行一般可以使用 continue 命令（简写为 c）。

确认了行为之后，需要在 gdb 和进程分离时使用 detach 命令。这样调试中的进程就被从 gdb 的控制下释放出来。进程被 detach 后会继续运行。

进程信息可以用 info proc 命令显示。

```
(gdb) info proc
process 17606
cmdline = './sleep'
cwd = '/home/hyoshiok/work/coreutils-6.10/build-tree/coreutils-6.10/src'
exe = '/home/hyoshiok/work/coreutils-6.10/build-tree/coreutils-6.10/src/sleep'
```

35

条件断点

有一种断点仅在特定条件下中断。

格式：

```
break 断点 if 条件
```

这条命令将测试给定的条件，如果为真则暂停运行。

[例]

```
(gdb) b iseq_compile if node==0
Breakpoint 1 at 0x81126f6: file compile.c, line 422.
(gdb) run -e ' p 1'
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby -e ' p 1'
[Thread debugging using libthread_db enabled]
[New Thread 0xb80df6b0 (LWP 10586)]
[Switching to Thread 0xb80df6b0 (LWP 10586)]

Breakpoint 1, iseq_compile (self=166726860, node=0x0) at compile.c:422
```

格式：

```
condition 断点编号
```

```
condition 断点编号 条件
```

该命令可以给指定的断点添加或删除触发条件。第 1 个格式删除指定编号断点的触发条件，第 2 个格式给断点添加触发条件。

反复执行

格式：

```
ignore 断点编号 次数
```

在编号指定的断点、监视点（watchpoint）或捕获点（catchpoint）忽略指定的次数。

continue 命令与 ignore 命令一样，也可以指定次数，达到指定次数前，执行到断点时不暂停，二者的意义是相同的。

格式：

36

```
continue 次数
step 次数
stepi 次数
next 次数
nexti 次数
```

这些格式分别执行指定次数的相应命令。

格式：

```
finish
until
until 地址
```

finish 命令执行完当前函数后暂停，until 命令执行完当前函数等代码块后暂停，如果是循环，则在执行完循环后暂停，常用于跳出循环。

删除断点和禁用断点

用 clear 命令删除已定义的断点。如果需要保留定义，只想临时禁用断点的话，可以使用 disable 命令。将禁用的断点重新启用，则可使用 enable 命令。

格式：

```
clear
clear 函数名
clear 行号
clear 文件名:行号
clear 文件名:函数名
delete [breakpoints] 断点编号
```

格式：

```
disable [breakpoints]
disable [breakpoints] 断点编号
disable display 显示编号
disable mem 内存区域
```

37

如果不指定断点编号，则禁用所有断点，否则禁用指定的断点。第 3 种格式禁用 display 命令定义的自动显示，第 4 种格式禁用 mem 命令定义的内存区域。

可以省略 breakpoints 关键字。

格式：

```
enable [breakpoints]
enable [breakpoints] 断点编号
enable [breakpoints] once 断点编号
enable [breakpoints] delete 断点编号
enable display 显示编号
enable mem 内存区域
```

这些格式用于启用断点等。once 使指定的断点只启用一次，也就是说，程序运行到该断点并暂停后，该断点即被禁用。这与 delete 命令中的 once 不同，后者是在运行暂停后删除断点。

断点命令

断点命令（commands）可以定义在断点中断后自动执行的命令。

格式：

```
commands 断点编号
   命令
   ...
```

```
end
```

程序在指定的断点处暂停后，就会自动执行命令。下例设置在断点处暂停时执行 p *iseq（打印 iseq）。

38

```
(gdb) b 425
Breakpoint 2 at 0x811271a: file compile.c, line 425.
(gdb) command 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>p *iseq
>end
(gdb) c
Continuing.
[New Thread 0xb80eab90 (LWP 10836)]

Breakpoint 1, iseq_compile (self=166714140, node=0x0) at compile.c:422
(gdb) c
Continuing.

Breakpoint 2, iseq_compile (self=166714140, node=0x0) at compile.c:425
$4 = {type = 3, name = 166714160, filename = 166714160, iseq = 0x0, iseq_encoded = 0x0,
iseq_size = 0, mark_ary = 166714120, coverage = 0, insn_info_table = 0x0, insn_info_size = 0,
local_table = 0x0, local_table_size = 0, local_size = 0, argc = 0, arg_simple = 0,
arg_rest = -1, arg_block = -1, arg_opts = 0, arg_post_len = 0, arg_post_start = 0, arg_size=0,
arg_opt_table = 0x0, stack_max = 0, catch_table = 0x0, catch_table_size = 0, parent_iseq = 0x0,
local_iseq = 0x9f662d0, self = 166714140, orig = 0, cref_stack = 0x9efdacc, klass= 0,
defined_method_id = 0, compile_data = 0x9f59170}
(gdb) info b
Num Type       Disp Enb      Address    What
1   breakpoint  keep y        0x081126f6 in iseq_compile at compile.c:422
    stop only if node==0
    breakpoint already hit 2 times
2   breakpoint    keep y      0x0811271a in iseq_compile at compile.c:425
    breakpoint already hit 1 time
    p *iseq
```

另外，如果命令的第 1 行为 silent 命令，就不会显示在断点处暂停的信息。单独进行信息输出时这一点很有用。

与前面所述的条件断点组合使用，就可以在断点暂停时执行复杂的显示动作等。

```
break foo if x>0
 commands
39
  silent
  printf "x is %d\n", x
  cont
 end
```

常用命令及省略形式（别名）

命令名称在不与其他命令重复的前提下，可以按照下述方式简写，如表 2-5 所示。在命令行模式下使用时，按下 Tab 键，GDB 就会自动补全命令。

表 2-5 命令和简写形式

命令	简写形式	说明
常用命令		
backtrace	bt、where	显示 backtrace
break		设置断点
continue	c、cont	继续运行
delete	d	删除断点
finish		运行到函数结束
info breakpoints		显示断点信息
next	n	执行下一行
print	p	显示表达式
run	r	运行程序
step	s	一次执行一行，包括函数内部
x		显示内存内容
until	u	执行到指定行
其他命令		
directory	dir	插入目录
disable	dis	禁用断点
down	do	在当前调用的栈帧中选择要显示的栈帧
edit	e	编辑文件或函数
frame	f	选择要显示的栈帧
forward-search	fo	向前搜索
generate-core-file	gcore	生成内核转储
help	h	显示帮助一览
info	i	显示信息

（续表）

命令	简写形式	说明
其他命令		
list	l	显示函数或行
nexti	ni	执行下一行（以汇编代码为单位）
print-object	po	显示目标信息
sharedlibrary	share	加载共享库的符号
stepi	si	执行下一行

40

info 命令能显示调试对象的各种各样的信息。另一方面，show 命令能显示 GDB 内部的功能、变量和选项等。

总结

本 hack 介绍了 GDB 的使用技巧。

参考文献

- GDB: The GNU Project Debugger
  http://sources.redhat.com/gdb/
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html

——吉冈弘隆

调试器（GDB）的基本使用方法（之三）

本 hack 说明历史、初始化文件、命令定义等。

本 hack 继续介绍 Linux 环境下的标准调试器 GDB 的使用技巧。

值的历史

通过 print 命令显示过的值会记录在内部的值历史中。这些值可以在其他表达式中使用。

```
(gdb) p argc
$1 = (int *) 0xbf926e00
(gdb) p *argc
```

```
$2 = 1
```

最后的值可以用$访问，值历史的访问变量和说明如表 2-6 所示。

41

```
(gdb) p $
$3 = 1
```

用 show value 命令可以显示历史中的最后 10 个值。

```
(gdb) show value
$1 = (int *) 0xbf926e00
$2 = 1
$3 = 1
```

表 2-6 值历史的访问变量和说明

变量	说明
$	值历史的最后一个值
$n	值历史的第 *n* 个值
$$	值历史的倒数第 2 个值
$$n	值历史的倒数第 *n* 个值
$_	x 命令显示过的最后的地址
$__	x 命令显示过的最后的地址的值
$_exitcode	调试中的程序的返回代码
$bpnum	最后设置的断点编号

变量

可以随意定义变量。变量以$开头，由英文字母和数字组成。

```
(gdb) set $i=0
(gdb) p $i
$1 = 0
```

命令历史

可以将命令历史保存到文件中。保存命令历史后，就能在其他调试会话中重复利用这些命令（通过箭头键查找以前的命令），十分方便。默认命令历史文件位于./.gdb_history。

```
(gdb) show history
expansion: History expansion on command input is off.
```

```
filename:  The filename in which to record the command history is "/home/hyoshiok/
work/dbg/hyoshiok/chapter1/.gdb_history".
save: Saving of the history record on exit is on.
size: The size of the command history is 256.
```

42

格式：

```
set history expansion
show history expansion
```

可以使用 csh 风格的!字符。

格式：

```
set history filename 文件名
show history filename
```

可将命令历史保存到文件中。可以通过环境变量 GDBHISTFILE 改变默认文件名。

格式：

```
set history save
show history save
```

启用命令历史保存到文件和恢复的功能。

格式：

```
set history size 数字
show history size
```

设置保存到命令历史中的命令数量。默认值为 256。

初始化文件（.gdbinit）

Linux 环境下的初始化文件为.gdbinit。如果存在.gdbinit 文件，GDB 就会在启动之前将其作为命令文件运行。初始化文件和命令文件的运行顺序如下。

① $HOME/.gdbinit。
② 运行命令行选项。
③ ./.gdbinit。
④ 通过-x 选项给出的命令文件。

初始化文件的语法与命令文件的语法相同，都由 gdb 命令组成。

43

命令定义

利用 define 命令可以自定义命令，还可以使用 document 命令给自定义命令添加说明。用“help 命令名”可以查看定义的命令。

格式：

```
define 命令名
  命令
  ...
end
```

格式：

```
document 命令名
  说明
end
```

格式：

```
help 命令名
```

下例中定义了名为 li 的命令。它能显示当前$pc 所指的位置开始的 10 条指令。另外，用 document 命令给 li 命令定义了说明（list machine instruction）。用 help li 可以查看说明。

```
define li
  x/10i $pc
end
document li
  list machine instruction
end
```

[运行例]

```
(gdb) start
Breakpoint 1 at 0x805bd04: file main.c, line 28.
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby
[Thread debugging using libthread_db enabled]
```

44

```
[New Thread 0xb800e6b0 (LWP 8116)]
[Switching to Thread 0xb800e6b0 (LWP 8116)]
main (argc=1, argv=0xbfe23384) at main.c:28
28          setlocale(LC_CTYPE, "");
(gdb) li
0x805bd04 <main+20>:    movl  $0x8179826,0x4(%esp)
0x805bd0c <main+28>:    movl  $0x0,(%esp)
0x805bd13 <main+35>:    call  0x805b608 <setlocale@plt>
0x805bd18 <main+40>:    lea   0x4(%ebx),%eax
0x805bd1b <main+43>:    mov   %eax,0x4(%esp)
0x805bd1f <main+47>:    mov   %ebx,(%esp)
0x805bd22 <main+50>:    call  0x80d4390 <ruby_sysinit>
0x805bd27 <main+55>:    lea   -0xc(%ebp),%eax
0x805bd2a <main+58>:    mov   %eax,(%esp)
0x805bd2d <main+61>:    call  0x812f890 <ruby_init_stack>
(gdb) help li
list machine instruction
```

除了初始化文件，还可以把各种设置写在文件中，在运行调试器时读取这些文件。

格式：

```
source 文件名
```

有篇博客的文章就在 gdb 的命令文件中使用 libm.so 中定义的函数制作了简单的函数计算器。运行示例如下所示（引用文献①）。

```
(gdb) start
Breakpoint 1 at 0x805bd04: file main.c, line 28.
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby
[Thread debugging using libthread_db enabled]
[New Thread 0xb7f486b0 (LWP 9718)]
[Switching to Thread 0xb7f486b0 (LWP 9718)]
main (argc=1, argv=0xbfa5cfb4) at main.c:28
28          setlocale(LC_CTYPE, "");
(gdb) source gdbcalc
(gdb) p $log10(10000.0)
$1 = 4
(gdb) p $log2(1024.0)
$2 = 10
```

45

gdbcalc 文件如下所示。

```
#
# 源自以下博客
# http://www.keshi.org/blog/2006/03/gdb_hacks_gdbcalc.html
#
set $e = 2.7182818284590452354
set $pi = 3.14159265358979323846
set $fabs = (double (*)(double)) fabs
set $sqrt = (double (*)(double)) sqrt
set $cbrt = (double (*)(double)) cbrt
set $exp = (double (*)(double)) exp
set $exp2 = (double (*)(double)) exp2
set $exp10 = (double (*)(double)) exp10
set $log = (double (*)(double)) log
set $log2 = (double (*)(double)) log2
set $log10 = (double (*)(double)) log10
set $pow = (double (*)(double, double)) pow
set $sin = (double (*)(double)) sin
set $cos = (double (*)(double)) cos
set $tan = (double (*)(double)) tan
set $asin = (double (*)(double)) asin
set $acos = (double (*)(double)) acos
set $atan = (double (*)(double)) atan
set $atan2 = (double (*)(double, double)) atan
set $sinh = (double (*)(double)) sinh
set $cosh = (double (*)(double)) cosh
set $tanh = (double (*)(double)) tanh
set $asinh = (double (*)(double)) asinh
set $acosh = (double (*)(double)) acosh
set $atanh = (double (*)(double)) atanh
```

总结

本 hack 介绍了 GDB 的使用技巧。

引用文献

① gdb hacks - gdbcalc スクリプト（gdb hacks - gdbcalc 脚本）

ほげめも 深追いと佳境の日々

http://www.keshi.org/blog/2006/03/gdb_hacks_gdbcalc.html

参考文献

- GDB: The GNU Project Debugger
  http://sources.redhat.com/gdb/
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html

——吉冈弘隆

Intel 架构的基本知识

介绍 CPU 架构的基本知识。

作为调试的基本知识，这里简单介绍一下 CPU 架构。

字节序

Intel 系列 CPU 的位序和字节顺序如图 2-1 所示。

所谓 Endian，就是多字节数据在内存中的排列方式。
例如，0x12345678 这个数据像图 2-2 那样，低位数据排在内存低地址，就叫做 Little Endian，这是 Intel 架构采用的方法。相反，将高位数据放在低地址的方式叫做 Big Endian，为 SPARC、MIPS 架构采用。

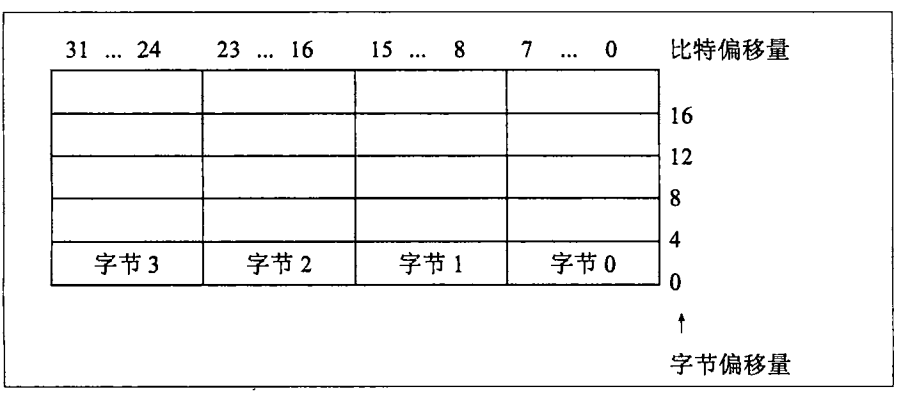


图 2-1 Intel 系列 CPU 的构造

47

0003	0002	0001	0000	
0x12	0x34	0x56	0x78	0

图 2-2 Little Endian 的例子

32 位环境中的寄存器

如图 2-3 所示，通用寄存器有 8 种，分别是 EAX、EBX、ECX、EDX、ESI、*EDI*、EBP、ESP，用于逻辑运算、数学运算、地址计算、内存指针等。

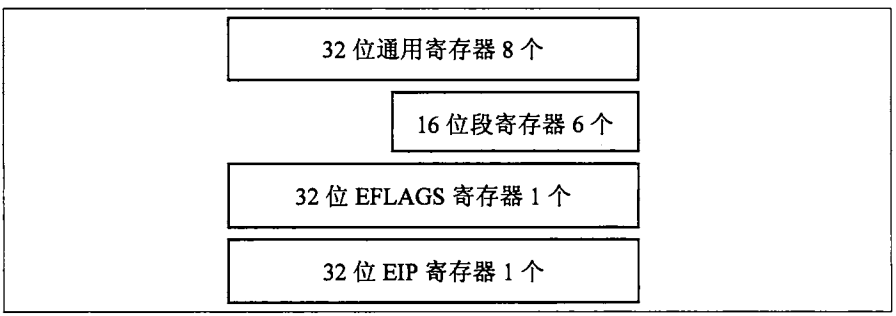


图 2-3 32 位环境中运行基本程序的寄存器

ESP 寄存器用于保存栈指针。

某些命令使用特定的寄存器。例如，字符串命令将 ECX、ESI 和 EDI 寄存器作为操作数使用。通用寄存器的主要用途请参见表 2-7。

表 2-7 主要寄存器的用途

寄存器	用途
EAX	操作数的运算、结果
EBX	指向 DS 段中数据的指针（主要段寄存器的用途见表 2-8）
ECX	字符串操作或循环的计数器
EDX	输入输出指针
ESI	指向 DS 寄存器所指示的段中某个数据的指针，或者是字符串操作中字符串的复制源（source）
EDI	指向 ES 寄存器所指示的段中某个数据的指针，或者是字符串操作中字符串的复制目的地（destination）
ESP	栈指针（SS 段）
EBP	指向栈上数据的指针（SS 段）

表 2-8 主要段寄存器的用途

寄存器	用途
CS	代码段
DS	数据段
SS	堆栈段
ES	数据段
FS	数据段
GS	数据段

程序代码放在代码段中，数据放在数据段中，程序所用的栈放在堆栈段中（表 2-8）。

但是，通用寄存器的用途并不限于上面所述，也可以用于一般用途，所以上表只能作为参考。寄存器的结构参见图 2-4。

EFLAGS 寄存器中包含状态标志（status flag）、控制标志（control flag）、系统标志（system flag）等（图 2-5）。

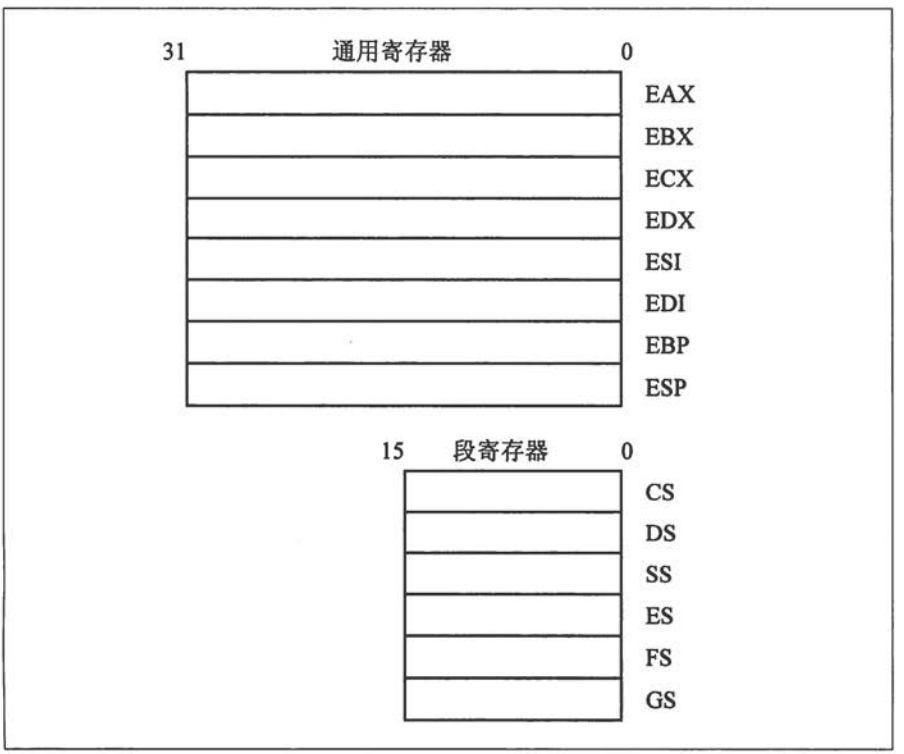


图 2-4 通用系统

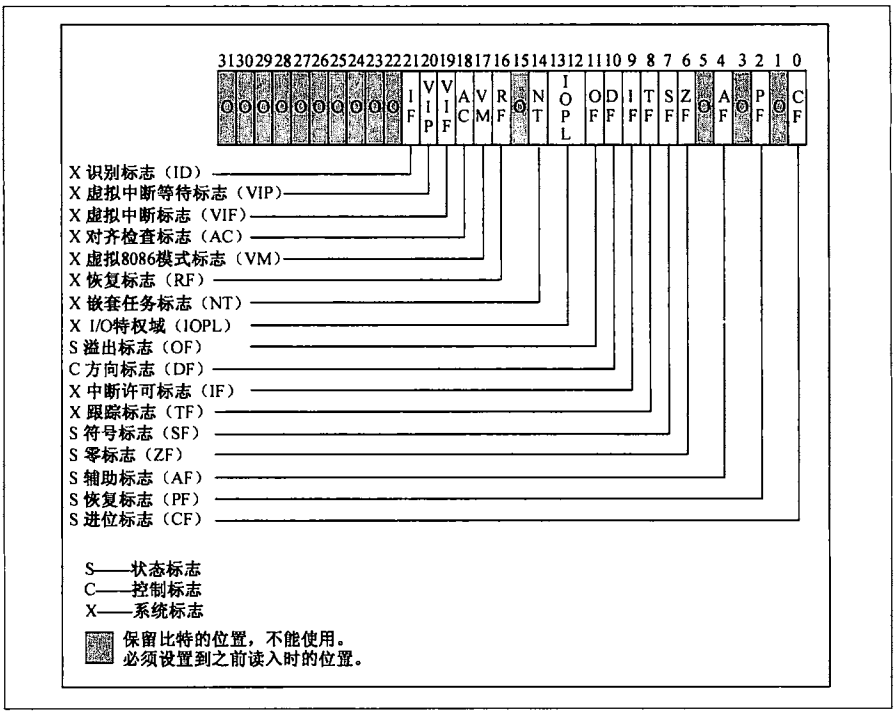


图 2-5 EFLAGS 寄存器

EIP（Instruction Pointer）寄存器是 32 位指令指针。

其他寄存器有控制寄存器（CR0～CR4）、GDTR、IDTR、TR、LDTR、调试寄存器（DR0/DR1/DR2/DR3/DR6/DR7）、内存类型范围寄存器 MTRR、MSR（Model Specific Register）寄存器、机器检查寄存器（Machine Check Register）、性能监视计数器（Performance Monitoring Counter）等（图 2-6、图 2-7）。

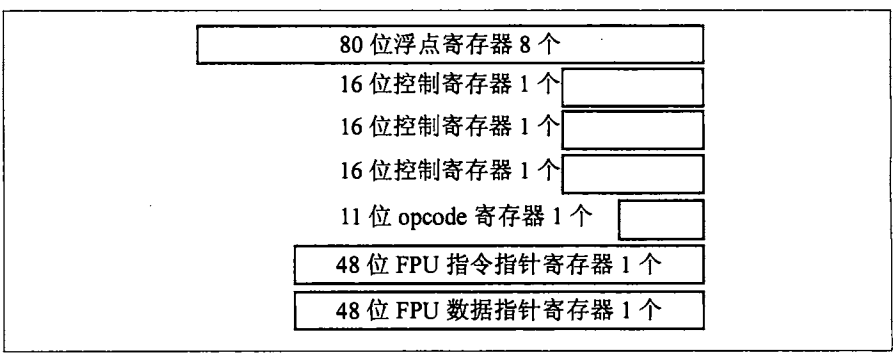
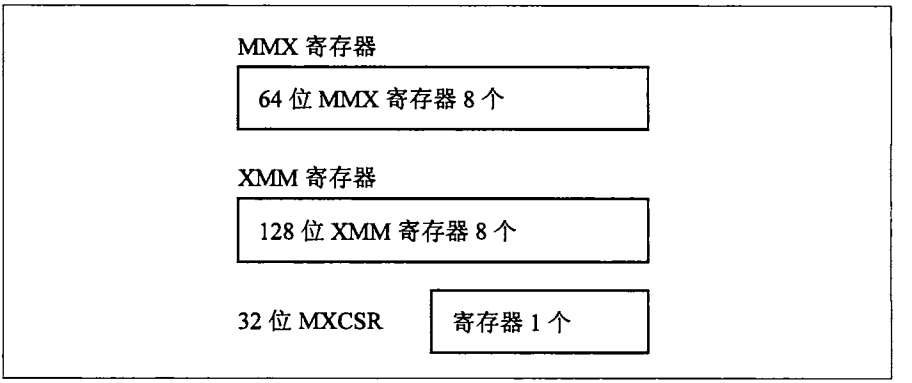


图 2-6 FPU 寄存器



50

图 2-7 MMX 寄存器和 XMM 寄存器

64 位环境中的寄存器

这些寄存器支持的地址空间为 2^{64} 字节。利用 CPUID 指令可以查看运行中的处理器支持的物理地址空间。

64 位模式下的通用寄存器在处理 32 位操作数时，可以通过 EAX/EBX/ECX/EDX/EDI/ESI/EBP/ESP/R8D～R15D 来使用。处理 64 位操作数时，可以使用 RAX/RBX/RCX/RDX/RDI/RSI/RBP/RSP/R8～R15。R8D～R15D/R8～R15 是 8 个新的通用寄存器。RIP 寄存器是 64 位指令指针（图 2-8）。

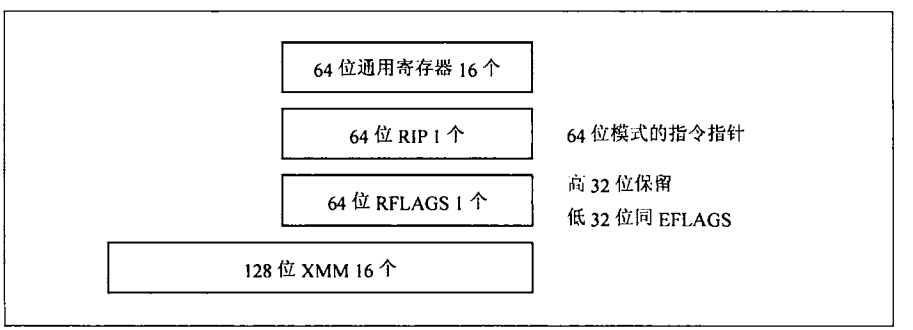


图 2-8 64 位环境寄存器

51

栈指针和控制寄存器都可以扩展到 64 位，并增加了 CR8 寄存器。而且，调试寄存器也可以扩展到 64 位，GDTR、IDTR 可以扩展到 10 字节，LDTR、TR 可以扩展到 64 位。

地址

CPU 可以通过内存总线访问到的地址称为物理地址。32 位模式下最大为 64GB（2^{36}）。而 64 位模式下的最大物理地址，Intel 当前的实现中为 2^{40} 字节，AMD 的实现中为 2^{48} 字节。

平坦模型（flat model）中（参见图 2-9），内存可以看做单一、平坦的连续地址空间，称为线性地址空间。Linux 采用这种内存模型。

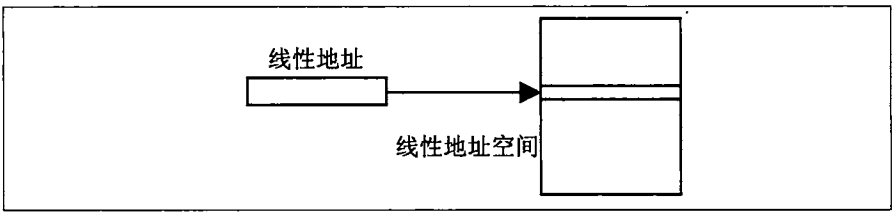


图 2-9 平坦模型

分段式内存模型（segment model）中，将内存看做被称为“段”（segment）的独立地址空间的集合。通过段选择器和偏移量组成的逻辑地址来访问段内地址。首先用段选择器识别出要访问的段，然后通过偏移量找到该段的地址空间中的内存（图 2-10）。32 位模式下最多能指定 16383 个段，各段的最大大小为 2^{32} 字节。

52

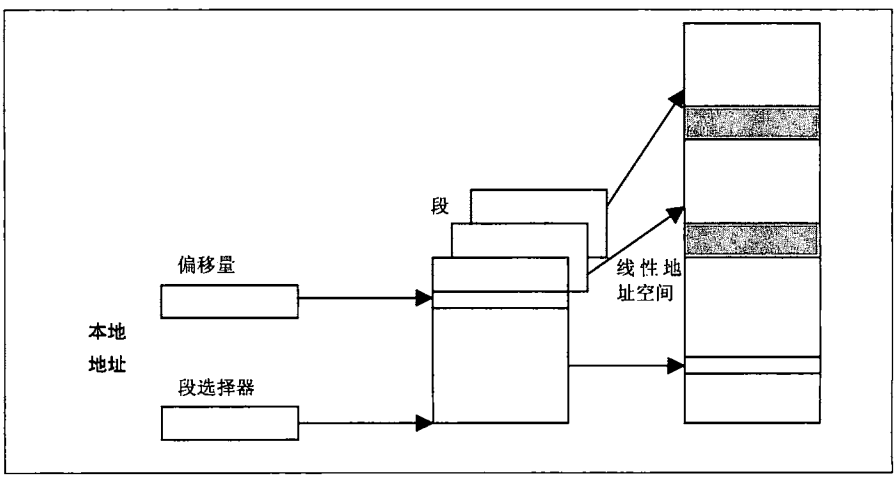


图 2-10 分段模型

64 位模式采用了平坦模型，因此可以使用 64 位线性地址。不能使用分段式内存模型。

数据类型

基本数据类型包括字节（8 比特）、字（16 比特）、双字（32 比特）、四字（64 比特）和双四字（128 比特）（图 2-11）。

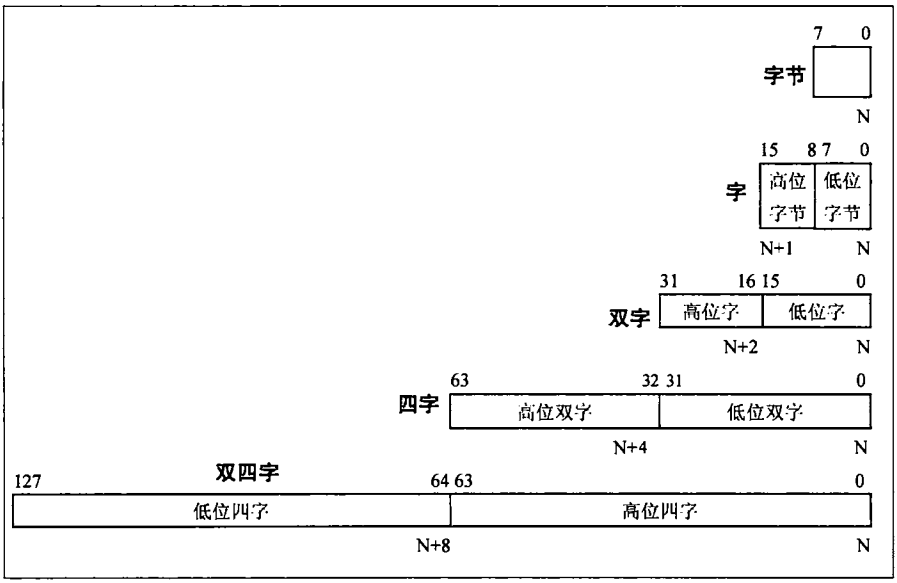


图 2-11 基本数据类型

整数数据类型

支持两种整数（无符号整数和符号整数）。无符号整数的范围为 0～最大整数，根据选择的操作数大小（字节、字、双字或四字）进行编码；符号整数能够表示正数和负数，采用补码表示。

浮点数据类型

支持单精度浮点数、双精度浮点数和扩展精度浮点数 3 种。这些数据类型的数据格式对应于 IEEE754 定义的格式。

53

单精度浮点数（32 位）的精度为 24 比特，双精度浮点数（64 位）的精度为 53 比特，扩展精度浮点数（80 位）的精度为 64 比特。

其他还有指针数据类型、bit-field（比特域）数据类型、字符串数据类型、压缩 SIMD 数据类型、BCD 和压缩 BCD 整数数据类型等。各种数据类型的详细说明请参考 Intel 的手册。

栈

有关栈的内容请参见[HACK#9]。

总结

本 hack 介绍了 Intel 架构的基本知识。

参考文献

- Intel 64 and IA-32 Architectures Software Developer's Manual (in five volumes)
  http://developer.intel.com/products/processor/manuals/index.htm

——吉冈弘隆

调试时必需的栈知识

本 hack 说明调试时不可或缺的栈的基本知识。

栈（stack）是程序存放数据的内存区域之一，其特征是 LIFO（Last In First Out，后进先出）式数据结构，即后放进的数据最先被取出。向栈中存储数据的操作称为 PUSH（压入），从栈中取出数据称为 POP（弹出）。在保存动态分配的自动变量时要使用栈。此外在函数调用时，栈还用于传递函数参数，以及用于保存返回地址和返回值。

本 hack 使用的示例程序如下所示。这个程序将命令行参数传递过来的数字作为终值，计算 0 到终值之间所有正数的总和。

```
$ cat sum.c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define MAX             (1UL << 20)

typedef unsigned long long u64;
typedef unsigned int u32;

u32 max_addend = MAX;
```

54

```
u64 sum_till_MAX(u32 n)
{
        u64 sum;
        n++;
        sum = n;

        if (n < max_addend)
                sum += sum_till_MAX(n);
        return sum;
}

int main(int argc, char** argv)
{
        u64 sum = 0;

        if ((argc == 2) && isdigit(*(argv[1])))
             max_addend = strtoul(argv[1], NULL, 0);
        if (max_addend > MAX || max_addend == 0) {
             fprintf(stderr, "Invalid number is specified\n");
             return 1;
        }

        sum = sum_till_MAX(0);
        printf("sum(0..%lu) = %llu\n", max_addend, sum);
        return 0;
}
```

下面求出从 0 到 10 的总和。

```
$ gcc -o sum -g sum.c
$ ./sum 10
sum(0..10) = 55
```

55

函数调用和栈的关系

下面讲述函数调用前后栈的变化情况。图 2-12 中的(a)为函数调用之前栈的状态，(b)为调用 sum_till_MAX()函数之后栈的状态，(c)为再次调用 sum_till_MAX()函数之后栈的状态。

栈上依次保存了传给函数的参数、调用者的返回地址、上层栈帧指针和函数内部使用的自动变量。此外，处理有些函数时还会用栈来临时保存寄存器。每个函数都独自拥有这些信息，称为栈帧（stack frame）。此时需要适当地设置表示栈帧起始地址的帧指针（FP）。此外，栈指针（SP）永远指向栈的顶端。

x86_64 中自动变量和工作空间有可能会超过栈指针。从栈指针指向的地址开始，再次扩展 128 字节，这部分空间称为 red zone（“危险区”），可以当做自动变量和工作空间使用。这是 AMD64 的 ABI 规格定义的内容。[注1]

现在来结合相应的汇编代码来看看它是如何运行的。

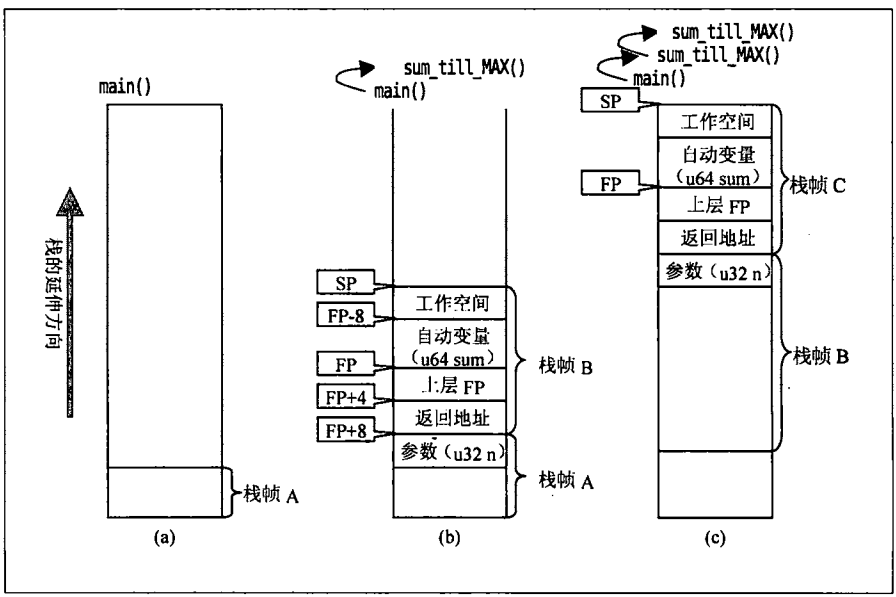


图 2-12 函数调用前后的栈状态

56

```
(gdb) disas main
...
0x08048544 <main+175>:  push  $0x0 ————————————————①
0x08048546 <main+177>:  call  0x8048458 <sum_till_MAX> ————②
0x0804854b <main+182>:  add   $0x8,%esp
...
```

注 1： 在 man gcc 中搜索-mno-red-zone 可以看到有关 red zone 的内容。

——译者注

调用函数时首先把传递给函数的参数压入栈中①，然后 sum_till_MAX()函数的 call 指令自动把返回地址（0x0804854b）压入栈中②。

下面来看看被调用的 sum_till_MAX()函数。

```
(gdb) disas sum_till_MAX
Dump of assembler code for function sum_till_MAX:
0x08048458 <sum_till_MAX+0>:    push  %ebp ──────────────────────────────③
0x08048459 <sum_till_MAX+1>:    mov   %esp,%ebp ─────────────────────────④
0x0804845b <sum_till_MAX+3>:    sub   $0x10,%esp ────────────────────────⑤
0x0804845e <sum_till_MAX+6>:    incl  0x8(%ebp) ─────────────────────────⑥
0x08048461 <sum_till_MAX+9>:    mov   0x8(%ebp),%eax
0x08048464 <sum_till_MAX+12>:   mov   $0x0,%edx
0x08048469 <sum_till_MAX+17>:   mov   %eax,-0x8(%ebp) ───────────────────⑦
0x0804846c <sum_till_MAX+20>:   mov   %edx,-0x4(%ebp)
0x0804846f <sum_till_MAX+23>:   mov   0x804978c,%eax
0x08048474 <sum_till_MAX+28>:   cmp   %eax,0x8(%ebp)
0x08048477 <sum_till_MAX+31>:   jae   0x804848d <sum_till_MAX+53>
0x08048479 <sum_till_MAX+33>:   sub   $0xc,%esp
0x0804847c <sum_till_MAX+36>:   pushl 0x8(%ebp)
0x0804847f <sum_till_MAX+39>:   call  0x8048458 <sum_till_MAX>
0x08048484 <sum_till_MAX+44>:   add   $0x10,%esp
0x08048487 <sum_till_MAX+47>:   add   %eax,-0x8(%ebp)
0x0804848a <sum_till_MAX+50>:   adc   %edx,-0x4(%ebp)
0x0804848d <sum_till_MAX+53>:   mov   -0x8(%ebp),%eax
0x08048490 <sum_till_MAX+56>:   mov   -0x4(%ebp),%edx
0x08048493 <sum_till_MAX+59>:   leave ──────────────────────────────────⑧
0x08048494 <sum_till_MAX+60>:   ret ────────────────────────────────────⑨
End of assembler dump.
```

首先在栈上保存上层帧的帧指针③，然后将新的栈帧赋给帧指针④。接下来在栈上分配用于保存自动变量的空间⑤。到此为止，图 2-12 所示的栈帧就准备好了。

从⑥开始为 sum_till_MAX()函数的处理过程。0x8(%ebp)指向帧指针＋8 字节的地址，根据图 2-12 可知，它指向传递给函数的参数（u32 n）。也就是说，n++;部分写成汇编代码即为如此。⑦为帧指针-8 字节的地址，表示自动变量（u64 sum）。但是，由于变量 sum 是 64 位的，⑦中只表示 sum 的低 32 位。%eax 中保存的是参数 n 的值，因此可知⑦相当于 sum=n;。⑧的 leave 指令为删除栈帧的指令，它执行与③和④完全相反的处理，以释放当前的栈。⑨为子程序返回指令，将栈中保存的返回地址 POP 到程序计数寄存器（program counter register）中，将控制权返回给调用者。

57

调试器的 backtrace

GDB 等调试器的 backtrace 功能是通过搜索栈中保存的信息来实现的。

下面在第 2 次调用 sum_till_MAX()时中断执行程序。正好与图 2-12 的(c)的状态一致。

```
(gdb) bt
#0 sum_till_MAX (n=2) at sum.c:18
#1 0x08048484 in sum_till_MAX (n=1) at sum.c:19
#2 0x0804854b in main (argc=1, argv=0xbfd89b34) at sum.c:34
```

下面来手动执行一下与 GDB 的 backtrace 相同的操作，将栈上保存的信息与图 2-12 的栈的示意图对照一下。首先获取当前的执行位置和 FP（帧指针）。当前执行位置可以通过程序计数器（PC）获得，在 x86 处理器上为 eip 寄存器。FP 是 ebp 寄存器。

```
(gdb) i r eip ebp
eip            0x804846f        0x804846f <sum_till_MAX+23>
ebp            0xbfd89a28       0xbfd89a28
```

接下来查看栈的内容。具体操作如下，从表示栈顶的 SP 开始显示适当的大小。

```
(gdb) x/40w $sp
```

实际的栈的显示结果如图 2-13 所示。为说明相应的部分在图中加入了注释，请对照阅读。

58

图 2-13 栈的显示结果

通过栈上保存的返回地址信息，可以获得与 GDB 的 backtrace 结果相同的调用跟踪

信息（图 2-14）。

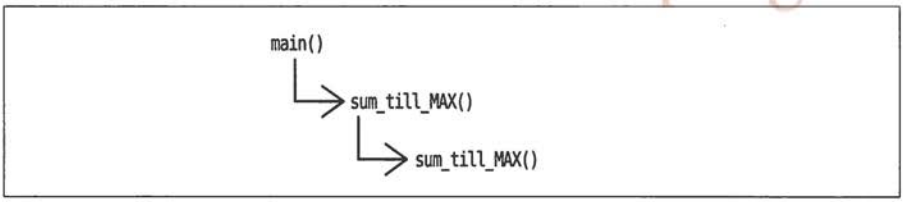


图 2-14 通过栈信息得到的调用跟踪情况

看到这里应该明白，对于调试器来说，栈上的数据是极其重要的信息。万一栈上的信息被破坏，就无法使用调试器跟踪调用过程。关于栈破坏，请参见 **“HACK#27 backtrace 无法正确显示”** 和 **“HACK#28 数组非法访问导致内存破坏”**。

编译时为 gcc 指定-fomit-frame-pointer 选项，即可生成不使用帧指针的二进制文件。在这种情况下，图 2-12 的栈示意图中的 FP 和上层 FP 信息不会被记录在栈上。但是即使如此，GDB 也能正确理解帧，这是因为 GDB 是根据记录在调试信息中的栈使用量来计算帧的位置的。

59

使用 GDB 操作栈帧

GDB 提供了操作栈帧的命令，这里介绍该命令的使用例。假设 GDB 中进程停止在以下状态。

```
(gdb) bt
#0 sum_till_MAX (n=4) at sum.c:18
#1 0x08048484 in sum_till_MAX (n=3) at sum.c:19
#2 0x08048484 in sum_till_MAX (n=2) at sum.c:19
#3 0x08048484 in sum_till_MAX (n=1) at sum.c:19
#4 0x0804854b in main (argc=1, argv=0xbfb92454) at sum.c:34
```

用 frame 命令查看现在选择的帧。

```
(gdb) frame
#0 sum_till_MAX (n=4) at sum.c:18
18                  if (n < max_addend)
```

现在选择的帧为#0，查看该帧内的自动变量 sum，其值为 4。

```
(gdb) p sum
$1 = 4
```

接下来选择上一层的#1 帧，用同样的方法确认自动变量 sum。

```
(gdb) frame 1
#1 0x08048484 in sum_till_MAX (n=3) at sum.c:19
19                              sum += sum_till_MAX(n);
(gdb) p sum
$1 = 3
```

帧#1 中的自动变量 sum 的值为 3。虽然查看的是帧#0 和帧#1 中的同名变量 sum，但 GDB 返回的是帧上选中的值。此外，选择帧还可以使用 up 和 down 命令。up 命令选择上一层的帧，down 命令选择下一层的帧。

```
(gdb) up
#2 0x08048484 in sum_till_MAX (n=2) at sum.c:19
19                              sum += sum_till_MAX(n);
(gdb) p sum
$2 = 2
```

60

用 info 命令的 frame 选项可以看到更详细的栈帧信息。可以用帧编号作为该命令的选项。

```
(gdb) i frame 1
Stack frame at 0xbfd8e218:
 eip = 0x8048484 in sum_till_MAX (sum.c:19); saved eip 0x8048484
 called by frame at 0xbfd8e240, caller of frame at 0xbfd8e1f0
 source language c.
 Arglist at 0xbfd8e210, args: n=3
 Locals at 0xbfd8e210, Previous frame's sp is 0xbfd8e218
 Saved registers:
  ebp at 0xbfd8e210, eip at 0xbfd8e214
```

栈大小的限制

实际上，本 hack 使用的示例程序如果不带参数执行，会引发 segmentation fault（段错误）。执行一下试试看。

```
$ ./sum
Segmentation fault
```

此处发生了栈溢出（stack overflow）。通过 GDB 执行示例程序，看看执行到什么时

发生了 segmentation fault。查看程序计数器（PC）即可看到程序执行位置。

```
$ gdb ./sum
...
(gdb) r
Starting program: /home/toyo/work/test/sum

Program received signal SIGSEGV, Segmentation fault.
0x0804847c in sum_till_MAX (n=209442) at sum.c:19
19                                      sum += sum_till_MAX(n);
(gdb) x/i $pc
0x804847c <sum_till_MAX+36>: pushl 0x8(%ebp)
```

61

这正是将 sum_till_MAX()的参数 n PUSH 到栈顶端的命令。现在查看一下栈指针（SP）的位置。

```
(gdb) p $sp
$1 = (void *) 0xbf06dffc
```

下面查看该进程的内存映射（memory map）。要查看 GDB attach 了的进程的内存映像，可以执行以下命令。执行该命令后，GDB 就会显示与被调试的进程相对应的 /proc/<PID>/maps 的信息。

```
(gdb) i proc mapping
process 11545
cmdline = '/home/toyo/work/test/sum'
cwd = '/home/toyo/work/test'
exe = '/home/toyo/work/test/sum'
Mapped address spaces:
        Start Addr   End Addr    Size        Offset      objfile
        0x8048000  0x8049000     0x1000      0           /home/toyo/work/test/sum
        0x8049000  0x804a000     0x1000      0           /home/toyo/work/test/sum
        0xb7e56000 0xb7e57000    0x1000      0xb7e56000
        0xb7e57000 0xb7f9a000    0x143000    0           /lib/libc-2.7.so
        0xb7f9a000 0xb7f9b000    0x1000      0x143000    /lib/libc-2.7.so
        0xb7f9b000 0xb7f9d000    0x2000      0x144000    /lib/libc-2.7.so
        0xb7f9d000 0xb7fa0000    0x3000      0xb7f9d000
        0xb7fae000 0xb7fb0000    0x2000      0xb7fae000
        0xb7fb0000 0xb7fb1000    0x1000      0xb7fb0000  [vdso]
        0xb7fb1000 0xb7fcd000    0x1c000     0           /lib/ld-2.7.so
```

```
0xb7fcd000 0xb7fcf000    0x2000      0x1b000      /lib/ld-2.7.so
0xbf06e000 0xbf86e000    0x800000    0xbf800000   [stack]
```

请注意最后一行的[stack]。它表示栈空间，栈空间的顶端是0xbf06e000。然而，刚才看到的栈指针的值却是0xbf06dffc，超出了栈的范围。访问地址超出了栈的范围，也就是说，发生了栈溢出。

62

使用该命令时，GDB会打开/proc/<PID>/maps，因此在分析core dump（内核转储）时无法使用。分析core dump时可以利用以下命令获得相同信息。

```
(gdb) info files
```

或者

```
(gdb) info target
```

该示例程序默认情况下会递归调用sum_till_MAX()一百万次以上。如前所述，每次函数调用都会生成栈帧，随之消耗栈空间。这里由于栈空间的消耗量超出了进程的许可范围，因而发生了栈溢出。

笔者的环境中，进程允许的栈大小为8MB。

```
$ ulimit -s
8192
```

将该值增大到10倍，再次执行示例程序，就不会发生segmentation fault而正常结束了。

```
$ ulimit -Ss 81920
$ ./sum
sum(0..1048576) = 549756338176
```

它计算的是0到1048576的正数之和，与以下计算等价。

```
(1 + 1048576) * (1048576 / 2)
= 1048577 * 524288
= 549756338176
```

可以证明，计算结果是正确的。

总结

本hack说明了栈的基本原理，以及调试器的backtrace功能是如何利用栈信息的。此外还介绍了GDB中可以使用的操作栈帧的命令，并举例说明了栈溢出。本hack

中提到，每个进程都有允许的栈大小，但实际上每个线程的栈大小也有限制。多线 63
程编程时，各个线程使用的栈的总和不能超过进程许可的栈大小，同时还要注意各个线程的栈大小限制。设计应用程序时一定要注意栈的使用量。

参考文献

- AMD64 Application Binary Interface
  http://www.x86-64.org/documentation/abi.pdf

——安部东洋

函数调用时的参数传递方法（x86_64 篇）

讲述在 x86_64 架构上参数是如何传递给被调用的函数的。

函数参数与调试

程序异常结束、与预期行为不一致，这是十分常见的故障。有错误信息的话，只需进行字符串查找就能确定显示该信息的源代码位置，相对比较容易。但是，故障的真正原因有可能在显示错误信息之前很远的地方。例如，某个函数计算出错误的值，以该值为参数调用其他函数的情况即如此。这种情况下，找出程序出错的位置才是解决故障的线索。然而实际上，找不到出错位置的情况十分普遍。此时可以检查可能与故障有关的函数的参数，以缩小有问题的函数的范围。本 hack 以下面的程序为例，说明使用 GDB 查看函数参数的方法。

参数传递方法根据架构、语言、编译器的不同而有所不同。本 hack 介绍的是 x86_64 架构上使用 C 语言的情况。后面的[HACK#11]介绍的是 i386 架构上使用 C 语言的情况，[HACK#12]介绍的是使用 C++的情况。这几个 hack 所用的编译器都是 GCC（G++）。

```
#include <stdio.h>
#include <stdlib.h>

int v1 = 1;
float v2 = 0.01;

void func(int a, long b, short c, char d, long long e, float f, double g, int *h, float
```
64

```
*i, char*j)
{
   printf("a: %d, b: %ld, c: %d, d: %c, e: %lld\n"
        "f: %.3e, g: %.3e\nh: %p, i: %p, j: %p\n", a, b, c, d, e, f, g, h, i, j);
}

int main(void)
{
  func(100, 35000L, 5, 'A', 123456789LL, 3.14, 2.99792458e8, &v1, &v2, "string");

  return EXIT_SUCCESS;
}
```

在笔者的环境下，该程序的执行结果如下所示。

```
a: 100, b: 35000, c: 5, d: A, e: 123456789
f: 3.140e+00, g: 2.998e+08
h: 0x600990, i: 0x600994, j: 0x4006a3
```

通过 GDB 确认

最简单的确认方法就是使用 GDB。带着-g 选项构建上述示例程序，用 GDB 在被调用的函数 func()开头设置断点，即可像下面这样显示参数。

```
(gdb) b func
Breakpoint 1 at 0x4004a0: file func_call.c, line 10.
(gdb) run
...
Breakpoint 1, func (a=100, b=35000, c=5, d=65 'A', e=123456789, f=3.1400001, g=299792458,
h=0x600990, i=0x600994, j=0x4006a3 "string") at func_call.c:10
10          printf("a: %d, b: %ld, c: %d, d: %c, e: %lld\n"
```

如果在构建时不指定-g 选项，即无法使用调试信息的情况下，就会像下面这样只显示暂停的地址，而不会显示参数的值。这种情况下获取参数的方法如下所示。

```
(gdb) b func
Breakpoint 1 at 0x40047c
(gdb) run
...
Breakpoint 1, 0x000000000040047c in func ()
```

x86_64 下的调用

在 x86_64 中，整型和指针型的参数会从左至右依次保存到 rdi、rsi、rdx、rcx、r8、r9 中，浮点型参数会保存到 xmm0、xmm1……中。多于这些寄存器的参数会被保存到栈上。因此，利用 GDB 在希望确认的函数开头中断之后，查看寄存器或栈即可获得参数内容。

此外，刚才那个例子设置断点时使用了函数名 func，但以后应当在函数名之前加上*（星号）。为什么呢？如果不加*，断点就不会设置到汇编语言层次的函数开头，而是设置到地址偏后一点的源代码级别的开头。大多数情况下，函数开头会进行以下的栈操作。由于参数也可能保存到栈上，如果在 break 命令中不加*直接使用函数名，就无法用于参数确认。

```
(gdb) disas func
Dump of assembler code for function func:
0x0000000000400478 <func+0>:    push   %rbp
0x0000000000400479 <func+1>:    mov    %rsp,%rbp
0x000000000040047c <func+4>:    sub    $0x50,%rsp
...
```

赶快确认一下参数吧。

```
(gdb) b *func
Breakpoint 1 at 0x400478
(gdb) run
...
Breakpoint 2, 0x0000000000400478 in func ()
```

在该状态下确认寄存器的内容。可以看出，开头 5 个参数 a、b、c、d 和 e 分别保存到了 rdi、rsi、rdx、rcx 和 r8 中。

```
(gdb) i r
rax            0x7fff93d32328   140735673475880
rbx            0x37d8019bc0     239847185344
rcx            0x41     65      ————————————  参数d
rdx            0x5      5         ————————————  参数c
rsi            0x88b8 35000      ————————————  参数b
rdi            0x64     100        ————————————  参数a
rbp            0x7fff93d32330   0x7fff93d32330
rsp            0x7fff93d322f8   0x7fff93d322f8
```

```
r8             0x75bcd15       123456789 ————————————参数e
r9             0x7fff93d3232c  140735673475884————————— 参数h
r10            0x0     0
r11            0x37d821d7b0    239849297840
r12            0x0     0
r13            0x7fff93d32410  140735673476112
r14            0x0     0
r15            0x0     0
rip            0x400478 0x400478 <func>
eflags         0x206   [ PF IF ]
cs             0x33    51
ss             0x2b    43
ds             0x0     0
es             0x0     0
fs             0x0     0
gs             0x0     0
```

此外，浮点型的第 6、第 7 个参数 f、g 分别保存到了 xmm0、xmm1 中。这两个值可以这样获得。

```
(gdb) p $xmm0.v4_float[0]
$4 = 3.1400001
(gdb) p $xmm1.v2_double[0]
$5 = 299792458
```

给 xmm0/xmm1 加上后缀（v4_float、v2_double）的原因是，GDB 将这些寄存器看做下面的联合：

```
union {
  float       v4_float[4];
  double      v2_double[2];
  int8_t      v16_int8[16];
  int16_t     v8_int16[8];
  int32_t     v4_int32[8];
  int64_t     v2_int64[8];
  int128_t    unit128;
} xmm0;
```

67

这是因为，xmm0 和 xmm1 的实际长度为 128 比特，但也能同时保存更小的变量。

指针类型的第 8 个参数 h 的处理方法与整型相同，保存到 r9 中。由于第 9、第 10 个参数也是指针，参数少的情况下会被保存到寄存器中，但在本 hack 的例子中，

寄存器不够用，因此这两个参数被保存到栈中进行传递。

查看栈的方法如下所示。虽然现在我们只关心两个参数，但却要显示 3 个，这是因为栈的开头保存了函数的返回地址。此外，用 g（giant word，双字）格式显示的原因是，在 x86_64 架构中，整数和指针等的大小就是 giant word。

```
(gdb) x/3g $rsp
0x7fff4c79fd78: 0x0000000000400558    0x0000000000600994
0x7fff4c79fd88: 0x00000000004006a3
```

如图 2-15 所示。

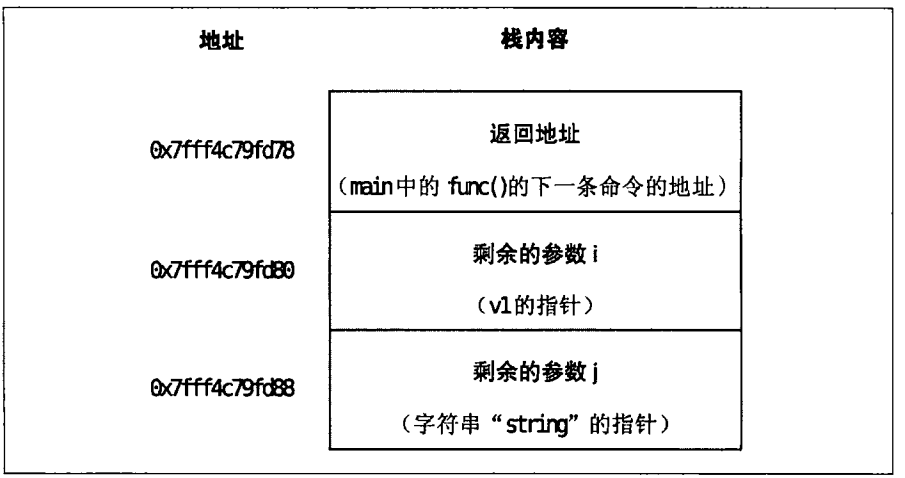


图 2-15 栈的内容

栈开头（地址 0x7fff4c78fd78）的值为返回地址，忽略掉，可以看出接下来的两个值分别是第 9、第 10 个参数 i、j。此外，由于 i 和 j 是指针，有时会希望确认它们指向的值或字符串。操作方法如下。

```
(gdb) printf "%.2f\n", *(float*)0x0000000000600994
0.01
(gdb) p (char*)0x00000000004006a3
$28 = 0x4006a3 "string"
```

68

有人可能会认为这里用十六进制数输入地址并不明智（当然，也有人认为只需用鼠标复制，十分简单）。用下面的方法也能获得同样的结果，尽管输入的字符数变化不大：

```
(gdb) pritnf "%.2f\n", *(float*)(*(unsigned long*)($rsp+0x8))
0.01
```

```
(gdb) p (char*)(*(unsigned long*)($rsp+0x10))
$29 = 0x4006a3 "string"
```

参考文献

- AMD64 Application Binary Interface
  http://www.x86-64.org/documentation/abi.pdf

总结

本 hack 说明了函数传递参数的确认方式。x86_64 下基本上使用寄存器，寄存器不够时才把参数放到栈中。

——大和一洋

函数调用时的参数传递方法（i386 篇）

讲述在 i386 架构上参数是如何传递给被调用的函数的。

i386 下的函数调用

这里使用[HACK#10]讲过的示例程序。笔者的环境中构建执行的结果如下所示。显然，只有指针的值 h、i、j 与[HACK#10]的结果不同。

```
a: 100, b: 35000, c: 5, d: A, e: 123456789
f: 3.140e+00, g: 2.998e+08
h: 0x80496d0, i: 0x80496d4, j: 0x80485bb
```

在 i386 上，原则上参数全部堆放到栈中。因此，在函数开头中断后，用下面的方式即可取得参数。取得第一个参数时使用 esp+4 是因为，i386 架构中栈的开头保存了返回地址，并且整数或指针类型的大小为 4 字节。

69

```
(gdb) p *(int*)($esp+4)
$4 = 100
(gdb) p *(long*)($esp+8)
$6 = 35000
(gdb) p *(short*)($esp+12)
$7 = 5
(gdb) p *(char*)($esp+16)
```

```
$8 = 65 'A'
(gdb) p *(long long*)($esp+20)
$9 = 123456789
```

下一个参数 f 也可以用同样的方法查看，但该参数的存储地址要比上一个参数的存储地址大 8 个字节，这是因为 i386 架构中 long long 类型和 double 类型的长度为 8 字节。

```
(gdb) printf "%.2e\n", *(float*)($esp+28)
3.14e+00
(gdb) printf "%.3e\n", *(double*)($esp+32)
2.998e+08
```

显示下一个参数时，要注意上一个值为 double 类型，它用掉了栈上的 8 个字节。

```
(gdb) p/x *(int*)($esp+40)
$15 = 0x80496d0
(gdb) p/x *(int*)($esp+44)
$16 = 0x80496d4
(gdb) p/x *(int*)($esp+48)
$17 = 0x80485bb
```

i386 中的寄存器调用

i386 也像 x86_64 那样可以将部分参数放在寄存器中进行函数调用。一般这种调用方式称为 fastcall（快速调用）。具体使用哪个寄存器则依赖于具体实现。

GCC 在函数声明中添加__attribute__((regparm(3)))，即可进行这种调用。这样即可使用 eax、edx 和 ecx 传递开头 3 个参数。

70

在 Linux 内核中可以使用 FASTCALL 或 asmregparm 等宏来实现该功能。

以下的说明中，对[HACK#10]的程序中的 func()做了如下改变。

```
__attribute__((regparm(3)))
void func(int a, long b, short c, char d, long long e, float f, double g, int *h, float
*i, char*j)
{
...
```

下面在 func()函数开头中断，确认一下寄存器的值。可以看出，第 1、2、3 个参数分别存储在 eax、edx 和 ecx 中。

```
(gdb) b *func
Breakpoint 1 at 0x8048374
(gdb) run
...
(gdb) i r
eax            0x64       100          ————————————参数a
ecx            0x5        5            ————————————参数c
edx            0x88b8     35000        ————————————参数b
ebx            0x8c5ff4   9199604
esp            0xbfdddf2c 0xbfdddf2c
ebp            0xbfdddf58 0xbfdddf58
esi            0xbfdddff4 -1075978252
edi            0xbfdddf80 -1075978368
eip            0x8048374  0x8048374
eflags         0x200286   2097798
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
fs             0x0        0
gs             0x33       51
```

第 4 个之后的参数与一般情况相同，保存到栈中。

71

```
(gdb) p *(char*)($esp+4)
$1 = 65 'A'
(gdb) p *(long long*)($esp+8)
$2 = 123456789
...
```

如果第 1 个参数或第 2 个参数为 long long 类型（64 比特类型）的变量，那么就会组合使用 eax、edx 等寄存器来传递参数。但是，如果第 2 个参数为 long long 类型，那么只有在第 1 个参数为 32 比特时才能进行寄存器传递。

总结

基本上，i386 将所有参数保存到栈上。但是，通过 GCC 的扩展功能__attribute__(regparm())，也可以实现部分参数的寄存器传递。

函数调用时的参数传递方法（C++篇）

介绍用 C++写成的程序中参数是如何传递给被调用的函数的。

C++语言的函数调用

考虑下述 C++源代码的执行情况。

```
#include <cstdio>

class foo {
   int a;
   int b;
public:
   void func(int x, int y);
};

void foo::func(int x, int y)
{
   a = x;
   b = y + 2;
}

int main(void)
{
   foo f1, f2;
   printf("f1: %p, f2: %p\n", &f1, &f2);  ⓪
   f1.func(5, 1);                          ①
   f2.func(-4, 2);                         ②

   return 0;
}
```

72

每个 class foo 的实例中（本例中为 f1、f2），其成员变量 a 和 b 的值都不同，通常是有多少个实例，就要有多少个存储这些成员的空间。相反，方法 func 的实体只需一个就足够了。实际上，在构建之后的 ELF 文件中，foo::func()函数的处理方式与 C 语言函数相同。不过，在执行 f1.func(5, 1)时，尽管没有在 func()中明确地指明要访问 f1 的 a 和 b，但它会自动把计算结果赋给 f1 的 a 和 b。实际上，func(int x,

int y)收到了要访问的实例的信息。也就是说，func()的行为相当于下述C语言函数。因此，调用方法时，传递的参数要比原型声明的参数多一个。

```
void func(class foo *this, int x, int y)
{
   this->a = x;
   this->b = y + 2;
}
```

在编译后的ELF文件中，不论是C++函数还是C函数，甚至是汇编语言函数，它们之间没有任何区别。但是，C++函数在编译时会经过函数名变换（mangle）。例如，上述代码中的foo::func在ELF文件中被变换成下面的符号。此外，mangle之后的符号名依赖于具体实现。

```
# nm foo | grep foo
0000000000400508 T _ZN3foo4funcEii
```

另外，要从被mangle后的符号（mangled symbol）中恢复源代码中的函数原型（即进行demangle），可以使用c++filt命令（或者nm的-C选项）。

73

```
# nm foo | grep foo | c++filt
0000000000400508 T foo::func(int, int)
# nm -C foo | grep foo
0000000000400508 T foo::func(int, int)
```

在x86_64中查看参数

要查看参数的值，应该先编译上述源代码，再用GDB运行。但是如前所述，对象的实例地址会被当做第一个参数传递。因此，首先要知道f1和f2的地址。源代码中的⓪就是显示地址的。此外，如果没有添加调试选项，就不能用foo::func这种源代码中的记法来设置断点，而必须像下面这样使用mangle之后的符号。

```
(gdb) b *_ZN3foo4funcEii      ———————— foo::func()的mangle过的符号
Breakpoint 1 at 0x400558
(gdb) run
...
f1: 0x7fffa34c3ab0, f2: 0x7fffa34c3aa0 —— f1和f2的实例地址
...
```

现在程序暂停在源代码中的①调用的foo::func()的开头。

```
(gdb) i r
...
rdx            0x1    1
rsi            0x5    5
rdi            0x7fffa34c3ab0  140735933070000
...
```

在 x86_64 中，C 语言函数会依次使用 rdi、rsi、rdx 传递参数（参见[HACK#10]），可知源代码中的第 1、第 2 个参数是通过 rsi 和 rdx 传递的。也就是说，rdi 中传递的是 f1 的地址。让我们来继续执行。接下来 foo::func()被调用时，可以看到 rdi 中传递的是源代码中②对应的对象实例的地址。

```
(gdb) c
...
(gdb) i r
...
rdx            0x2    2
rsi            0xfffffffc        4294967292
rdi            0x7fffa34c3aa0    140735933069984
...
```

74

在 i386 中查看参数

[HACK#11]中讲过，i386 中参数基本上通过栈来传递。我们使用与刚才同样的方法查看一下参数。可以看到，栈上传递的首先是实例地址，然后是源代码上的参数。

```
(gdb) b *_ZN3foo4funcEii
Breakpoint 1 at 0x8048454
(gdb) run
...
f1: 0xbf9874fc, f2: 0xbf9874f4 ————————— f1和f2的实例地址
...
(gdb) x/3 $esp+4
0xbf9874e0:    0xbf9874fc    0x00000005    0x00000001
(gdb) c
...
(gdb) x/3 $esp+4
0xbf9874e0:    0xbf9874f4    0xfffffffc    0x00000002
```

总结

本 hack 说明了 C++程序中参数是怎样传递给被调用的函数的。与 C 语言函数的不同点是，除了参数之外，对象指针也会被传递。

——大和一洋

怎样学习汇编语言

人们认为汇编语言很难理解，但对测试程序进行反汇编，就能很容易理解。

用 GCC 编译源代码，就能生成由机器语言（指令代码）构成的二进制文件。在分析 core dump 或 kernel dump 时，通过 gdb、crash 等调试工具可以看到反汇编之后的汇编代码。而最终，必须在由 C 语言等写成的源代码中找出问题的所在。

汇编语言的学习的确很难下手。本 hack 并不会分析 Intel 的架构手册，而是介绍如何制作测试程序和直观地学习汇编语言的方法。

75

查看反汇编的输出结果

要理解汇编语言，就必须了解 CPU 的寄存器和机器语言。寄存器相关内容请参见 **“HACK#8 Intel 架构的基本知识”**。

本 hack 要看一看，简单的 C 语言程序是如何变成汇编语言的。编译下述 assemble.c，编译环境为 32 位 Fedora8，gcc 版本为 4.1.2，objdump 版本为 2.17.50.0.18-1。

```
$ cat assemble.c
#include <stdio.h>

int global;
int func_op(void) { return 0; }

void func(void)
{
   unsigned long long val64 = 0;

   val64 = 0xffffeeeeddddcccc; ————————⑦
   global = 0x5555; ————————————————⑧
```

```
}
#define MAX_WORD 16

int main(void)
{
   unsigned int i = 0;
   char words[MAX_WORD]="Hello World";
   char word;

   int (*func_pointer)(void) = &func_op;

   i = 0xabcd; ————————————————①

   if (i != 0x1234) ————————————②
      i = 0;————————————————③

   while (i == 0) ——————————————④
      i++;————————————————⑤

   func();————————————————⑥
   i = func_pointer();————————⑨

   for (i=0; i<MAX_WORD-1 ; i++) ————⑩
      word = words[i]; ————————⑪

   return 0; ————————————————⑫
}
```

为了让汇编代码更容易理解，该测试程序花了一些心思。①、⑦、⑧没有像“i=0”这样赋十进制数，而是赋十六进制数，以方便阅读汇编代码。而且，main()中书写了代表性的 if 语句②、while 语句④和函数调用⑥等。

接下来进行编译，再用 objdump 反汇编。为了让汇编代码更容易理解，这里禁用了 gcc 的优化选项（-O0：横线、英文字母 O、数字 0），还给 objdump 加上了 --no-show-raw-insn 选项，使其不输出机器语言。

```
$ gcc -Wall -O0 assemble.c -o assemble
$ objdump -d --no-show-raw-insn assemble
```

```
...
0804839e <func>:
 804839e:  push   %ebp
 804839f:  mov    %esp,%ebp
 80483a1:  sub    $0x10,%esp
 80483a4:  movl   $0x0,-0x8(%ebp)
 80483ab:  movl   $0x0,-0x4(%ebp)
 80483b2:  movl   $0xddddcccc,-0x8(%ebp) ———————— ❼
 80483b9:  movl   $0xffffeeee,-0x4(%ebp)
 80483c0:  movl   $0x5555,0x80496c4 ———————— ❽
 80483ca:  leave
 80483cb:  ret

080483cc <main>:
 80483cc:  lea    0x4(%esp),%ecx
 80483d0:  and    $0xfffffff0,%esp
 80483d3:  pushl  -0x4(%ecx)
 80483d6:  push   %ebp
77 80483d7:  mov    %esp,%ebp ———————— ❶
 80483d9:  push   %ecx
 80483da:  sub    $0x24,%esp
...
 804840a:  movl   $0xabcd,-0x10(%ebp) ———————— ❶
 8048411:  cmpl   $0x1234,-0x10(%ebp) ———————— ❷-1 ┐
 8048418:  je     8048427 <main+0x5b> ———————— ❷-2 ├ if()
 804841a:  movl   $0x0,-0x10(%ebp) ———————— ❸      │
 8048421:  jmp    8048427 <main+0x5b> ———————— ❷-3 ┘
 8048423:  addl   $0x1,-0x10(%ebp) ———————— ❺ ┐
 8048427:  cmpl   $0x0,-0x10(%ebp) ———————— ❹-1 ├ while()
 804842b:  je     8048423 <main+0x57> ———————— ❹-2 ┘
 804842d:  call   804839e <func> ———————— ❻
 8048432:  mov    -0x8(%ebp),%eax
 8048435:  call   *%eax ———————— ❾
 8048437:  mov    %eax,-0x10(%ebp)
 804843a:  movl   $0x0,-0x10(%ebp) ———————— ❿-1 ┐
 8048441:  jmp    8048452 <main+0x86>           │
 8048443:  mov    -0x10(%ebp),%eax ———————— ⓫-1 ├ for()
 8048446:  movzbl -0x20(%ebp,%eax,1),%eax ———— ⓫-2 │
 804844b:  mov    %al,-0x9(%ebp) ———————— ⓫-3 ┘
```

```
804844e:  addl   $0x1,-0x10(%ebp) ──────────── ❿-2 ┐
8048452:  cmpl   $0xe,-0x10(%ebp) ──────────── ❿-3 ┘ for()
8048456:  jbe    8048443 <main+0x77>
8048458:  mov    $0x0,%eax ──────────────────── ⓬
804845d:  add    $0x24,%esp
8048460:  pop    %ecx
8048461:  pop    %ebp
8048462:  lea    -0x4(%ecx),%esp
8048465:  ret
...
```

汇编代码开头几行用 push 等指令生成栈帧。详细内容请参见**“HACK#9 调试时必须的栈知识”**。

设置变量的值：movl 指令

源代码中的带圈数字和汇编代码中的黑圈数字是一一对应的。❶用 movl 指令将 0xabcd 的值赋给-0x10(%ebp)。这里使用了 0xabcd，可知❶对应于①的变量初始化。-0x10(%ebp)表示从 ebp 寄存器中的地址值减去 0x10 后的地址值。假如 ebp 寄存器中 78 的地址值为 0x801000，那么变量 i 的地址为 0x801000 - 0x10，即 0x800ff0。

有的反汇编器会将-0x10 显示成 0xfffffff0。

只看❶的“mov %esp,%ebp”是无法理解到底是将%esp 的值赋给%ebp，还是将%ebp 的值赋给%esp。但是，看了测试程序就能知道，开头的汇编指令（movl）是将紧随其后的值（0xabcd）赋给寄存器（-0x10(%ebp)），因此可知 movl 指令相当于 C 语言的=。后面也继续使用了变量 i，后面的-0x10(%ebp)就是变量 i。

用 if 语句比较变量：cmpl 指令

❷-1 为 cmpl 指令，比较（compare）0x1234 与-0x10(%ebp)（变量 i）。接下来的❷-2 为如果“i == 0x1234”就不执行“i = 0”（❸），用 je 指令（jump equal）跳转到❹-1。cmpl 命令比较变量 i 和 0x1234，如果为真，则设置 CPU 的 ZF 寄存器为 1。je 指令在 ZF 标志为 1（即 i==0x1234）时执行跳转。

while 语句的汇编代码

❹-1 的 cmpl 指令判断 while 语句的条件表达式。如果 i == 0，则用 je 指令跳转到❹-2，用 addl 指令执行 i++（❺）；如果 i != 0，就不跳转到❹-2，而是执行❻。

函数调用：call 指令

call 指令可以跳转到函数后再返回。从❻可见，从<func>处控制权转移到了 func()函数。

❼处两次执行 movl 指令，赋予 32 比特的值。在 32 位操作系统上定义“unsigned long long”和 64 比特变量，其汇编代码就会成为这样。

❽给全局变量赋值。到目前为止的局部变量都像-0x10(%ebp)这样以 ebp 寄存器的偏移量来表示，值写进了栈内的地址。而全局变量则使用“0x80496c4”这样的地址值来表示。

func()函数处理结束后，返回 main()的❻。

79

函数指针调用

调用函数指针 func_op()的代码如❾所示。函数指针保存在 eax 中，要像*%eax 这样加上*号。

数组操作：movzbl 指令

接下来依次分析⑩的 for 语句和⑪。首先，for 语句的 i=0 就是❿-1。接下来无条件跳转到❿-3，❿-3 比较变量 i 和 MAX_WORD-1。如果继续，for 语句则跳转到⓫-1，⓫-1 将变量 i 赋给 eax 寄存器。至于⓫-2，其含义是将 words(-0x20%ebp)的第 i(%eax)个位置上的 1 字节赋给 eax 寄存器，也就是 word = words[i]。⓫-3 的%al 是 eax 寄存器的低 8 比特。由于是 char 类型的变量，所以只需 8 比特。

如果是 int（4 字节）数组，其代码如下。

```
8048427:  mov   -0x50(%ebp,%eax,4),%eax
804842b:  mov   %eax,-0xc(%ebp)
```

返回值设置

⓬为 return 语句，给返回值赋 0。return 语句的返回值为 int 等 4 字节以下的情况，则要将返回值放到 eax 中，是以 eax 寄存器为通用寄存器的。

总结

像本 hack 这样，测试程序有助于理解汇编语言。通过-O0 学习之后，被-O2 优化过的汇编语言也会变得容易理解。还能学到分析 core dump 或 kernel dump 时的大部分必要知识。

参考文献[注2]

- IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 A：命令セット・リファレンス A-M
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2A_i.pdf
- IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 B：命令セット・リファレンス N-Z
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2B_i.pdf

——大岩尚宏

80

从汇编代码查找相应的源代码

crash 命令的反汇编器可以找出在源代码中的相应位置。

不论是调查用户应用程序，还是调查内核，抑或是调查 core dump 和 kernel dump，都需要通过汇编代码定位相应的源代码。

本 hack 以 Linux 内核版本 2.6.19 的 journal_commit_transaction()为例，介绍相关的经验技巧。

注 2：参考文献的英文版：《Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M》和《Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z》，下载地址为 http://www.intel.com/products/processor/manuals/。

——译者注

用 crash 反汇编

首先反汇编 journal_commit_transaction()。

```
# crash /boot/vmlinux-2.6.19
...
crash> dis journal_commit_transaction
0xf88580e0 <journal_commit_transaction>:        push %ebp
0xf88580e1 <journal_commit_transaction+1>:      mov  %eax,%ebp
...
0xf88585bc <journal_commit_transaction+1244>:   call 0xf88580a0 <journal_do_ submit_data>——❷
0xf88585c1 <journal_commit_transaction+1249>:   mov  %ebx,%eax
0xf88585c3 <journal_commit_transaction+1251>:   call 0xc120e1b9 <_spin_lock> ————❸
0xf88585c8 <journal_commit_transaction+1256>:   movl $0x0,0x24 (%esp)
0xf88585d0 <journal_commit_transaction+1264>:   jmp  0xf885869c ————————————❹
0xf88585d5 <journal_commit_transaction+1269>:   mov  0x24(%eax), %esi ————❶、❺-1
```

我们来试着找找❶这一行对应的源代码。在分析空指针访问、内存非法访问等时，经常会查找 mov 命令在源代码中的相应位置。

根据前后的信息确定源代码范围

❶为 journal_commit_transaction()的第 1269 字节，但从 journal_commit_ transaction+1 开始按顺序阅读就太麻烦了，先来看看前后的信息吧。从❷可以看出，它调用了 journal_do_submit_data()，但是源代码中 journal_commit_transaction()并没有直接调用 journal_do_submit_data()。这是由于优化使得没有显式声明为__inline__的 static 函数也被展开了。源代码中 journal_do_ submit_data()的调用路径如图 2-16 所示。

81

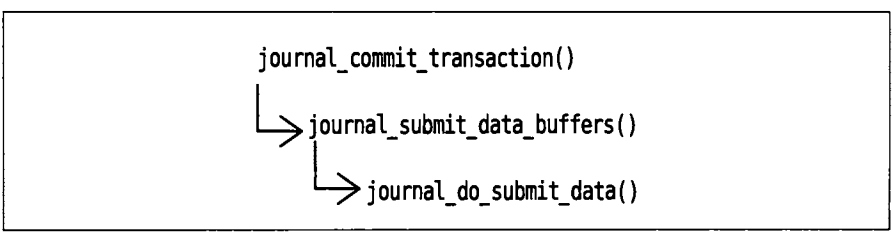


图 2-16 journal_do_submit_data()的调用路径

journal_submit_data_buffers()中调用 journal_do_submit_data()的地方有 3 处。只要能确定❷对应于这 3 处中的哪一处，应该就能明确了。

❸的地方调用了 spin_lock()。而 journal_do_submit_data()之后调用 spin_lock()的地方只有一处，那就是 journal_submit_data_buffers()中的最后那次调用。

[fs/jbd/commit.c]

```
static void journal_submit_data_buffers(journal_t *journal,
                        transaction_t *commit_transaction)
{
...
write_out_data:
   cond_resched();
   spin_lock(&journal->j_list_lock);

   while (commit_transaction->t_sync_datalist) {
   ...
      if (buffer_dirty(bh)) {
         if (test_set_buffer_locked(bh)) {
         ...
            journal_do_submit_data(wbuf, bufs); /* 第1个 */
            bufs = 0;
            lock_buffer(bh);          /* 有这一行，因为不一样 */
            spin_lock(&journal->j_list_lock);
         }
         locked = 1;
      }
   ...
      if (locked && test_clear_buffer_dirty(bh)) {
      ...
            journal_do_submit_data(wbuf, bufs);  /* 第2个 */
            bufs = 0;
             goto write_out_data;          /* goto之后调用了cond_resched()*/
      }                                    /* 因此不一样 */
      }
      ...
      }
      spin_unlock(&journal->j_list_lock);
      journal_do_submit_data(wbuf, bufs); /* 确认一下这行之后(离开journal_submit_
}                                          data_buffers()之后)会怎样 */ ————②
```

82

请看下面的源代码。journal_submit_data_buffers()返回之后立即调用 spin_lock()。因此可以认为，②与❷是一致的。

接下来看看❹，它无条件跳转到了❹-1。

```
0xf8858697 <journal_commit_transaction+1463>: call 0xc1023496 <cond_resched_lock>
0xf885869c <journal_commit_transaction+1468>:  mov  0x14(%esp), %edx ————————❹-1
0xf88586a0 <journal_commit_transaction+1472>:  mov  0x18(%edx), %eax ————————❹-2
0xf88586a3 <journal_commit_transaction+1475>:  test %eax,%eax
0xf88586a5 <journal_commit_transaction+1477>:  jne  0xf88585d5 ————————————❺
0xf88586ab <journal_commit_transaction+1483>:  mov  $0x1,%al
```

与源代码对比就很明白了。这与 **"HACK#13 怎样学习汇编语言"** 中的 while 语句是一样的。接下来看看下面的源代码。黑圈数字和带圈数字一一对应。

```
void journal_commit_transaction(journal_t *journal)
{
   transaction_t *commit_transaction;
...
   err = 0;
   journal_submit_data_buffers(journal, commit_transaction);
                                          /* 立即调用了 spin_lock() */
   /*
    * Wait for all previously submitted IO to complete.
    */
   spin_lock(&journal->j_list_lock); ————————————————③
   while (commit_transaction->t_locked_list) { ————————④、④-1、④-2
      struct buffer_head *bh;

      jh = commit_transaction->t_locked_list->b_tprev; ————⑤-1
      bh = jh2bh(jh);
83 ...
   }
...
```

❹无条件跳转到了❹-1。此处把某个变量赋给了 edp。这一点与源代码不一致，就不再细看了。接下来看看❹-2。如果它是 while()语句，那么 0x18(%edx)应该是 commit_transaction->t_locked_list。

确认寄存器偏移量和结构的成员

下面确认 0x18(%edx)是不是 commit_transaction->t_locked_list，不用看源代码，可使用 crash 的 struct 命令。

```
crash> struct -o transaction_t
struct: invalid data structure reference: transaction_t
crash>
```

看不到该结构的详细情况。这是由于 jbd 模块的符号无法解析的缘故。那么执行以下的 mod 命令。

```
crash> mod
 MODULE   NAME         SIZE  OBJECT FILE
...
f8863100 jbd         58152  (not loaded)  [CONFIG_KALLSYMS]
...
crash>
```

可见符号无法解析（not loaded）。这种情况下要用 mod 命令加载模块。

```
crash> mod -s jbd
 MODULE   NAME         SIZE  OBJECT FILE
f8863100 jbd          58152  /lib/modules/2.6.19/kernel/fs/jbd/jbd.ko
crash>
```

加载模块之后再次确认结构。

```
crash> struct transaction_t
No struct type named transaction_t.
struct transaction_s {
   journal_t *t_journal;
   tid_t t_tid;
   enum {T_RUNNING, T_LOCKED, T_RUNDOWN, T_FLUSH, T_COMMIT, T_FINISHED} t_state;
   long unsigned int t_log_start;
   int t_nr_buffers;
   struct journal_head *t_reserved_list;
   struct journal_head *t_locked_list;
...
```

84

符号已可以解析，可以看到结构的内容了。但这跟看源代码没什么两样，而且还出

现了 No struct type……这是因为该结构是使用 typedef 定义的，真正的结构名为 transaction_s。

```
[include/linux/journal-head.h]
   typedef struct transaction_s   transaction_t;  /* Compound transaction type */
```

我们要与汇编代码对照，因此加上-o参数，显示结构成员的偏移量。

```
crash> struct -o transaction_s
struct transaction_s {
  [0] journal_t *t_journal;
  [4] tid_t t_tid;
  [8] enum {T_RUNNING, T_LOCKED, T_RUNDOWN, T_FLUSH, T_COMMIT, T_FINISHED} t_state;
 [12] long unsigned int t_log_start;
 [16] int t_nr_buffers;
 [20] struct journal_head *t_reserved_list;
 [24] struct journal_head *t_locked_list;
...
crash> eval 24
hexadecimal: 18     /* 十六进制值 */
    decimal: 24
      octal: 30
     binary: 00000000000000000000000000011000
crash>
```

t_locked_list 的偏移量为 24，写成十六进制数就是 18，可以确认❹-2 的 0x18(%edx) 就是 commit_transaction->t_locked_list。上例中由于偏移量为十进制数而使用了 eval 命令进行变换，其实 crash 有个 hex 命令可以把显示内容变成十六进制数。另外，上面已知道要确认的成员是 t_locked_list，可以直接在 struct 命令中指定成员。

85

```
crash> hex
output radix: 16 (hex)
crash> struct -o transaction_s.t_locked_list
struct transaction_s {
 [0x18] struct journal_head *t_locked_list;
}
crash>
```

这样就非常容易理解了。像这样将结构的偏移量与汇编代码中的偏移量比较，就可以跟源代码对应起来。

❹-2之后为test命令，对两个%eax寄存器进行AND运算，其结果为0，则设置ZF标志为1。eax寄存器（commit_transaction->t_locked_list）为NULL时，ZF标志为1。下一步❺使用jne命令在ZF标志为0时跳转，即commit_transaction->t_locked_list != NULL的情况下跳转到❺-1，进入while语句。

❺-1为while语句的第一条指令。

那么来看看源代码的⑤-1。这里用的是commit_transaction->t_locked_list->b_tprev，但刚才while语句的条件中用到了commit_transaction->t_locked_list，这个变量已经放进了eax寄存器中。0x24(%eax)的偏移量又多了一些，应该就是commit_transaction->t_locked_list->b_tprev。确认一下。

```
crash> struct -o journal_head
struct journal_head {
  [0x0] struct buffer_head *b_bh;
...
  [0x24] struct journal_head *b_tprev;
  [0x28] transaction_t *b_cp_transaction;
...
crash>
```

这样就能看出，汇编代码的❶就是源代码中的jh = commit_transaction->t_locked_list-> b_tprev;。

确认源代码文件名和行号

crash的dis命令有个选项-l，可以输出相应源代码的文件名和行号。

86

```
crash> dis -l journal_commit_transaction
/root/linux-2.6.19/fs/jbd/commit.c: 281
0xf88580e0 <journal_commit_transaction>:        push    %ebp
0xf88580e1 <journal_commit_transaction+1>:      mov     %eax,%ebp
0xf88580e3 <journal_commit_transaction+3>:      push    %edi
0xf88580e4 <journal_commit_transaction+4>:      push    %esi
0xf88580e5 <journal_commit_transaction+5>:      push    %ebx
0xf88580e6 <journal_commit_transaction+6>:      sub     $0x64,%esp
/root/linux-2.6.19/fs/jbd/commit.c: 284
0xf88580e9 <journal_commit_transaction+9>:      mov     0x114(%eax), %eax
0xf88580ef <journal_commit_transaction+15>:     mov     %eax, 0x1c(%esp)
```

```
/root/linux-2.6.19/fs/jbd/commit.c: 309
0xf88580f3 <journal_commit_transaction+19>:    testb    $0x8, 0x0(%ebp)
0xf88580f7 <journal_commit_transaction+23>:    je       0xf8858105 <journal_commit_
transaction+37>
...
0xf884f5d0 <journal_commit_transaction+1264>:  jmp      0xf884f69c
<journal_commit_transaction+1468>
/root/linux-2.6.19/fs/jbd/commit.c: 437
0xf884f5d5 <journal_commit_transaction+1269>:  mov      0x24(%eax), %esi
include/linux/jbd.h: 324
0xf884f5d8 <journal_commit_transaction+1272>:  mov      (%esi), %ebx
```

在 Linux 内核版本 2.6.19 的源代码中有如下代码。左侧数字为行号。

```
434   while (commit_transaction->t_locked_list) {
435      struct buffer_head *bh;
436
437      jh = commit_transaction->t_locked_list->b_tprev;
438      bh = jh2bh(jh);
439      get_bh(bh);
440      if (buffer_locked(bh)) {
```

要使用该选项，必须打开内核的 CONFIG_DEBUG_INFO。但是并不一定完全一致，因此先理解汇编代码，再以此作为参考比较方便。

总结

本 hack 介绍了根据汇编代码确定内核源代码位置的技巧，以及所用的 crash 这个极其方便的命令。

87

参考文献[注3]

- IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中

注 3：参考文献的英文版：《Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M》和《Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z》，下载地址为 http://www.intel.com/products/processor/manuals/。

——译者注

巻 A：命令セット・リファレンス A-M
http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2A_i.pdf

- IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 B：命令セット・リファレンス N-Z
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2B_i.pdf

——大岩尚宏

第3章

内核调试的准备

hack #15~#25

本章介绍 Linux 内核调试的基本方法。包括 Oops 信息的解读方法、串行控制台的使用方法、通过网络获取内核信息、SysRq 键、各种 dump 的获取方法、crash 命令的使用方法、使用 IPMI 和 NMI watchdog 获取 crash dump 的方法、内核的特殊汇编语言等各种内核调试基础知识。

Oops 信息的解读方法

讲述内核中发生致命错误时输出的 Oops 信息的解读方法。

Oops 信息

Oops 信息是内核中发生致命错误时输出的内核信息。下面的例子是最典型的 Oops 信息的一种：x86_64 架构上发生无法处理的页面错误。这个 Oops 信息中大致包含了错误概况、加载的模块、寄存器信息、栈跟踪信息等。不同架构和内核版本的显示稍稍有些不同，但大体内容是一样的。

```
Unable to handle kernel NULL pointer dereference at 0000000000000000 RIP:
 [<ffffffff881d0002>] :demo:init_oopsdemo+0x2/0xe
PGD 18b4c067 PUD 18b52067 PMD 0
Oops: 0002 [1] SMP
last sysfs file: /block/dm-1/range
CPU 1
Modules linked in: demo nfs lockd fscache nfs_acl sunrpc ipv6 dm_multipath video sbs
backlight i2c_ec i2c_core button battery asus_acpi acpi_memhotplug ac lp floppy sg
serio_raw parport_pc parport pcspkr e1000 shpchp ide_cd cdrom dm_snapshot dm_zero
dm_mirror dm_mod ata_piix libata mptspi mptscsih mptbase scsi_transport_spi sd_mod
```

```
90  scsi_mod ext3 jbd ehci_hcd ohci_hcd uhci_hcd
    Pid: 2473, comm: insmod Not tainted 2.6.18-53.5 #1
    RIP: 0010:[<ffffffff881d0002>] [<ffffffff881d0002>] :demo:init_oopsdemo+0x2/0xe
    RSP: 0000:ffff81001701be60 EFLAGS: 00010246
    RAX: 0000000000000000 RBX: ffffffff883e1400 RCX: 0000000000000000
    RDX: ffffffff883e1400 RSI: 0000000000000296 RDI: ffffffff802ea344
    RBP: ffff81001cb98800 R08: ffff81001f669820 R09: 0000000000000020
    R10: 0000000000000001 R11: 0000000000000000 R12: ffff81001cb98c00
    R13: ffffffff883e1400 R14: ffff81001cb98bb8 R15: ffffc20000155f70
    FS:  00002aaaaaad8210(0000) GS:ffff81001fc40840(0000) knlGS:0000000000000000
    CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
    CR2: 0000000000000000 CR3: 0000000016ade000 CR4: 00000000000006e0
    Process insmod (pid: 2473, threadinfo ffff81001701a000, task ffff81001f643820)
    Stack: ffffffff800a397c 000000000000001a 0000000000000000 000000001701be78
     0000000000000000 0000000000000000 0000000000000000 0000000000000000
     0000000000000000 0000000000000000 ffffc200001605a8 ffff81001c852ea0
    Call Trace:
     [<ffffffff800a397c>] sys_init_module+0x16aa/0x185f
     [<ffffffff8000c1b0>] _atomic_dec_and_lock+0x39/0x57
     [<ffffffff8005c116>] system_call+0x7e/0x83


    Code: c7 04 25 00 00 00 00 17 08 76 19 c3 ff ff 05 00 00 00 ff ff
    RIP  [<ffffffff881d0002>] :demo:init_oopsdemo+0x2/0xe
     RSP <ffff81001701be60>
```

Oops 信息的第 1 行是错误内容：Unable to handle kernel NULL pointer dereference at 0000000000000000[注1]，随后的 RIP: [<ffffffff881d0002>]是错误发生地址，也就是 RIP 寄存器的值。:demo:init_oopsdemo+0x2/0xe 的意思是，错误发生的地址是 demo 模块内的 init_oopsdemo()函数的第 2 个字节。最后的/0xe 是 init_oopsdemo()函数的大小。此外，如果没有 KALLSYMS 内核选项，就只显示逻辑地址，而不会显示模块名和函数名等。接下来的一行 PGD 18b4c067 PUD 18b52067 PMD 0 是试图访问的地址（本例中为 0）的页表（page table）信息。

Oops:之后的数值是错误代码，在后面的[]内的数值是与页面有关的 Oops 信息被显示的次数。之后显示内核的重要特性 SMP 和 PREEMPT 的配置情况。这条信息所在

注 1： 意为：无法处理指向地址 0 的 NULL 指针。

的内核启用了 SMP 支持，是非可抢占内核（kernel preemption），所以只显示 SMP。last sysfs file:是最后打开的 sysfs 文件名。

91

下一行的 CPU 后面的数字是错误所在逻辑 CPU 的编号，接下来的 Modules linked in: 之后是加载了的模块列表。下一行显示了错误发生时该 CPU 正在运行的进程的进程 ID（2473）、进程名（insmod）、内核污染原因（Not tainted）、版本（2.6.18-53.5）。内核的污染原因包括私有（proprietary）驱动加载（P）、模块强制加载（F）、模块强制卸载（R）、机器检查（machine check）异常发生（M）、检测到错误页（B）等。如果涉及了某项原因，就会显示成 Tainted: PF R B 这样；如果不存在上述问题，就像上例那样显示 Not tainted。

从 RIP:到 CR2:的 9 行只有当错误发生在内核模式下时才显示。Stack:是栈开头部分的值，此处显示的大小为 kstack 内核选项指定的大小，不指定的情况下默认值为 12。Code:是错误发生时 RIP 指向的地址处的开头 20 个字节的代码。

最后的 RIP 和 RSP 是错误发生时 RIP 和 RSP 的值。像 RIP 和 RSP 这样，有些内容是重复显示的，其值当然是相同的。

Oops 显示测试

上例中给出的 Oops 信息是由下述代码发生的。将该代码编译成 demo.ko 模块，加载后就会发生 Oops。

```
static __init int init_oopsdemo(void)
{
   *((int*)0x00) = 0x19760817;
   return 0;
}
module_init(init_oopsdemo);

static __exit void cleanup_oopsdemo(void)
{
}
module_exit(cleanup_oopsdemo);
MODULE_LICENSE("GPL");
```

92

这段代码在模块的初始化函数 init_oopsdemo()的开头试图对逻辑地址 0 进行写

入。地址 0 没有相应的实际页面，当然会发生处理异常错误，显示 Oops 信息。init_oopsdemo()的汇编代码如下所示。

```
0:  31 c0                   xor  %eax,%eax
2:  c7 04 25 00 00 00 00    movl $0x19760817,0x0
9:  17 08 76 19
d:  c3                      retq
```

错误的原因是地址 2 处的 movl 指令，可见 Oops 信息的 demo:init_ oopsdemo+0x2/0xe 是正确的。

首先要看什么

Oops 信息显示的是调试信息。用不着 crash 工具，或者在无法取得转储文件的设备的情况下，单凭 Oops 信息也能明白错误原因。那么，首先应该看哪里呢？答案是开头的信息，本例中直接显示了原因。接下来是错误发生的地址，本例中开头第 2 行的信息包含了错误地址，如果没有这个信息，可以看 RIP（i386 下为 EIP）。一般，分析原因需要查看汇编代码的相应部分。本例中 RIP（EIP）指向 demo 模块中 init_oopsdemo()函数的第 2 字节。从这一部分就可以知道，显然是对地址 0 进行了非法写入操作。

另外，EFLAGS 的第 9 个比特 IF（中断许可标志）的值有时也有用。比如，用户空间的程序通常不会进行中断的许可和禁止操作，相反在内核中要频繁进行这种操作。因此，对于由于在不能发生中断的地方发生了中断处理等引起的 bug，如果在确认调用路径的同时确认该标志，就比较容易发现问题。

总结

本 hack 说明了内核中发生致命问题时输出的 Oops 内核消息的解读方法。

——大和一洋

93

使用 minicom 进行串口连接

介绍 minicom 的设置方法，以及使用 SysRq 键时所需的 break 信号。

Linux 的 minicom 命令可以进行串口连接。使用串口控制台和 minicom 命令可以远程使用控制台界面，而且在网络无法连通的机器上也能进行远程操作。如果想查

看控制台显示的信息，而一般的显示器画面无法全部显示，导致信息显示不全，此时使用 minicom 的日志采集功能就非常方便了。

调试内核时经常将串口控制台和 minicom 组合使用。这里介绍一般的使用方法和一些有助于调试的功能。

准备

把要连接的机器（目标机）和执行 minicom 的机器（宿主机）用交叉串口线连接起来，然后在目标机上设置内核参数。以下是设置例子。

```
console=ttyS0,115200n8 console=tty0
```

串口 ttyS0 的波特率被设置为 115200，无奇偶检验（n），8 个数据位。要想同时连接串口控制台和普通显示器，可像上例这样同时设置 console=tty0，这样就能在两边同时输出。

要想从串口控制台进行 GRUB 等启动器的操作，可能需要在 BIOS 设置界面中设置串口控制台。

接下来确认是否已设置好 getty。下面的例子假设目标机使用的是 RedHat 系列发行版。

```
# vi /etc/inittab
...
co:2345:respawn:/sbin/agetty ttyS0 115200 vt100-nav /* 添加这一行 */
1:2345:respawn:/sbin/mingetty tty1
...
```

94

进一步为了让 root 能从串口控制台登录，需要添加以下设置。

```
# vi /etc/securetty
...
ttyS0 /* 添加 */
...
```

使用 minicom

登录到宿主机，执行 minicom 命令。

```
# minicom
```

画面变为如图 3-1 所示。

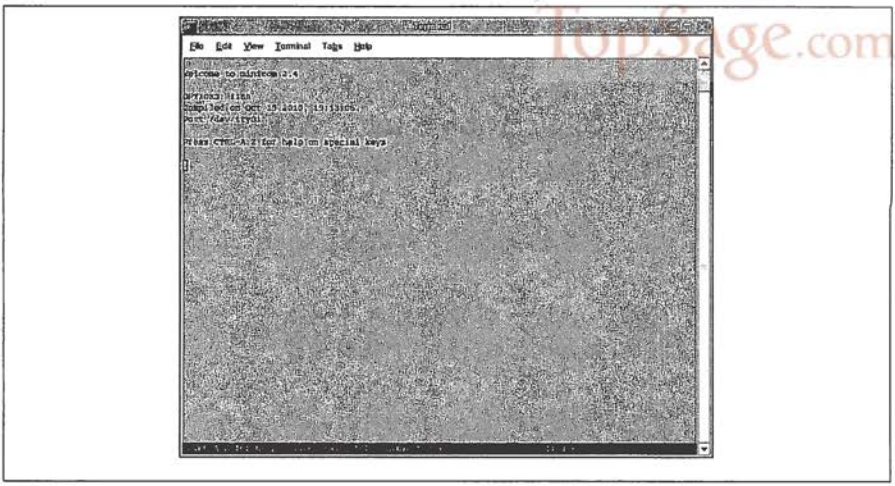


图 3-1 minicom 界面

同时按下 Ctrl 和 A（Ctrl-A）键，再按 Z 键，即可显示帮助画面。以后这种快捷键操作记做 Ctrl-AZ，画面变为如图 3-2 所示。

95

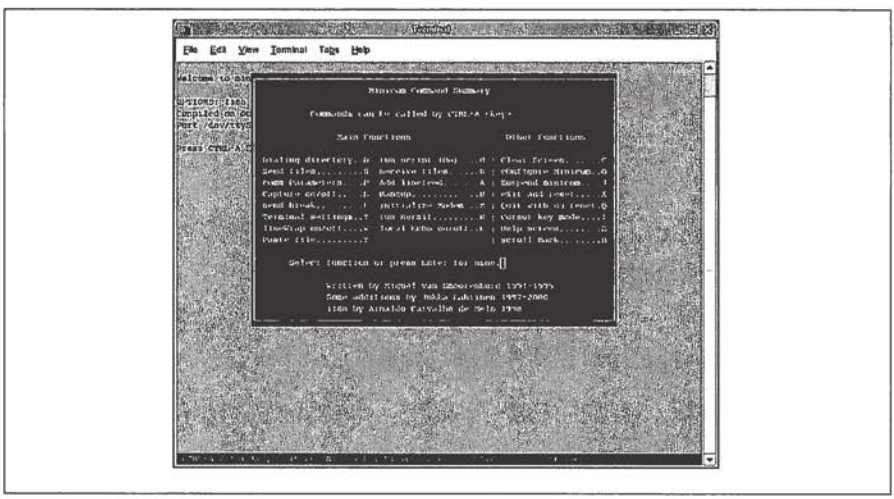


图 3-2 minicom 帮助界面

按 Ctrl-AO 组合键，设置波特率与串口的相同，即内核参数中的 ttyS0 为 115200。波特率一致后即显示登录提示符，登录后即可正常操作。使用 Ctrl-AL 组合键可以记录串口控制台的日志。

调试经常使用 break 信号，按 Ctrl-AF 组合键（图 3-3）即可发送。

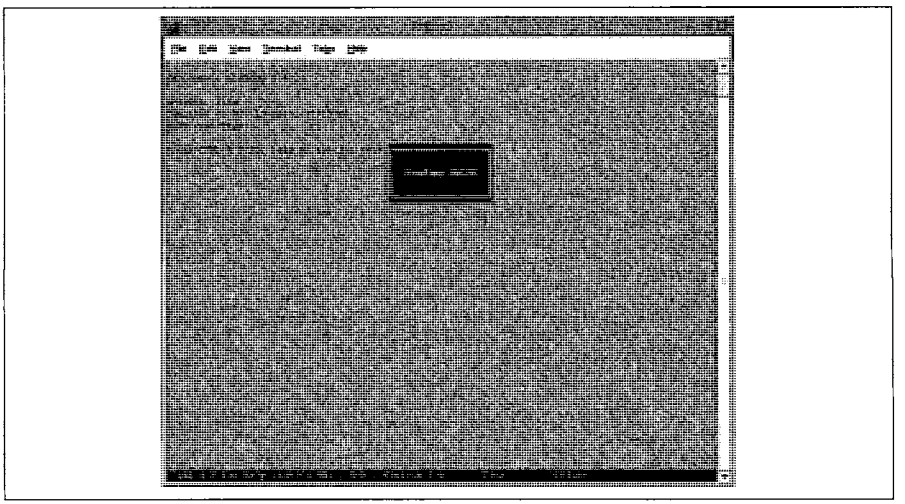


图 3-3 minicom 发送 break

break 信号的效果与键盘上同时按下 Alt 和 SysRq 键的效果相同。

发送 break 信号后要在 5 秒钟之内输入下一个命令键，按 Enter 键后就会显示下面的帮助信息。

```
SysRq : HELP : loglevel0-8 reBoot Crash tErm kIll saK showMem powerOff showPc unRaw Sync
showTasks Unmount shoWcpus
```

96

总结

本 hack 介绍了 minicom 的设置和使用方法。通过 minicom 可以远程使用 SysRq 键（break 信号），而且还能记录 SysRq 命令的日志，对于内核调试十分方便。有关 SysRq 键请参见 **“HACK#18 使用 SysRq 键调试”**。

参考文献

- Linux 内核源代码的附属文档
  Documentation/kernel-parameters.txt
  Documentation/serial-console.txt
  Documentation/sysrq.txt
- Remote Serial Console HOWTO

http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/index.html

——大岩尚宏

通过网络获取内核消息

本 hack 介绍使用 netconsole 模块将内核信息通过网络发送至远程主机的方法。

调查 kernel panic 时十分有用的 Oops 信息是作为内核信息输出的。本 hack 将告诉您，如何通过网络发送内核信息，以便获取那些重启后会消失或者是 panic 发生时的 Oops 信息。

netconsole 功能本身就十分有用，如能与 kdump、diskdump 等 dump 配置组合起来放到服务器上，这更利于发生故障时查找原因。

Oops 信息是什么

Oops 信息指 Linux 内核中发现致命错误时输出的信息。调查问题原因时要用到的 CPU 寄存器信息、系统调用的跟踪信息等，都包含在 Oops 信息中。

通常，这种信息会输出到控制台，而不会保存到日志中。另外，如果控制台上一页显示不完导致需要滚动屏幕，滚动部分的信息就会丢失，使得调查原因异常困难。

97

为调查 kernel panic 的原因，就必须收集 panic 发生时控制台输出的所有信息，为此需要做好事先准备。

Oops 信息的详细内容请参见 **"HACK#15 Oops 信息的解读方法"**。

netconsole 是什么

netconsole 模块可以让 printk 信息（输出到控制台的信息）通过网络（UDP）发送到远程主机。

netconsole 模块不能用于获取 crash dump，也不能进行一般的控制台输入输出。

netconsole 的长处

与串口控制台不同，netconsole 不需要串口线和串口。另外，它可以轻易地将多台服务器的内核信息收集到一台服务器上。在无法使用串口的环境下，使用 netconsole 将 Oops 信息保存到远程主机更方便、可行。

网络相关的 kernel panic，以及与操作系统启动到网络和 netconsole 模块启动之间发生的 panic 相关的内核信息，请通过串口控制台或其他方法获得。串口控制台的详细情况参见 **“HACK#16 使用 minicom 进行串口连接”**。

netconsole 的设置方法

使用 netconsole 时，除了发送端（启动了 netconsole 模块的一端）之外，还要有接收的远程主机。

本 hack 的环境如下所示。

```
（发送端机器）
IP 地址：10.1.1.100

（日志接收用远程机器）
IP 地址：10.1.1.200
```

98

发送端设置

1. 加载 netconsole 模块

netconsole 是个字符串参数“netconsole”，其格式如下。

```
netconsole=[src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-macaddr]
           src-port……发送端端口号
           src-ip……发送端 IP 地址
           dev……网络设备
           tgt-port……接收端端口号
           tgt-ip……接收端 IP 地址
           tgt-macaddr……接收端 MAC 地址
```

下面的设置可以发送至日志接收端，即远程机器的 syslog，端口号设置为 UDP 514（syslog）。

```
# modprobe netconsole netconsole=6665@10.1.1.100/eth0,514@10.1.0.200/00:0C:29:45:EB:FA
```

2. 确认

确认 netconsole 模块已经加载。

```
# lsmod | grep netconsole
```

发送端的/var/log/messages 中有下面的信息。

```
Feb 13  17:39:36 hostname kernel: netconsole: local port 6665
Feb 13  17:39:36 hostname kernel: netconsole: local IP 10.1.1.100
Feb 13  17:39:36 hostname kernel: netconsole: interface eth0
Feb 13  17:39:36 hostname kernel: netconsole: remote port 514
Feb 13  17:39:36 hostname kernel: netconsole: remote IP 10.1.0.200
Feb 13  17:39:36 hostname kernel: netconsole: remote ethernet address 00:0c:29:45:eb:fa
Feb 13  17:39:36 hostname kernel: netconsole: network logging started
```

netconsole 模块在操作系统启动时不会自动加载。RedHat 系列的发行版中，要想在操作系统启动时加载 netconsole，需要在/etc/sysconfig/network-scripts/ifcfg-ethX 文件或/etc/rc.local 文件中添加 1 中的命令。

99

接收端设置

1. 修改 syslog 设置

为了让接收端的 syslog 能接收远程主机的日志，需要在/etc/sysconfig/syslog 文件的 SYSLOGD_OPTIONS 中添加-r 选项。

```
[修改前]
SYSLOGD_OPTIONS="-m 0"

[修改后]
SYSLOGD_OPTIONS="-m 0 -r"
```

2. 重新启动 syslogd

保存修改之后重新启动 syslogd。

在 RedHat 系列发行版中使用以下命令。

```
# service syslog restart
```

输出测试

确认一下发送端控制台上输出的内核信息能否在接收端的 syslog 中出现。

本 hack 中使用 SysRq 键在控制台界面输出内核信息。

1. 启用 SysRq 键

要在发送端机器上启用 SysRq 键，需要按照下述方式修改/etc/sysctl.conf。

```
[修改前]
kernel.sysrq = 0

[修改后]
kernel.sysrq = 1
```

执行下列命令，使修改生效。

```
# sysctl -p
```

100

2. 本地测试并确认

执行下列命令，向控制台输出内核信息。

```
# echo h > /proc/sysrq-trigger
```

3. 在接收端确认

确认日志接收用的远程机器的/var/log/messages 中是否包含以下信息（hostname 是发送端服务器的主机名）。

```
Feb 13 17:39:36 hostname SysRq :
Feb 13 17:39:36 hostname HELP :
Feb 13 17:39:36 hostname loglevel0-8
Feb 13 17:39:36 hostname reBoot
...
Feb 13 17:39:36 hostname Unmount
Feb 13 17:39:36 hostname shoWcpus
```

总结

本 hack 介绍了 netconsole 模块，它能将 panic 时的 Oops 信息、内核信息等通过网络发送到其他服务器。在服务器上设置好 netconsole，有助于故障发生时分析原因。与 kdump、diskdump 等 dump 功能组合使用效果更好。

参考

- “HACK#19 使用 diskdump 获取内核崩溃转储”
- “HACK#20 使用 kdump 获取内核崩溃转储”

——吉田俊辅

使用 SysRq 键调试

本 hack 介绍在内核调试中经常用到的 SysRq 键，以及通过 SysRq 键能获得什么信息。

使用 SysRq 键在内核调试中极其方便。SysRq 键使用了中断，因此在无法登录、按键无响应时也能正常使用，但是在内核禁止中断时失去响应的情况下则不能使用（禁止中断时失去响应的情况，可以使用 NMI watchdog（参见 **“HACK#23 用 NMI watchdog 在死机时获取崩溃转储”**））。

101

此外，在启动时、重新启动之前等无法获取转储的情况下也可以使用。用 SysRq 键获得的信息对于调试来说弥足珍贵。

设置

使用 SysRq 键，要启用内核配置 CONFIG_MAGIC_SYSRQ。

```
# make menuconfig
Kernel hacking --->
 [*] Magic SysRq key
```

RedHat 系列发行版中，默认启用该配置。

启动后可以用 sysctl 设置启用还是禁用。另外，在有的发行版启动时该配置是禁用的，执行下列命令以启用该功能。

```
# sysctl -w kernel.sysrq=1
```

或者

```
# echo 1 > /proc/sys/kernel/sysrq
```

设为 1 后，所有的命令键均可以使用。该值为比特掩码，通过数字的组合来控制 SysRq 键命令，各个值如表 3-1 所示，（）内为命令键。

表 3-1 /proc/sys/kernel/sysrq 的比特掩码

值	允许的命令
2	允许控制控制台日志的级别（0~9）
4	允许控制键盘（kr）
8	允许显示进程等信息（lptwmc）
16	允许 Sync 命令（s）
32	允许以只读方式重新挂载（u）
64	允许发送信号（ei）
128	允许重新启动（b）
256	允许控制实时进程（q）

102

要想允许 Sync（s）和重新挂载（u），禁止其他操作的话，设置方法如下。

```
# echo 48 > /proc/sys/kernel/sysrq
```

该操作限制了控制台输入，而后述的通过/proc/sysrq-trigger 进行的操作就无法通过/proc/sys/kernel/sysrq 进行限制了。

此外，还可以通过内核参数忽略/proc/sys/kernel/sysrq 的设置，使 SysRq 键一直有效。

```
boot> linux sysrq_always_enabled
```

2.6.20 以后的版本支持该参数。

SysRq 键的输入方法

键盘输入的方法是同时按下 Alt 和 SysRq 键的情况下输入命令键。使用串口控制台的话，就在发送 break 信号之后输入命令键。详情参见 **“HACK#16 使用 minicom 进行串口连接”**。SysRq 键有多种，能实现机器控制、输出信息等，而命令键就是指定这些动作的键盘输入。

下面给 proc 文件系统中的/proc/sysrq-trigger 中写入命令键，可以实现与 SysRq 键同样的功能。

```
# echo [命令键] > /proc/sysrq-trigger
```

SysRq 命令键

首先表 3-2 总结了各标准内核[注2]版本支持的命令。

○表示内核支持该命令，命令名表示命令键的内容。例如命令键 b，其命令名为 reBoot，B为大写字母（命令名中的大写字母就是命令键）。

103

表 3-2 各个内核版本的命令键支持情况

命令键	命令名	2.6.9～2.6.11	2.6.12	2.6.13～2.6.15	2.6.16～2.6.19	～2.6.20	2.6.21	2.6.26
0-9	loglevel0-8	○	○	○	○	○	○	○
b	reBoot	○	○	○	○	○	○	○
c	Crashdump[A]			○	○	○	○	○
d	show-all-locks (D)[B]				○	○	○	○
e	tErm	○	○	○	○	○	○	○
f	Full		○	○	○	○	○	○
i	kIll	○	○	○	○	○	○	○
k	saK	○	○	○	○	○	○	○
l	aLlcpus[C]							○
m	showMem	○	○	○	○	○	○	○
n	Nice		○	○	○	○	○	○
p	showPc	○	○	○	○	○	○	○
q	show-all-timers(Q)[D]						○	○
r	unRaw	○	○	○	○	○	○	○
s	Sync	○	○	○	○	○	○	○
t	showTasks	○	○	○	○	○	○	○
u	Unmount	○	○	○	○	○	○	○
w	shoW-blocked-tasks					○	○	○

A. 需要启用 CONFIG_KEXEC。

B. 2.6.17 之前需要启用 CONFIG_DEBUG_MUTEXES，2.6.18～2.6.26 需要启用 CONFIG_LOCKDEP。

C. 多 CPU 环境下要启用 CONFIG_SMP。

D. 需要启用 CONFIG_GENERIC_CLOCKEVENTS。

输入表 3-2 以外的键，则显示以下帮助信息，从中可以查看命令键。

```
SysRq : HELP : loglevel0-8 reBoot Crashdump tErm kIll saK showMem Nice powerOff showPc
show-all-timers(Q) unRaw Sync showTasks Unmount shoW-blocked-tasks
```

注 2： 标准内核（stock kernel）指由 kernel.org 提供的官方内核。大多数发行版都会对内核做修改，标准内核就是相对于这些而言的。——译者注

调试常用的主要命令如表 3-3 所示。

104

表 3-3 SysRq 命令键详细内容

命令键	说明
0-9	设置控制台日志级别。与设置/proc/sys/kernel/printk 相同，但 SysRq 键输出信息时，该级别自动变成 7 或 8，因此无须考虑 SysRq 键的信息显示问题
b	执行重启操作
c	获得 crash dump（kdump）。需要在故意 panic 时使用
d	输出所有已获得的锁。不显示 TASK_RUNNING 状态的进程正在获得的锁，因为锁有可能被立即释放。为使内核能访问所有进程而使用的 tasklist_lock 锁，即使仍在获取，也强制显示
e	给 init（PID 为 1）之外的所有进程发送 SIGTERM
f	启动 OOM Killer（详情参考 **“HACK#56 OOM Killer 的行为和原理”**）。
i	给 init（PID 为 1）之外的所有进程发送 SIGKILL
l	输出系统中所有 CPU 的栈，并显示进程的 backtrace
n	强制将所有实时进程变成普通进程。相当于通过 sched_setscheduler(2) 将任务调度策略设置为 SCHED_NORMAL
m	输出内存和交换文件的状态
p	输出 CPU 寄存器和正在运行的进程的信息。多个 CPU 情况下，只输出处理了键盘终端的 CPU 的信息
q	输出所有正在运行的计时器的信息
s	在所有文件系统上尝试 sync（将内存中的缓存写入磁盘中）。内部强制执行 pdflush 动作，在执行命令键 b 之前执行，可以安全地重新启动
t	输出所有正在运行的进程的栈 backtrace
u	试图将所有文件系统以只读方式重新挂载
w	只输出处于 UNINTERRUPTABLE 的等待状态且忽略各种信号的进程信息

RHEL 的命令键 w 与标准内核不同。RHEL 的命令键 w（showcpus）显示所有 CPU

的栈。该功能与标准内核的命令键 l（aLlcpus）相同。RHEL4/5 的实现情况如表 3-4 所示。

105

表 3-4 RHEL4/5 中的 SysRq 键实现情况

命令键	命令名	RHEL4	RHEL5
0~9	loglevel0~8	○	○
b	reBoot	○	○
c	Crashdump	○	○
d	show-all-locks(D)		○
e	tErm	○	○
f	Full		○
i	kIll	○	○
k	SAK	○	○
m	showMem	○	○
n	Nice		○
p	showPc	○	○
r	unRaw	○	○
s	Sync	○	○
t	showTasks	○	○
u	Unmount	○	○
w	shoWcpus		○

标准内核的 SysRq 键显示示例

下面是命令键 m（showMem）的输出示例，显示的是内存使用量等。

```
SysRq : Show Memory
Mem-info:
Node 0 DMA per-cpu:
...
Active:128436 inactive:87353 dirty:93 writeback:0 unstable:0
 free:6411 slab:30787 mapped:1294 pagetables:435 bounce:0

Swap cache: add 0, delete 0, find 0/0
Free swap  = 1020116kB
Total swap = 1020116kB
261920 pages of RAM
```

```
5716 reserved pages
9262 pages shared
0 pages swap cached
```

下面是命令键 t（showTasks）的输出示例。

106

```
SysRq : Show State
 task                 PC stack  pid father
...
sshd       S ffffffff8048d5e0    0 3121  2908
 ffff81003ec31a28 0000000000000082 0000000000000000 ffff810032c455b8
...
Call Trace:
 [<ffffffff80471eb6>] schedule_timeout+0x1e/0xad
 [<ffffffff8036bc83>] tty_poll+0x5f/0x6d
...
 [<ffffffff802982f4>] sys_select+0xc1/0x183
 [<ffffffff8028c00a>] sys_write+0x45/0x6e
...
```

命令键 w（shoW-blocked-tasks）也可以输出同样的信息。

下面是命令键 l（aLlcpus）的输出示例。这里省略了一部分，实际上输出结果中包含了加载的模块、运行 SysRq 键处理程序的 CPU 的寄存器值、栈、backtrace 等信息。

```
SysRq : Show backtrace of all active CPUs
CPU 0:
Modules linked in: ipmi_watchdog ipmi_devintf ipmi_si ipmi_msghandler
...
Pid: 0, comm: swapper Not tainted 2.6.26 #2
RIP: 0010:[<ffffffff80211d85>] [<ffffffff80211d85>] mwait_idle+0x41/0x44
RSP: 0018:ffffffff8087df60 EFLAGS: 00000246
RAX: 0000000000000000 RBX: 0000000000000000 RCX: 0000000000000000
...
Call Trace:
 [<ffffffff8020ab7b>] ? cpu_idle+0x6d/0x8b

CPU1:
...
```

```
Call Trace:
 <IRQ> [<ffffffff8037c71d>] showacpu+0x0/0x52
 [<ffffffff8037c75d>] showacpu+0x40/0x52
 [<ffffffff8021a118>] smp_call_function_interrupt+0x3b/0x62
 [<ffffffff8020c8d6>] call_function_interrupt+0x66/0x70
 <EOI> [<ffffffffa0049994>] :ext3:ext3_bmap+0x0/0x78
 ...
```

107

命令键 p（showPc）也能输出同样的信息。

命令键 q（show-all-timers）输出的信息与/proc/timer_list 相同。

启用了 Collect scheduler debugging info (CONFIG_SCHEDSTATS)的情况下，命令键 t 和 w 的输出信息中会添加与/proc/sched_debug 相同的信息。

总结

本 hack 讲述了 SysRq 键。每次版本升级，都会添加方便的功能。SysRq 键的实现很简单，所以有什么想用的功能的话，可以向后移植[注3]。如果内核在中断许可的状态下失去响应，可以禁用 watchdog，并复现失去响应的现象。故障出现后使用几次命令键 l 或 p，就能知道失去响应的位置。

使用 diskdump 获取内核崩溃转储

本 hack 讲述 RHEL4 等采用的 diskdump 的使用方法。

diskdump 是 RHEL4 等 RedHat 系列的部分发行版采用的内核崩溃转储（kernel crash dump）功能。这里介绍的部分功能在有的发行版中不能使用，请留意这一点。本 hack 介绍的步骤在 RHEL4.7 上确认过，使用的架构为 x86_64。

理解功能限制

diskdump 为发生 kernel panic 等故障时获取转储文件的功能，因此应当考虑到转储功能本身无法正常运行的情况。其中经常发生的情况就是，在中断处理程序中或中断禁止区域中发生故障，或者是某种自旋锁发生死锁时，试图获取转储文件。diskdump 在转储过程中会禁止中断，使得即使发生这类故障，转储也不会失败。

注3：向后移植（backport），指把软件中的某个改动移植到更低版本上。——译者注

为了 diskdump 在中断禁止状态下依然能进行磁盘的输入输出，磁盘驱动器必须支持 polling I/O（轮询输入输出）。因此，支持 diskdump 的磁盘驱动程序是有限制的，RHEL4.7 中的以下驱动程序支持 diskdump。

aic7xxx
aic79xx
ipr
megaraid
mptfusion
sym53c8xx
sata_promise
ata_piix
CCISS
megaraid_sas
IDE
qla2xxx
lpfc
stex
ips
ibmvscsi
sata_nv
aacraid

108

另外，diskdump 会直接访问磁盘驱动，因此无法使用 LVM，也无法使用在 device mapper 上生成的磁盘分区。

启用崩溃转储

需要给 diskdump 指定转储用的分区。可以准备一个转储专用分区，也可以让它转储到 swap 分区上，但是分区要大于系统中安装的内存大小。本例中没有使用交换分区，而是使用/dev/sda3 作为转储专用分区。给配置文件/etc/sysconfig/diskdump 加入以下内容。

```
DEVICE=/dev/sda3
```

接下来将/dev/sda3 格式化为转储专用分区。

```
# service diskdump initialformat
```

启用 diskdump 服务。

109

```
# chkconfig diskdump on
# service diskdump start
```

利用 service 命令或/proc/diskdump 可以确认 diskdump 是否生效。/proc/diskdump 的内容应当像下面这样。

```
# cat /proc/diskdump
# sample_rate: 8
# block_order: 2
# fallback_on_err: 1
# allow_risky_dumps: 1
# dump_level: 0
# compress: 0
# total_blocks: 98197
#
sda 14329980 2441880
```

此外，设置 sysctl 的变量 kernel.panic，可以在获取转储完成后自动重启。这里通过/etc/sysctl.conf 设置。

```
kernel.panic=10
```

意思是转储结束 10 秒后重启，设置后用 sysctl 启用之。

```
# sysctl -p
```

重启之后，转储文件保存成/var/crash/127.0.0.1-<日期>/vmcore，用 crash 命令可以查看其内容。关于 crash 命令请参见 **“HACK#21 crash 命令的使用方法”**。

使用压缩和部分转储功能缩小转储文件

diskdump 像 kdump 一样可以缩小转储文件。有关 kdump 的内容请参见**“HACK#20 使用 kdump 获取内核崩溃转储”**。

110

压缩功能或部分转储功能可以通过 diskdump 模块的选项来设置。设置 compress 选项为 1 即可启用压缩功能，而部分转储功能则通过 dump_level 选项设置转储级别来实现。表 3-5 为各个转储级别不会转储的页面类型。

表 3-5 不转储的页面类型[注4]

转储级别	缓存页面	带私有页的缓存页面	全零页面	空闲页面	用户页面
0					
1	×	×			
2			×		
3	×	×	×		
4				×	
5	×	×		×	
6			×	×	
7	×	×	×	×	
8					×
9	×	×			×
10			×		×
11	×	×	×		×
12				×	×
13	×	×		×	×
14			×	×	×
15	×	×	×	×	×
17	×				
19	×		×		
21	×			×	
23	×		×	×	
25	×				×
27	×		×		×
29	×			×	×
31	×		×	×	×

注 4：在 diskdump.c 的第 888 行有以下定义，该表即为以下定义的组合。

```
#define DUMP_EXCLUDE_CACHE      0x00000001    /* Exclude LRU &
                                                 SwapCache pages*/
#define DUMP_EXCLUDE_CLEAN      0x00000002    /* Exclude all-zero pages */
#define DUMP_EXCLUDE_FREE       0x00000004    /* Exclude free pages */
#define DUMP_EXCLUDE_ANON       0x00000008    /* Exclude Anon pages */
#define DUMP_SAVE_PRIVATE       0x00000010    /* Save private pages */
```

——译者注

但是，有一些需要注意的事项。要跳过特定页面，就需要搜索内核内部的内存管理链表。假如故障是由于该链表被破坏而引起的，那么 diskdump 搜索该链表时就会再次 panic，或是停止响应。使用该功能的推荐级别为 19，因为 19 无须搜索链表。在/etc/modprobe.conf 中给 diskdump 模块指定选项。

111

```
options diskdump dump_level=19 compress=1
```

启用该设置需要重启 diskdump。

```
# service diskdump restart
```

查看/proc/diskdump，看看设置的选项是否正确加载了。

```
# cat /proc/diskdump
# sample_rate: 8
# block_order: 2
# fallback_on_err: 1
# allow_risky_dumps: 1
# dump_level: 19
# compress: 1
# total_blocks: 98197
#
sda 14329980 2441880
```

发生故障时邮件通知

diskdump 能够在将 vmcore 文件保存到/var/crash/下之后执行用户设置的脚本。利用该功能，可以在发生转储时（即发生系统故障时）发送邮件通知。/usr/share/doc/diskdumputils-<版本号>/example_scripts/下面有例子，可以尝试一下，本次利用的是 diskdump-success。将该文件复制到/var/crash/scripts/下，并编辑成下面这样。

```
# cat /var/crash/scripts/diskdump-success
#!/bin/sh

ADDRESS=tabe@miraclelinux.com

mail -s "[diskdump] `hostname` crashed" $ADDRESS <<_EOF
The machine `hostname` crashed.
```

112

```
Writing crash dump to $1
_EOF

# savecore always returns 0 whatever the result of this script because this is
# called after a dump file is created.

exit 0
```

转储输出目的设备的冗余化

开头的“理解功能限制”（107 页）也提到，diskdump 不通过文件系统直接访问磁盘驱动器。因此，如果转储分区所在的设备驱动发生故障，就可能无法获取转储。这时 diskdump 将多个分区设置为转储分区，像下面这样设置为/etc/sysconfig/diskdump 即可。

```
DEVICE=/dev/sda3:/dev/hda
```

/dev/sda3 是之前使用的转储专用分区，笔者的环境中其驱动程序为 mptfusion。另一方面，/dev/hda 为使用 IDE 驱动的其他分区。假如 mptfusion 驱动程序发生故障，就可能无法将转储写入/dev/sda3。这种情况下 diskdump 会转储至下一个/dev/hda 中，它是 IDE 驱动的，可以正常地写入转储。

总结

在 kdump 被合并到主线之前，出现了各种各样的转储功能，各个发行版采用的也不尽相同，其中包括 netdump、LKCD、mkdump，还有本 hack 介绍的 diskdump。本 hack 选择了使用者较多的 diskdump 来介绍内核崩溃转储的获取方法。

参考

这里未能详细介绍的设置请参见 diskdumputils 的 README。

/usr/share/doc/diskdumputils-<version>/README

——安部东洋

113

使用 kdump 获取内核崩溃转储

本 hack 说明最近的发行版采用的 kdump。

kdump 是 linux-2.6.13 之后合并进主线的内核崩溃转储功能。内核版本在 2.6.13 之后的发行版中均可以使用 kdump。本 hack 的内容以 RHEL5.1 为准，所用的架构为 x86_64。

启用崩溃转储

首先给内核启动参数中添加 crashkernel=128M@16M，RHEL5.1 中可以按照下面这样编辑/etc/grub.conf。

```
title Red Hat Enterprise Linux Server (2.6.18-53.1.21.el5)
    root (hd0, 0)
    kernel /boot/vmlinuz-2.6.18-53.1.21.el5 ro root=LABEL=/1 crashkernel=128M@16M rhgb quiet
    initrd /boot/initrd-2.6.18-53.1.21.el5.img
```

含义为将物理地址 0x1000000 开始的 128MB 内存保留用于崩溃转储。编辑完 grub.conf 之后重新启动，使该设置生效。

接下来利用 chkconfig 命令和 service 命令启用 kdump 服务。

```
# chkconfig kdump on
# service kdump start
```

可以通过 service 命令或/sys/kernel/kexec_crash_loaded 来确认设置是否成功。

使用 service 命令显示下面的内容即表示设置已生效。

```
# service kdump status
kdump is operational
```

kexec_crash_loaded 内容为 1 则表示设置已生效。

```
# cat /sys/kernel/kexec_crash_loaded
1
```

114

设置生效后，试着获取一下崩溃转储。

```
# echo c > /proc/sysrq-trigger
```

转储成功后，/var/crash/下就会生成目录，以及名为 vmcore 的文件，使用 crash 命令即可查看该文件。关于 crash 命令请参见“HACK#21 crash 命令的使用方法”。

利用 makedumpfile 缩小转储文件

之前设置生成的崩溃转储文件与安装的内存容量大小相同，8GB 内存的话转储文件也有 8GB。但是，kdump 也像 diskdump 那样可以压缩转储镜像，使之变得更小。相应的工具为 kexec-tools 中包含的 makedumpfile 命令。要使用该命令，需要在/etc/kdump.conf 中加入 core_collector 的设置。

```
ext3 /dev/sda5
core_controller makedumpfile -c
```

开头的“ext3 /dev/sda5”要设置为根文件系统所在的设备，转储文件输出到该分区的./var/crash 下。如果想把转储输出到其他分区，例如改为挂载到/dump 目录下的/dev/sda6 下时，可以这样写。

```
ext3 /dev/sda6
path .
```

这样就会在/dump 目录下生成日期目录，并将转储文件放在其中。

-c 为压缩选项。此外，还有设置转储级别的-d 选项，可以设置崩溃转储中不包含哪些页面（内存）类型。表 3-6 为各个转储级别跳过的页面类型。

115

表 3-6 跳过的页面类型

转储级别	全零页面	缓存页面	带私有页面的缓存页面	用户数据	空闲页面
0					
1	×				
2		×			
4		×	×		
8				×	
16					×
31	×	×	×	×	×

转储级别用表 3-6 中的数字之和来设置。例如，不转储全零页面和空闲页面，则设置转储级别为 1+16=17。笔者使用下述设置。

```
core_collector makedumpfile -c -d 17
```

接下来安装带有调试信息的内核。RHEL5.1 为 kernel-debuginfo 和 kernel-debuginfo-common 包。

```
# rpm -ivh kernel-debuginfo-2.6.18-53.1.21.el5.x86_64.rpm \
  kernel-debuginfo-common-2.6.18-53.1.21.el5.x86_64.rpm
```

重新启动 kdump 服务。

```
# service kdump restart
```

我们使用刚才讲过的方法来获取崩溃转储。获取的转储文件是否压缩，只需看看 vmcore 文件的大小便知。压缩后的转储文件不是 ELF 格式，无法使用 GDB 调试，请使用 crash 命令查看。

如果想用 makedumpfile 获取自己编译的内核的崩溃转储，需要将带有调试信息的内核配置到下面的地方。

```
/usr/lib/debug/lib/modules/`uname -r`/vmlinux
```

116

Fedora9 等使用了最新内核的发行版，不再需要带有调试信息的内核包，这是因为内核中增加了名为 vmcoreinfo 的功能，同时 kexec-tools 包也发生了变化。如果所用的系统内核和 kexec-tools 都支持 vmcoreinfo，就没有必要安装带有调试信息的内核了。2008 年 9 月 16 日的 RHEL5.1 版本（kexec-tools-1.102pre-21.el5-x86_64.rpm）已支持该功能。

将崩溃转储发送到远程服务器

可以在/etc/kdump.conf 中添加 net 设置。

```
/* 通过NFS挂载来发送 */
net <服务器名或IP地址>:<导出的目录名>
/* 通过SSH发送*/
net <用户名>@<服务器名或IP地址>
```

使用 NFS 的话，需要事先在导出的目录下建立./var/crash 目录。SSH 的话，转储会发送到远程服务器的/var/crash 目录下。因此，这里设置的登录用户必须拥有对/var/crash 的写入权限。出于安全考虑而不想这样做的话，可以另行准备转储专用目录，然后像下面这样改变转储目的地。

```
path /dump
```

SSH 下 path 要设置成绝对路径，而不是相对路径。此外，为了能无密码登录，需要事先设置好公钥。利用 kdump 的 init 脚本中的 propagate 选项可以执行该操作。

```
# service kdump propagate
```

link_delay 为从开始连接网卡到发送之间的等待时间，单位为秒。为以防万一，笔者把它也加上了。

```
link_delay 10
```

117

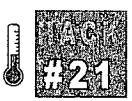
如果要将 makedumpfile 和 SSH 配合使用，就必须对转储文件进行转换。输出到远程服务器上的转储文件名为 vmcore.flat，crash 命令无法直接读取，使用下面的命令可转换文件格式。

```
# makedumpfile -R vmcore < vmcore.flat
```

总结

本 hack 说明了使用 kdump 获取内核崩溃转储的方法。panic 发生时，kdump 可以跳过磁盘控制器等设备的结束处理，利用 kexec 启动转储内核。因此，在 panic 时的某些设备状态下可能会无法启动转储内核。笔者曾遇到过磁盘 I/O 中发生 panic，转储内核对磁盘控制器驱动的初始化失败，导致无法获取转储的事情。当时我在驱动程序初始化时添加了重置磁盘控制器的处理，绕过了这个问题。

——安部东洋



crash 命令的使用方法

介绍 crash 这个方便的命令使用方法。

crash 有各种各样的命令。用好该命令，就能轻易获得想要的信息。

启动 crash 之后会显示提示符，可以用对话的方式进行操作。本 hack 介绍 crash 中很方便的命令，以及有利于调试的东西。

命令输出结果以内核版本 2.6.18 为例。

crash 的功能包括查看 vmcore 这种崩溃转储文件，以及实时查看系统。查看实时

系统的功能由于不是查看崩溃转储文件，而是查看运行中的内核，所以有功能限制，后面会具体说明这一点。

启动 crash

启动 crash 命令查看崩溃转储文件的命令如下。

```
# crash vmlinux vmcore
```

vmlinux 是未压缩的内核镜像。RedHat 系列的发行版中，安装名为 kernel-debuginfo
118 的 RPM 包之后，vmlinux 位于/usr/lib/debug/lib/modules/[版本号]/下。vmcore 是用转储功能获得的崩溃转储文件。

用 crash 命令查看实时系统的命令如下。

```
# crash vmlinux
```

vmlinux 指定为运行中的内核的 vmlinux 文件。

工具命令

首先介绍 crash 中的工具命令。

set 命令

set 命令应用广泛，可以指定进程并显示其进程上下文。如果不指定，就显示 kernel panic 时正在运行的进程。

```
crash> set
    PID: 4525
    COMMAND:"umount"
    TASK: 101040df7f0 [THREAD_INFO: 1009face000]
    CPU: 1
  STATE: TASK_RUNNING (PANIC)
```

通过-c选项可以指定CPU，以显示在该CPU上运行的进程。-p选项可以显示kernel panic 时运行的进程。

set 命令还可以设置 crash 的环境。以下命令可以改变 crash 的编辑器。

```
crash> set vi
```

```
crash> set emacs
```

或者用下述格式在 crash 启动时设置。

```
# crash -e [vi | emacs] ...
```

设置好编辑器后，crash 的命令输入界面的键盘绑定就会变成 vi 风格或 emacs 风格了。设置成 emacs，其键盘绑定与 bash 的相同。

119

有时 crash 会输出大量信息。下述命令可以禁用滚动。

```
crash> set scroll off
```

也可以用下面的命令。

```
crash> sf
```

sf 是个别名，是 set scroll off 的简写。默认的别名可以用 alias 命令查看。

```
crash> alias
ORIGIN      ALIAS     COMMAND
builtin     man       help
builtin     ?         help
builtin     quit      q
builtin     sf        set scroll off
builtin     sn        set scroll on
builtin     hex       set radix 16
builtin     dec       set radix 10
builtin     g         gdb
builtin     px        p -x
builtin     pd        p -d
builtin     for       foreach
builtin     size      *
builtin     dmesg     log
builtin     last      ps -l
```

hex、eval 命令

关于 hex、eval 命令请参考 **“HACK#14 从汇编语言查找相应的源代码”**。

ascii 命令

ascii 命令可以将十六进制数转换成字符串。

```
crash> rd modprobe_path 2
ffffffff813213a0: 6f6d2f6e6962732f 000065626f727064  /sbin/modprobe..
crash> ascii 6f6d2f6e6962732f
6f6d2f6e6962732f: /sbin/mo
```

120

h 命令

h 命令可以显示输入过的命令历史。

```
crash> h
[1] set
[2] set vi
[3] set scroll off
[4] sf
[5] alias
[6] hex
[7] eval
```

查看内核信息的命令

下面讲一下查看内核内部信息的命令。

bt 命令

bt 命令可以输出进程的 backtrace，使用频率相当高。下面的命令可以显示所有进程的 backtrace，十分方便。

```
crash> foreach bt -tf
```

-t 选项表示显示栈中的所有文本符号（text symbol）。

```
crash> bt 2157
PID: 2157  TASK: ffff81007e095040 CPU: 1  COMMAND: "syslogd"
 #0 [ffff810075a5f938] schedule at ffffffff80061f29
 #1 [ffff810075a5fa20] schedule_timeout at ffffffff800627cd
 #2 [ffff810075a5fa70] do_select at ffffffff8001137f
 #3 [ffff810075a5fc10] journal_stop at ffffffff880327ae
...
crash> bt -t 2157
PID: 2157  TASK: ffff81007e095040 CPU: 1  COMMAND: "syslogd"
          START: thread_return (schedule) at ffffffff80061f29
```

```
 [ffff810075a5fa20] schedule_timeout at ffffffff800627cd
 [ffff810075a5fa40] add_wait_queue at ffffffff800477f9
 [ffff810075a5fa70] do_select at ffffffff8001137f
 [ffff810075a5fb08] __pollwait at ffffffff8001e2cf
 [ffff810075a5fb38] default_wake_function at ffffffff80089830
 [ffff810075a5fc10] journal_stop at ffffffff880327ae
...
```

121

-f选项能显示栈帧内的所有栈数据。该选项可以方便地查看函数参数。

-l选项显示文件名和行号。

```
crash> bt -l 2157
PID: 2157  TASK: ffff81007e095040  CPU: 1  COMMAND: "syslogd"
 #0 [ffff810075a5f938] schedule at ffffffff80061f29
    /usr/src/debug/kernel-2.6.18/linux-2.6.18.x86_64/kernel/sched.c: 1840
 #1 [ffff810075a5fa20] schedule_timeout at ffffffff800627cd
    /usr/src/debug/kernel-2.6.18/linux-2.6.18.x86_64/kernel/timer.c: 1543
 #2 [ffff810075a5fa70] do_select at ffffffff8001137f
    /usr/src/debug/kernel-2.6.18/linux-2.6.18.x86_64/fs/select.c: 288
 #3 [ffff810075a5fc10] journal_stop at ffffffff880327ae
 #4 [ffff810075a5fc80] __generic_file_aio_write_nolock at ffffffff80015e21
...
```

-a选项只显示当前进程。但是在运行的系统中无法显示当前进程。

```
crash> bt -a
bt: -a option not supported on a live system
```

可以用 task 命令获取栈指针，然后用 rd -s 命令查看，可以在运行中的系统中获得近似于当前进程的 backtrace 的信息。

```
crash> task | grep rsp
   rsp0 = 0xffff81004c57a000,
   rsp = 0xffff81004c579938,
   userrsp = 0x7fffd81bcb18,
crash> rd 0xffff81004c579938 -e 0xffff81004c57a000 -s
...
ffff81004c5799d8:  ffff81004c579a18 __wake_up+0x38
ffff81004c5799e8:  ffff81007c578000 ffff81007c578018
ffff81004c5799f8:  ffff81007c578018 0000000000000145
```

```
ffff81004c579a08: ffff81004c579b60 remove_wait_queue+0x1c
ffff81004c579a18: ffff81004c579b58 0000000000000000
ffff81004c579a28: 0000000000000296 free_poll_entry+0x11
ffff81004c579a38: ffff81004c579b90 poll_freewait+0x29
ffff81004c579a48: 0000000000000008 0000000000000001
ffff81004c579a58: ffff81007faf9280 0000000000000001
ffff81004c579a68: 0000000000000001 do_select+0x445
ffff81004c579a78: ffff81000237f880 ffff81004c579f50
ffff81004c579a88: 000000017ff50030 ffff81004c579dd8
...
ffff81004c579f68: 00007fffd81bd920 00007fffd81be401
ffff81004c579f78: 00007fffd81be3a0 system_call+0x7e
ffff81004c579f88: 0000000000000246 0000000000000001
ffff81004c579f98: 0000000000000001 000000004c579fa0
...
```

122

dev 命令

dev 命令可以显示字符设备列表。-p 选项显示 PCI 数据，其内容与 lspci 命令相同。-i 选项可以显示 I/O 端口和 I/O 设备内存映射，与下面的命令基本上相同。

```
# cat /proc/ioports
# cat /proc/iomem
```

dis 命令

dis 命令可以输出反汇编。详情参见 **“HACK#40 实时进程停止响应”**。

files 命令

files 命令可以显示进程打开的文件。**“HACK#39 内核停止响应（信号量篇）”** 中用到了该命令，详情请参考这个 HACK。

irq 命令

irq 命令显示内核内部管理的中断信息。

kmem 命令

显示有关内核内存的信息。-s 选项可以显示 slab 缓存信息，与/proc/slabinfo 的信息相同。

```
crash> kmem -s
CACHE           NAME           OBJSIZE  ALLOCATED  TOTAL  SLABS  SSIZE
ffff81007c5c6300 ip_fib_alias   64       21    59     1    4k
ffff81007b51f2c0 ip_fib_hash    64       18    59     1    4k
ffff81007ac50280 fib6_nodes     64       35    59     1    4k
ffff81007ac51240 ip6_dst_cache  320      29    48     4    4k
...
```

123

-i选项显示内存信息，与free命令相同。

```
crash> kmem -i
                PAGES        TOTAL      PERCENTAGE
 TOTAL MEM      514976       2 GB       ----
      FREE      421511       1.6 GB     81% of TOTAL MEM
      USED      93465        365.1 MB   18% of TOTAL MEM
    SHARED      0            0          0% of TOTAL MEM
   BUFFERS      3906         15.3 MB    0% of TOTAL MEM
    CACHED      53504        209 MB     10% of TOTAL MEM
      SLAB      3709         14.5 MB    0% of TOTAL MEM
...
 TOTAL SWAP     512069       2 GB       ----
 SWAP USED      0            0          0% of TOTAL SWAP
 SWAP FREE      512069       2 GB       100% of TOTAL SWAP
...
crash>
```

-p选项显示内存映射（memory map），还可以指定地址。[]括起来的部分是尚未释放的内存。

```
crash> kmem ffff81004fd64048
CACHE          NAME             OBJSIZE  ALLOCATED    TOTAL  SLABS  SSIZE
ffff81007f6c2380 ext3_inode_cache      760     73290    73295  14659    4k
SLAB           MEMORY           TOTAL  ALLOCATED  FREE
ffff81004fd64000  ffff81004fd64048     5       5    0
FREE / [ALLOCATED]
  [ffff81004fd64048]

      PAGE      PHYSICAL      MAPPING       INDEX CNT FLAGS
ffff8100018b7de0 4fd64000     -------       -----  1 48080000000080
```

访问某段内存产生 kernel panic 的话，用这种方法就能看出，那段内存实际上已被释放了。

list 命令

list 命令可以遍历 list_head 结构，按照顺序显示地址。

124

```
crash> whatis modules
struct list_head modules;
crash> list modules
ffffffff812cd420
ffffffff88302c08
...
ffffffff88017d88
ffffffff88009a88
crash> list modules | wc -l
38
crash>
```

可以从中看出 modules 链表连接着 38 项。

下面是结构中包含 list_head 结构成员的例子。这种情况下，可以用-o 选项指定偏移量，可以用同样的方式遍历链表。

```
crash> whatis ip_packet_type
struct packet_type ip_packet_type;
crash> struct -o packet_type
struct packet_type {
  [0x0] __be16 type;
  [0x8] struct net_device *dev;
 [0x10] int (*func)(struct sk_buff *, struct net_device *, struct packet_type *, struct
net_device *);
       struct sk_buff *(*gso_segment)(struct sk_buff *, int);
 [0x20] int (*gso_send_check)(struct sk_buff *);
 [0x28] void *af_packet_priv;
 [0x30] struct list_head list;        /* 出现了list_head */
}
SIZE: 0x40
crash>
```

可见 list_head 结构的偏移量为 0x30，下面用-o 选项指定。

```
crash> list -o 0x30 ip_packet_type
ffffffff813288a0
ffffffff814188a0
ffffffff814188d0
ffffffff81327730
```

125

可见 ip_packet_type 的 list 成员中连接着 4 项。

加上-s 选项，可以同时显示链表项的成员。下面的例子遍历链表，并进一步显示各项的 func 成员。

```
crash> list -o 0x30 ip_packet_type -s packet_type.func
ffffffff813288a0
  func = 0xffffffff810353e5 <ip_rcv>,
ffffffff814188a0
  func = 0xffffffff813052f0 <llc_tr_packet_type+48>,
ffffffff814188d0
  func = 0xffffffff813052b0 <llc_packet_type+48>,
ffffffff81327730
  func = 0xffffffff812275f3 <arp_netdev_event>,
```

mod 命令

mod 命令可以加载模块信息、符号信息和调试信息等。详情参见 **“HACK#14 从汇编语言查找相应的源代码”**、**“HACK#33 kernel panic（空指针引用篇）”** 等。

net 命令

net 命令可以列出网络设备。它能显示 net_device 结构的地址，据此可以进行进一步调查。

```
crash> net
    NET_DEVICE     NAME  IP ADDRESS(ES)
ffffffff8030f680  lo    127.0.0.1
ffff81007ea24000  eth2  192.168.0.155
ffff81007f7b8000  eth3  192.168.0.156
ffff81007ebbd000  eth0  172.16.0.153
ffff81007e1ff000  eth1
ffff81007d50a000  sit0
crash> struct net_device ffff81007ea24000
struct net_device {
```

```
  name = "eth2\00045090668\000\000",
  name_hlist = {
    next = 0x0,
    pprev = 0xffffffff804bae90
  },
...
```

126

ps 命令

ps 命令显示进程信息。

-a 选项可以显示命令行参数和环境变量。

```
crash> ps -a syslogd
PID: 2157  TASK: ffff81007e095040  CPU: 0  COMMAND: "syslogd"
ARG: syslogd -m 0
ENV: CONSOLE=/dev/console
    SELINUX_INIT=YES
    TERM=linux
    INIT_VERSION=sysvinit-2.86
    PATH=/sbin:/usr/sbin:/bin:/usr/bin
    runlevel=3
    RUNLEVEL=3
    PWD=/
    LANG=en_US.UTF-8
    previous=N
    PREVLEVEL=N
    SHLVL=3
    HOME=/
    _=/sbin/syslogd
```

-t 选项可以显示进程的运行时间、开始时间，以及在用户空间和内核空间的运行时间。详情参见 **“HACK#40 实时进程停止响应”**。

rd 命令

rd 命令可以直接读取内存地址。使用范例请参见本 hack 的 bt、wr 命令。

runq 命令

runq 命令显示进程调度的运行队列。

sig 命令

sig 命令显示进程的信号处理程序（signal handler），以及等待（pending）的信号的信息。-l 选项等同于 kill -l，显示已定义的信号编号。

127

struct 命令

struct 命令可以显示结构的定义，并结合显示实际地址中的数据。

```
crash> struct timespec xtime
struct timespec {
  tv_sec = 0x492ae39c,
  tc_nsec = 0x25218473
}
```

其他示例请参见[HACK#14]。

swap 命令

swap 命令显示各个交换设备的大小等信息。基本上等同于 swapon -s。

sym 命令

sym 命令用于解析符号（symbol）。sym -l 等同于 Systemp.map。

sys 命令

sys 命令显示系统信息，如时间、CPU 的平均负载、表示 kernel panic 原因的消息等。同 crash 启动时显示的信息。

如果在内核配置中启用了 CONFIG_IKCONFIG，那么执行 sys config 即可显示内核配置列表。内容同 zcat /proc/config.gz。

```
crash> sys config
#
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.18
# Tue Dec 16 22:33:31 2008
#
CONFIG_X86_64=y
CONFIG_64BIT=y
CONFIG_X86=y
```

```
CONFIG_LOCKDEP_SUPPORT=y
CONFIG_STACKTRACE_SUPPORT=y
128 CONFIG_SEMAPHORE_SLEEPERS=y
CONFIG_MMU=y
...
```

使用 sys -panic 可以故意发生 kernel panic，其效果同 echo c > /proc/sysrq- trigger。

task 命令

task 命令可以显示 task_struct 结构，详情参见[HACK#40]等。

timer 命令

timer 命令显示定时器队列中的项目。

whatis 命令

whatis 命令显示符号等结构的定义。下面是全局变量 modules 的显示范例，可见它是 list_head 结构的变量。

```
crash> whatis modules
struct list_head modules;
```

wr 命令

wr 命令可以更改内存的内容。下面的例子用 crash 更改了运行系统中表示时间的变量 jiffies。修改后，表示系统启动后运行时间的 UPTIME 增加了 3 天多。

```
crash> sys
     KERNEL: /boot/vmlinux-2.6.18
   DUMPFILE:  /dev/mem
       CPUS:  2
       DATE:  Fri Dec 19 20:26:07 2008
     UPTIME:  00:00:45
...
crash> rd jiffies_64
ffffffff81457200:  00000000fffc6751             Qg......
crash> wr jiffies_64 10ffc7651
crash> rd jiffies_64
129 ffffffff81457200:  000000010ffc88c0             ........
crash> sys
```

```
    KERNEL: /boot/vmlinux-2.6.18
  DUMPFILE: /dev/mem
      CPUS: 2
      DATE: Fri Dec 19 20:26:56 2008          /* 仅仅执行了命令50秒*/
    UPTIME: 3 days, 02:35:10                  /* UPTIME增加了3天多 */
...
```

crash 的启动选项

下面介绍 crash 启动时的一些很方便的选项。

-i 选项

把要输入 crash 的命令写到文件中，并用-i 选项指定该文件，即可将命令自动输入 crash。利用该选项可以让 crash 自动化。下面是执行 help 命令，然后再执行 exit 命令退出 crash 的例子。

```
[bash]# cat crash_cmd.txt
help
exit
[bash]# crash -s -i crash_cmd.txt
*               files       mod             runq            union
alias           foreach     mount           search          vm
ascii           fuser       net             set             vtop
bt              gdb         p               sig             waitq
btop            help        ps              struct          whatis
dev             irq         pte             swap            wr
dis             kmem        ptob            sym             q
eval            list        ptov            sys
exit            log         rd              task
extend          mach        repeat          timer

crash version: 4.0-7.4  gdb version: 6.1
For help on any command above, enter "help <command>".
For help on input options, enter "help input".
For help on output options, enter "help output".

[bash]#             /* 显示帮助之后返回bash */
```

130

-s 选项

如上例所示，指定-s 选项可进入安静模式，crash 启动时不再显示多余的信息。

crash 的初始化文件

在.crashrc 文件中写好命令，crash 启动时就会执行这些命令。.crashrc 应当放在主目录或者当前目录下，可用来写一些默认设置，十分方便，如 set 命令、alias 命令等。

总结

本 hack 介绍了 crash 命令。熟悉该命令可以进行有效的分析。

参考文献

- White Paper: Red Hat Crash Utility
  http://people.redhat.com/anderson/crash_whitepaper/

——大岩尚宏

死机时利用 IPMI watchdog timer 获取崩溃转储

讲述中高端服务器安装的 IPMI watchdog timer 的配置方法，以备死机时进行调试。

什么是 IPMI watchdog timer

IPMI watchdog timer 是个遵循 Intelligent Platform Management Interface 规格的 watchdog timer。IPMI 是 Intel 等几家计算机相关生产商创立的规格，规定了计算机各个部件的温度、电压、风扇等状态的获取方法，以及控制电源的接口等。这些规格也包含在本 hack 讲述的 watchdog timer（下文简写为 WDT）中。IPMI WDT 独立于 CPU，采用了专用硬件，因此系统死机时基本上可以保证执行重启等处理。可以说，watchdog timer 与其他不需要特殊硬件的 softdog、如 Intel 等部分处理器中可用的 NMI watchdog（参见[HACK#23]）相比，其可靠性更高。但是，该功能并非所有 PC 和服务器都有。一般而言，中高端服务器安装该功能的较多。

131

IPMI WDT 的本来目的是提高系统可用性，但与调试也有紧密关系。这是因为，一旦 WDT 执行了重启，就说明系统死机了。死机原因有软件 bug，也有硬件故障，但不论如何，我们需要某些信息才能寻找故障原因。在 Linux 中，利用 IPMI WDT 的功能，可以在系统死机后、重启之前获得崩溃转储（crash dump）。当然，只有在内核的转储获取功能仍然能正常工作的情况下才有用，但即使如此也非常有用了。**“HACK#36 内核停止响应（死循环篇）”**和**“HACK#40 实时进程停止响应”**介绍了利用 IPMI WDT 获取崩溃转储进行调试的例子。

在 Linux 中使用 IPMI

要在 Linux 中使用 IPMI，需要在内核配置中启用 IPMI，还要用到 ipmitools、FreeIPMI 等客户端程序。关于内核配置，在 RHEL 等的大多数发行版中，IPMI 是默认启用的，并以模块的方式提供。如果使用的发行版的默认值为禁用，或者希望由模块方式改为内嵌方式，可以执行 make menuconfig，启用 Device Drivers→Character devices →IPMI top-level message handler 里面的 IPMI Watchdog Timer 项。

ipmi_watchdog 模块

下面介绍一下当 IPMI 的内核功能作为模块提供的情况。模块名为 ipmi_watchdog.ko。该模块的主要参数如表 3-7 所示，各个参数的含义如表 3-8 至表 3-10 所示。另外，timeout、pretimeout、action、preaction 的关系如图 3-4 所示。

表 3-7 ipmi_watchdog 模块的主要参数

参数	说明
timeout	超时时间（秒）
action	超时时的操作（reset、none、power_cycle、power_off）
pretimeout	超时操作执行和预超时（pretimeout）操作执行之间的时间差（秒）
preaction	预超时时的操作（pre_none、pre_smi、pre_nmi、pre_int）
preop	预超时时的驱动程序操作（preop_none、preop_panic）

132

表 3-8 action 的各参数设置时的行为

参数	说明
none	什么都不做
reset	重启系统
power_cycle	关闭电源再打开
power_off	关闭电源

表 3-9 preaction 的各参数设置时的行为

参数	说明
pre_none	什么都不做
pre_smi	将信息通知 IPMI 驱动程序
pre_int	使用中断将信息通知 IPMI 启动程序
pre_nmi	引发 NMI 中断

表 3-10 preop 的各参数的含义

参数	说明
preop_none	什么都不做
preop_panic	引发 kernel panic

要获取转储，应当设置 pretimeout 为小于 timeout 的值，并将 preaction 设置为 pre_nmi 或 pre_int。例如，将 timeout 设置为 90，pretimeout 设置为 30，这样最后一次 watchdog timer 更新之后 60 秒之内如果不发生更新（即系统死机了），就执行 preaction 指定的动作。设置为 pre_nmi 的话，那么在超出 pretimeout 时间之外发生死机的话，主板上就会产生 NMI 信号报警。NMI 报警后，一般会启动内核的 NMI 处理程序，

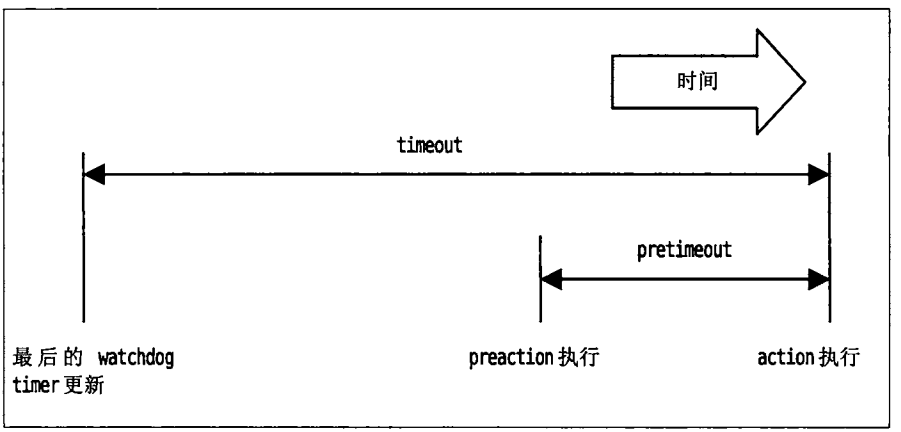


图 3-4 模块参数和执行的操作之间的关系

133

引发 panic。如果设置了 kdump（参见[HACK#20]），就能获取转储。另外，将 preaction 设置为 pre_int、preop 设置为 preop_panic，也能获得转储。此时，超过预超时时间的信息会通知给 ipmi 驱动程序，然后驱动程序执行 preop 指定的动作 preop_panic，引发 panic。

当然，这些参数可以在通过命令行加载模块时直接设置，也可以写到

/etc/modprobe.conf 中。RHEL5 中，在/etc/sysconfig/ipmi 中写入以下配置，打开 ipmi 服务，就能在启动时自动加载该模块。

```
IPMI_WATCHDOG=yes
IPMI_WATCHDOG_OPTIONS="timeout=90 action=reset pretimeout=30 preaction=pre_init preop=
preop_panic"
```

此外，设置参数时需要注意两点。第 1 点，如果内核在执行崩溃转储的过程中死机，那么当然不会执行 pretimeout 的动作，而超过 timeout 设置的时间之后，就会执行 action 指定的动作（硬件重启等）。第 2 点，即使 pretimeout 执行了，如果接下来又发生死机，那么在超过 timeout 设置的时间之后，就会执行 action 的动作。使用 kdump 的话，转储会在第 2 内核上执行，因此没有问题，但 preaction 引发 kernel panic 后利用 diskdump 等工具的话，在死机的第 1 内核上执行崩溃转储，就会发生问题。例如，安装的内存较多，在 timeout 时间内无法完成转储，就会在获取崩溃转储的过程中执行重启等动作。

/dev/watchdog 接口

加载 ipmi_watchdog.ko 之后，就能通过/dev/watchdog 控制 IPMI WDT。/dev/watchdog 提供 Linux 的 watchdog 标准控制功能，除了 IPMI watchdog 之外，还能用于像 softdog、厂商自带的硬件 WDT 的控制。但是，该接口为互斥的，比如不能同时使用 softdog 和 IPMI WDT。

有多个程序可以控制/dev/watchdog，都包含在 ipmitools 等中。另外，内核代码树中的 Documentation/watchdog/src/watchdog-simple.c 也是控制程序之一。下面是简化后的代码。

```
$ cat watchdog-very-simple.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
      int fd = open("/dev/watchdog", O_WRONLY);
      int ret = 0;
      if (fd == -1) {
```

134

```
            perror("watchdog");
            exit(EXIT_FAILURE);
        }
        while (1) {
            if (write(fd, "\0", 1) != 1) {
                ret = -1;
                break;
            }
            sleep(10);
        }
        close(fd);
        return ret;
}
```

输出崩溃转储

这里利用上述 watchdog-very-simple 确认死机时是否能输出崩溃转储。但是，实际让系统死机是很困难的，所以采取如下做法。

```
# modprobe ipmi_si
# modprobe impi_watchdog timeout=90 action=reset pretimeout=30 preaction=pre_int
preop=preop_panic
# watchdog-very-simple
Ctrl-Z
#
```

该方法停止 watchdog-very-simple 进程，模拟死机的发生。该操作发生之后，就会根据内核模块的 pre_timeout 设置，执行崩溃转储。

另外，开头的 ipmi_si 是与主板上安装的 IPMI 硬件通信的模块，如果未安装 IPMI 硬件，该模块就会加载失败。

135

参考文献

Linux 内核的 Documentation/IPMI.txt

总结

本 hack 讲述了使用 IPMI watchdog timer 在死机时获取崩溃转储的方法。

用 NMI watchdog 在死机时获取崩溃转储

本 hack 讲述利用 x86_64、i386 架构上的 NMI watchdog 在系统死机时获取崩溃转储的方法。

什么是 NMI watchdog

NMI 是 Non Maskable Interrupt 的缩写，意为不可屏蔽中断。这种中断本来用于将内存校验错误等系统致命错误通知 CPU，但是最近的 APIC [注5] 能够定时产生这种中断。NMI watchdog 是利用该功能实现的 watchdog 定时器。通过 NMI watchdog，Linux 内核可以检测系统死机，并执行 panic、获取崩溃转储。

内核利用 NMI watchdog 检测死机的原理如下。通常定时器每秒钟会产生一定次数的中断，该次数由内核配置设置（100～1000 次）。但是，如果在禁止中断的情况下陷入死循环或死锁状态，定时器处理就无法执行。相反，NMI 即使在这种状态下也能产生，CPU 就能执行 NMI 处理程序。NMI 处理程序监视定时器中断是否被执行，如果超过一定时间没有执行（各个发行版不同，通常为 5～30 秒左右），就认为发生了死机。

检查能否使用 NMI watchdog

现在的大多数机器都能使用该功能，可以用下述方法检查是否能使用。

136

```
# cat /proc/interrupts
           CPU0       CPU1
...
NMI:  598499293  598499235
...

 (2~3秒之后)
# cat /proc/interrupts
           CPU0       CPU1
...
```

注 5：APIC（Advanced Programmable Interrput Controllers），高级可编程中断控制器。——译者注

```
NMI: 598502039 598501981
...
```

如果 NMI 增加，说明 NMI watchdog 可用。如果 NMI watchdog 不可用，那么 NMI 就只有在致命错误发生时才产生，NMI 数字大部分情况下为 0。

在 NMI watchdog 超时时获取崩溃转储

用上述方法确认 NMI watchdog 可用后，可以在内核选项中加入以下设置。

```
nmi_watchdog=panic,N （N为1或2）
```

在安装了 IO-APIC 的设备上将 N 设置为 1（I/O-APIC 模式），没有 IO-APIC 的设备上将 N 设置为 2（local APIC 模式）。这样设置之后，NMI watchdog 在超时时就会产生 kernel panic，然后设置 kdump（[HACK#20]）、diskdump（[HACK#19]）使之获取崩溃转储即可。另外，有些内核版本上，即使不设置上述 nmi_watchdog 内核选项，只要满足几个条件，比如 irq 执行中、当前任务处于空闲状态或 init 状态等，就会通过 kexec 启动第 2 内核（second kernel），执行崩溃转储。不论如何，如果要保证通过 nmi_watchdog 的超时获得转储，就按照上面的方法设置。

另外，NMI watchdog 超时时的内核信息的例子请参见 **“HACK#38 内核停止响应（自旋锁篇之二）”**。

137

总结

本 hack 讲述了利用 NMI watchdog 在死机时获取崩溃转储的方法。

——大和一洋

内核独有的汇编指令（之一）

本 hack 介绍一些用户空间中不常见的汇编指令。

分析内核的转储，就是要对比阅读内核源代码和使用 crash 命令的 dis 等显示的汇编代码进行分析。

本 hack 介绍一些用户空间中很难见到，内核中却频繁出现的汇编指令。objdump 命令的输出以 Linux 2.6.19 版内核为例。

BUG：ud2 指令

阅读内核的汇编代码，会经常遇到 ud2 指令。下面是 free_buffer_head()。

```
# objdump -d vmlinux-2.6.19
...
c0184731 <free_buffer_head>:
c0184731:  89 c2                 mov   %eax,%edx
c0184733:  8d 40 28              lea   0x28(%eax),%eax
c0184736:  39 42 28              cmp   %eax,0x28(%edx)
c0184739:  74 08                 je    c0184743 <free_buffer_head+0x12>
c018473b:  0f 0b                 ud2a  ————————❶
c018473d:  8b 0b                 mov   (%ebx),%ecx
c018473f:  dc 34 33              fdivl (%ebx,%esi,1)
c0184742:  c0 a1 a0 35 4a c0 e8  shlb  $0xe8,0xc04a35a0(%ecx)
c0184749:  10 11                 adc   %dl,(%ecx)
...
```

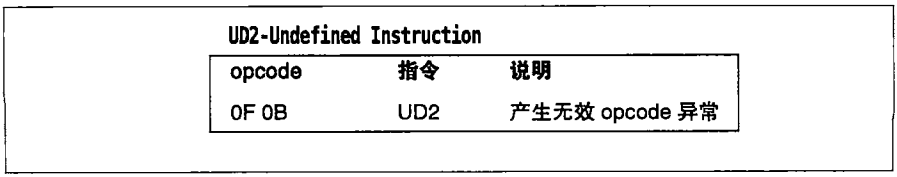
❶处有个 ud2a。阅读内核源代码就能立即明白，❶的源代码为①。

```
[fs/buffer.c]
void free_buffer_head(struct buffer_head *bh)
{
     BUG_ON(!list_empty(&bh->b_assoc_buffers)); ————①
     kmem_cache_free(bh_cachep, bh);
     get_cpu_var(bh_accounting).nr--;
     recalc_bh_state();
     put_cpu_var(bh_accounting);
}

[include/asm-generic/bug.h]
#ifndef HAVE_ARCH_BUG_ON
#define BUG_ON(condition) do { if (unlikely((condition)!=0)) BUG(); }
while(0)
#endif

[include/asm-i386/bug.h]
#define BUG() __asm__ __volatile__("ud2\n") ————①
```

138

可见，ud2 指令之后立即出现 BUG_ON()或 BUG()。

该指令在 Intel 的手册中如图 3-5 所示。



UD2-Undefined Instruction

opcode	指令	说明
0F 0B	UD2	产生无效 opcode 异常

图 3-5 Intel 手册中的说明

“无效 opcode 异常”就是 CPU 引发的异常（中断编号为 6），内核执行 ud2 指令，使 CPU 接受该异常。内核接受该异常后通过 handle_BUG()调用 BUG()。

禁止/允许中断：sti、cli 指令

内核的汇编代码中还经常见到 sti 和 cli 指令。下面是 on_each_cpu()。

```
# objdump -d vmlinux-2.6.19
...
c0129e5a <on_each_cpu>:
...
c0129e64:  8b 44 24 14        mov  0x14(%esp),%eax
c0129e68:  89 04 24           mov  %eax,(%esp)
c0129e6b:  89 f8              mov  %edi,%eax
139
c0129e6d:  e8 70 cc fe ff     call c0116ae2 <smp_call_function>
c0129e72:  89 c3              mov  %eax,%ebx
c0129e74:  fa                 cli  ————————❷
c0129e75:  89 f0              mov  %esi,%eax
c0129e77:  ff d7              call *%edi
c0129e79:  fb                 sti  ————————❸
c0129e7a:  5a                 pop  %edx
c0129e7b:  89 d8              mov  %ebx,%eax
c0129e7d:  5b                 pop  %ebx
...
```

内核源代码如下所示。

```
 [kernel/softirq.c]
int on_each_cpu(void (*func) (void *info), void *info, int retry, int wait)
{
      int ret = 0;
```

```
        preempt_disable();
        ret = smp_call_function(func, info, retry, wait);
        local_irq_disable();————————————②
        func(info);
        local_irq_enable(); ————————————③
        preempt_enable();
        return ret;
}

[include/linux/irqflags.h]
#ifdef CONFIG_TRACE_IRQFLAGS
...
#else
# define trace_hardirqs_on()        do { } while (0)
# define trace_hardirqs_off()       do { } while (0)
...
#endif

#define local_irq_enable() \
        do { trace_hardirqs_on(); raw_local_irq_enable(); } while (0)
#define local_irq_disable() \
        do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
...
```

140

```
[include/asm-i386/irqflags.h]
static inline void raw_local_irq_disable(void)
{
        __asm__ __volatile__("cli" : : : "memory"); ————②
}

static inline void raw_local_irq_enable(void)
{
        __asm__ __volatile__("sti" : : : "memory"); ————③
}
```

cli指令通过local_irq_disable()禁止中断，而sti指令通过local_irq_ enable()允许中断。

总结

本hack介绍了内核中常见的汇编指令ud2、sti、cli。

参考文献

- IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 A：命令セット・リファレンス A-M
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2A_i.pdf
- IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 B：命令セット・リファレンス N-Z
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2B_i.pdf

——大岩尚宏

内核独有的汇编指令（之二）

本 hack 从 x86 架构的汇编层面讲述内核中频繁出现的 current 宏。

什么是 current

在内核处理中，经常出现获取现在运行进程的 task_struct 结构的处理。在内核代码中，该处理被写成 current 宏。内核代码上出现 current 时，就表示要获取正在运行的进程的 task_struct 结构。所谓 task_struct 结构，就是内核内部管理进程状态的数据结构。

141

在内核的崩溃转储中分析 bug，大多数情况下必须阅读汇编代码。由于 current 被频繁使用，在对照内核源代码和汇编代码时，以 current 为标志就很方便。

task_struct 结构和 thread_info 结构

内核 2.6 与内核 2.4 不同，管理进程的 task_struct 结构位于 SLAB 中。

task_struct 结构和 thread_info 结构互相拥有指向对方的指针。

进程在内核中占据了一定的栈空间，而栈空间的开头就是 thread_info 结构。另外，内核栈包含 2 页，大小为 8KB，从后往前使用。4KB 栈的话就只有 1 页。

内核 2.4 中不存在 thread_info 结构，内核栈的开头保存的是 task_struct 结构。

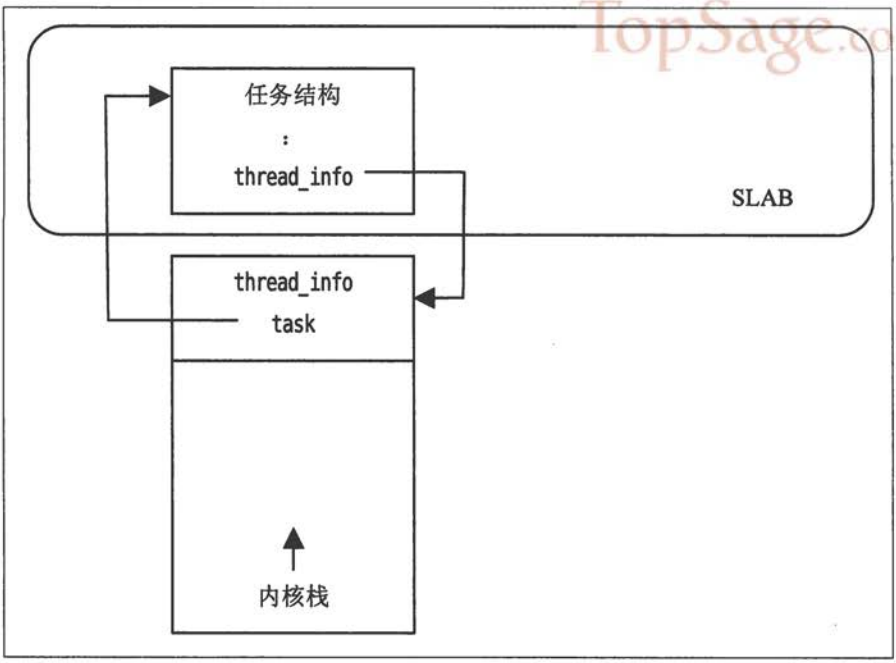


图 3-6 task_struct 结构和 thread_info 结构

142

current 获取处理的详细内容

下面介绍一下 current 获取处理的汇编代码的阅读方法。

不同架构、不同内核版本的 current 获取处理也不相同，这里分别说明 x86 架构下的 32 位（i386）和 64 位（x86_64）版本。

i386：32 位

2.6.19 以前的内核处理如下。

```
c0101628:     be 00 e0 ff ff        mov   $0xffffe000,%esi
c010162d:     21 e6                 and   %esp,%esi
c010162f:     8b 1e                 mov   (%esi),%ebx
```

首先用当前栈指针 esp 获取 thread_info 结构的指针，并从中获取 task_struct 结构的指针。这步处理的关键之处是将栈指针 esp 与 0xffffe000 执行逻辑与操作（AND）。清除栈指针的低 13 位，就能获取 8KB 的内核栈空间的开头地址。因此，寻找 current 时，只需查找对内核栈执行逻辑与操作（AND）的地方即可。

从 2.6.20 开始，percpu 空间中保存了指向 current 的指针，因此利用段选择器 fs 访问 percpu 空间。2.6.20 之后的处理如下。

```
c1002173:     64 8b 35 00 f0 69 c1   mov   %fs:0xc169f000,%esi
```

利用段选择器 fs 和偏移量 0xc169f000 进行访问。2.6.20 之后 current 指针位于 percpu 空间，因此寻找通过段选择器 fs 访问 percpu 空间的处理即可找到。

x86_64：64 位

x86_64 在 PDA（per processor data structure）空间中保存了指向 current 的指针，因此利用 gs 段寄存器访问 PDA 即可获得 current。

```
ffffffff8010cfdf:     65 48 8b 3c 25 00 00   mov   %gs:0x0,%rdi
ffffffff8010cfe6:     00 00
```

143

利用段选择器 gs 和偏移量 0 来访问。由于当前进程的 task_struct 结构指针保存在 PDA 空间的开头，因此段选择器 gs 和偏移量 0 的访问，就是 current 的处理所在。

开发版内核（预定合并到 2.6.30）上，x86_64 专用的 PDA 空间已被删除，准备统一到 percpu 中。

```
ffffffff81009186:     65 48 8b 04 25 00 b0   mov   %gs:0xb000,%rax
ffffffff8100918d:     00 00
```

尽管段选择器不同，但与 2.6.20 之后的 i386 中的处理过程是很相似的。

总结

本 hack 讲述了 Linux 内核中常见的 current 在 x86 架构下的汇编代码。current 是个频繁使用的命令，追踪内核代码时非常有用。

参考文献

- Intel® 64 and IA-32 Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm

——岛本裕志

第4章

应用程序调试实践

hack #26~#32

本章讲述在实践中如何调试用户应用程序。栈溢出而导致 segmentation fault（SIGSEGV）[注1]、backtrace 无法正确显示、由于数组非法访问导致栈被破坏、利用 watch point 检测非法内存访问、malloc()/free()引发的故障、应用程序停止响应等，本章将通过这些各种各样的案例来介绍调试方法。

发生 SIGSEGV，应用程序异常停止

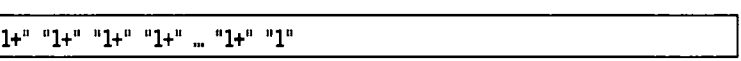
由于栈溢出导致 segmentation fault 的调试。

应用程序执行了非法访问内存等操作后，就会发生 SIGSEGV 异常而停止。SIGSEGV 发生的情况包括：(1) NULL 指针访问，(2) 指针被破坏等原因导致的非法地址访问，(3) 栈溢出导致访问超出了已分配的地址空间等。

此处讲解由于栈溢出而导致 SIGSEGV 发生的情况下的调试方法。

下面是个发生 segmentation fault 的例子。其中 eval("1+"*100000+"1")程序如图 4-1 所示。

```
$ ruby1.8 -e 'eval("1+" * 100000 + "1")'
Segmentation fault
```



注 1：segmentation fault 通常被翻译成“段错误”，但这种译法的含义并不明确，再加上 segmentation fault 本身就是错误信息，因此本书保留原文不翻译。

——译者注

图4-1 eval("1+" * 100000 + "1")程序

146

eval("1+" * 100000 + "1")这个表达式的意思是，将100000个"1+"字符串连接在一起，然后再连接"1"这个字符串，最后将结果作为Ruby程序来计算（eval）。

设置系统以生成core文件（内核转储文件）。

```
$ ulimit -c unlimited
$ ruby1.8 -e 'eval("1+" * 100000 + "1")'
Segmentation fault (core dumped)
$ ls core
core
```

这样应用程序就会生成core文件。用调试器试着分析一下。

```
$ gdb ruby1.8 core
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(no debugging symbols found)
中间省略
Loaded symbols for /lib/ld-linux.so.2
(no debugging symbols found)
Core was generated by `ruby1.8 -e eval("1+" * 100000 + "1")'.
Program terminated with signal 11, Segmentation fault.
[New process 24488]
#0  0xb7e22cb7 in ?? () from /usr/lib/libruby1.8.so.1.8
```

系统中安装的应用程序（此例中为ruby1.8）中不包含调试信息，所以无法显示符号（symbol）。但是，根据栈帧（stack frame）等信息也可以在某种程度上推测出原因。

通过 bt（backtrace）命令可以获得栈帧信息。若执行后显示大量的栈帧信息，这表明函数被递归调用了。其实，只需显示开头的几条即可，本例中显示开头 10 条。

147

命令格式为 bt <数字> 。

```
(gdb) bt 10
#0 0xb7e22cb7 in ?? () from /usr/lib/libruby1.8.so.1.8
#1 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#2 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#3 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#4 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#5 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#6 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#7 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#8 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#9 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
```

仔细观察栈帧信息就能发现，函数被 0xb7e22f3a 这个地址调用多次。因此可以怀疑，递归函数调用产生栈溢出，导致应用程序异常结束。

源代码层面的调试

接下来用 gdb 跟踪一下源代码。首先，要用 gcc 的 -g 选项编译应用程序（ruby），然后用 gdb 启动。

```
(gdb) run -e 'eval("1+" * 100000 + "1")'
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby -e 'eval("1+" * 100000 + "1")'
[Thread debugging using libthread_db enabled]
[New Thread 0xb7d3d6b0 (LWP 24646)]
[New Thread 0xb7f24b90 (LWP 24649)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb7d3d6b0 (LWP 24646)]
iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b138, node=0x931dd74, poped=0) at compile.c:2883
```

可以看到应用程序引发了 SIGSEGV。使用 gdb 时，收到信号就会执行预定的动作，SIGSEGV 信号会使程序在相应位置自动停止。

用 info signal 命令可以查看 gdb 能够处理的信号列表。

停止位置位于源代码下面显示的地方。用 emacs 启动 gdb 的话就能自动显示出来，十分方便。

148

```
/**
  compile each node
  self: InstructionSequence
  node: Ruby compiled node
  poped: This node will be poped
 */
static int
iseq_compile_each(rb_iseq_t *iseq, LINK_ANCHOR *ret, NODE * node, int poped)
{  /* 在这里停止 */
   enum node_type type;
   if (node == 0) {
  if (!poped) {
     debugs("node: NODE_NIL(implicit)\n");
     ADD_INSN(ret, iseq->compile_data->last_line, putnil);
  }
  return COMPILE_OK;
   }
```

用 bt 命令获取栈帧信息。取 5 个就够了。

```
(gdb) bt 5
#0  iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b138, node=0x931dd74, poped=0) at
compile.c:2883
#1  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b278, node=0x931dd38,
poped=0) at compile.c:3954
#2  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b3b8, node=0x931dcfc,
poped=0) at compile.c:3954
#3  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b4f8, node=0x931dcc0,
poped=0) at compile.c:3954
#4  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b638, node=0x931dc84,
poped=0) at compile.c:3954
(More stack frames follow...)
```

可以看出，函数被 0x0811154d 这个地址反复调用。通过 up 命令，将栈帧向上回溯一层。

```
(gdb) up
```

```
#1  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b278, node=0x931dd38,
poped=0) at compile.c:3954
```

149

该地址的源代码为下述 COMPILE()的位置。

```
    else {
      ADD_LABEL(ret, label);
    }
    break;
     }
  }
#endif
  /* reciever */
  if (type == NODE_CALL) {
     COMPILE(recv, "recv", node->nd_recv); /* 从这里调用 */
  }
  else if (type == NODE_FCALL || type == NODE_VCALL) {
     ADD_CALL_RECEIVER(recv, nd_line(node));
  }
  /* args */
  if (nd_type(node) != NODE_VCALL) {
     argc = setup_args(iseq, args, node->nd_args, &flag);
  }
```

从源代码可以看出，COMPILE 的宏定义如下所示，递归调用了 iseq_compile_each()函数。

```
/* compile node */
#define COMPILE(anchor, desc, node) \
  (debug_compile("== " desc "\n", \
            iseq_compile_each(iseq, anchor, node, 0)))
```

通过分析源代码可以得知，本例中反复递归调用函数导致了栈溢出。

栈溢出导致 SIGSEGV 的应对方法

一般而言，捕获到信号后应当准备信号处理程序并执行某种操作。但是，栈溢出导致 SIGSEGV 发生的情况下，栈空间已经溢出，已经出现了非法访问，就连启动信号处理程序所需的栈都无法保证，所以不能这样处理。因此，为捕获栈溢出，需要使用备用栈，相应的函数为 sigaltstack(2)。

150

man page 中的例子如下所示。

```
stack_t ss;

ss.ss_sp = malloc(SIGSTKSZ);
if (ss.ss_sp == NULL)
   /* 错误处理 */;
ss.ss_size = SIGSTKSZ;
ss.ss_flags = 0;
if (sigaltstack(&ss, NULL) == -1)
   /* 错误处理 */;
```

参考这段程序，可以写出以下的补丁。

```
$ svn diff signal.c
Index: signal.c
===================================================================
--- signal.c  (revision 20086)
+++ signal.c  (working copy)
@@ -47,6 +47,10 @@
 # define NSIG (_SIGMAX + 1)     /* For QNX */
 #endif

+#ifdef SIGSEGV
+static int is_altstack_defined = 0;
+#endif
+
 static const struct signals {
    const char *signm;
    int signo;
@@ -410,6 +414,28 @@
 typedef RETSIGTYPE (*sighandler_t)(int);

 #ifdef POSIX_SIGNAL
+#define ALT_STACK_SIZE (4*1024)
+#ifdef SIGSEGV
+/* alternate stack for SIGSEGV */
+static void register_sigaltstack() {
+   stack_t newSS, oldSS;
+
+   if(is_altstack_defined)
```

```
+    return;
+
+   newSS.ss_sp = malloc(ALT_STACK_SIZE);
+   if(newSS.ss_sp == NULL)
+     /* should handle error */
+      rb_bug("register_sigaltstack. malloc error\n");
+   newSS.ss_size = ALT_STACK_SIZE;
+   newSS.ss_flags = 0;
+
+   if (sigaltstack(&newSS, &oldSS) < 0)
+      rb_bug("register_sigaltstack. error\n");
+   is_altstack_defined = 1;
+}
+#endif
+
 static sighandler_t
 ruby_signal(int signum, sighandler_t handler)
 {
@@ -432,7 +458,12 @@
    if (signum == SIGCHLD && handler == SIG_IGN)
  sigact.sa_flags |= SA_NOCLDWAIT;
 #endif
-   sigaction(signum, &sigact, &old);
+#ifdef SA_ONSTACK
+   if (signum == SIGSEGV)
+      sigact.sa_flags |= SA_ONSTACK;
+#endif
+   if (sigaction(signum, &sigact, &old) < 0)
+      rb_bug("sigaction error.\n");
    return old.sa_handler;
 }

@@ -663,6 +694,7 @@
 #ifdef SIGSEGV
      case SIGSEGV:
       func = sigsegv;
+      register_sigaltstack();
       break;
 #endif
 #ifdef SIGPIPE
@@ -1070,6 +1102,7 @@
```

152

```
    install_sighandler(SIGBUS, sigbus);
#endif
#ifdef SIGSEGV
+   register_sigaltstack();
    install_sighandler(SIGSEGV, sigsegv);
#endif
    }
```

应用该补丁后的程序执行结果如下所示。

```
$ ./ruby -e 'eval("1+" * 100000 + "1")'
-e:1: [BUG] Segmentation fault
ruby 1.9.0 (2008-11-01 revision 20086) [i686-linux]

-- control frame ----------
c:0004 p:---- s:0010 b:0010 l:000009 d:000009 CFUNC  :eval
c:0003 p:0017 s:0006 b:0006 l:000005 d:000005 TOP   -e:1
c:0002 p:---- s:0004 b:0004 l:000003 d:000003 FINISH :inherited
c:0001 p:0000 s:0002 b:0002 l:000001 d:000001 TOP   <dummy toplevel>:17
---------------------------
-e:1:in `eval': stack level too deep (SystemStackError)
 from -e:1:in `<main>'
```

不管怎样，segmentation fault（SIGSEGV）总是要异常结束的，但与其什么信息也不给，还不如输出异常结束的提示信息，这样有助于应用程序调试。

此外，这个补丁已在 http://redmine.ruby-lang.org/repositories/revision/ruby-19?rev=20293 中应用到了 ruby 上。

总结

本 hack 讲述了栈溢出导致 SIGSEGV 从而异常结束时的调试方法。

参考文献

- 《BINARY HACKS》的“HACK#76 用 sigaltstack 处理 stack overflow”（pp.291~300）

——吉冈弘隆

153

backtrace 无法正确显示

本 hack 以在多线程应用程序中由于线程间冲突导致的栈破坏为例，讲述 backtrace 无法正确显示时的调试方法。

概要

栈破坏有时会导致问题难以分析。特别是，由于无法获取 backtrace 信息，追溯问题现象发生的路径就变得异常困难。此外，由于存在栈破坏，可以说 backtrace 信息也并不完整。要记住，调试器的 backtrace 并非万能钥匙。

问题内容

某个进行线程间通信的程序中含有 bug，生成了 core 文件（内核转储文件）。

检查 backtrace

用调试器分析时的固定做法就是先 backtrace。但是，对该例中的程序运行时生成的 core 执行 backtrace，却完全看不出什么函数被调用，似乎是 nanosleep()执行过程中产生了 SIGSEGV，但从 th_req()是如何到 nanosleep()的呢？

```
(gdb) bt
#0 0x0000003b4869ac80 in nanosleep () from /lib64/libc.so.6
#1 0x000ee1c2000ee1c1 in ?? ()
#2 0x000ee1c4000ee1c3 in ?? ()
#3 0x000ee1c6000ee1c5 in ?? ()
#4 0x000ee1c8000ee1c7 in ?? ()
#5 0x000ee1ca000ee1c9 in ?? ()
#6 0x000ee1cb000ee1ca in ?? ()
#7 0x000ee1cd000ee1cc in ?? ()
#8 0x0000000000000002 in ?? ()
#9 0x0000000001877c90 in ?? ()
#10 0x000000004162f130 in ?? ()
#11 0x0000000000400d02 in th_req (p=0x1877c90) at bug.c:167
```

154

为何出现这样的 backtrace 信息？

要理解这一点，需要了解 GDB 等调试器是如何输出 backtrace 的。

什么是 backtrace

调试器的 backtrace 是根据栈里保存的函数返回地址来显示的。根据栈空间上的返回地址和调试信息得出的栈使用量，依次求出调用者函数。也就是说，调试器的 backtrace 的地址来自进程的栈上。关于栈的内容请参见“**HACK#9 调试时必需的栈知识**”。

如上所述，backtrace 信息依赖于栈的信息。因此，像本例这种 backtrace 不正确的情况，基本上可以认为栈被破坏了。

本例比较极端，实际上也有比较实际的例子。但是，如果栈被破坏，就不能信任调试器生成的 backtrace 信息。极端一点说，信赖 backtrace 只有在栈没有被破坏的前提下才成立。认为调试器的 backtrace 信息绝对正确是十分危险的。

查看寄存器和栈

利用 GDB 进行分析时，寄存器信息绝不可忽视。来看看当前的寄存器信息。

```
(gdb) info reg
...
rsp            0x4162f0c8          0x4162f0c8
...
rip            0x3b4869ac80        0x3b4869ac80 <nanosleep+96>
...
```

指令指针 RIP 的值为 0x3b4869ac80，下面来查看一下要执行什么指令。

```
(gdb) x/i 0x3b4869ac80
0x3b4869ac80 <nanosleep+96>:   retq
```

155

这是个 retq 指令，是函数的返回指令。在 x86 中，函数返回就是从栈指针的地址处取出返回地址，然后跳转到该地址。因此，可以认为跳转地址不正确，也就是栈上的返回地址不正确。

```
(gdb) x/g 0x4162f0c8
0x4162f0c8:    0x000ee1c2000ee1c1
```

本例中要返回到地址 0x000ee1c2000ee1c1，也就是说跳转到该地址，此时只能怀疑返回地址被破坏了。其实，可以认为是栈被破坏了。

```
(gdb) bt
#0 0x0000003b4869ac80 in nanosleep () from /lib64/libc.so.6
#1 0x000ee1c2000ee1c1 in ?? ()
```

没错，最初的 backtrace 显示的结果对于调试器来说没有问题，但从该进程的栈数据来看，最后的 nanosleep() 是从地址 0x000ee1c2999ee1c1 调用的，而 0x000ee1c2999ee1c1 不是正确的地址。

像这种栈破坏导致无法获取 backtrace 信息的情况，从调试角度来看是极其严重的问题。另外，栈空间还被用做局部变量的保存空间，因此局部变量内容也有可能被破坏。也就是说，GDB 输出的局部变量也变得不可信。

调查栈破坏的方法有许多种，但最现实的方法就是根据被破坏的数据内容，判断执行写入的位置，看看有没有对栈空间（也就是自动变量空间）的引用、指针传递处理。

本例的应用程序中，在线程间的数据处理上传递了栈的指针，导致其他线程向该地址写入了数据。

而且这个“其他线程”向栈内写入数据的操作被推迟了，从而导致了栈破坏。

图 4-2 和图 4-3 分别为该应用程序正常时的行为示意图和问题发生时的行为示意图。图中根据栈的状态画出了请求线程。

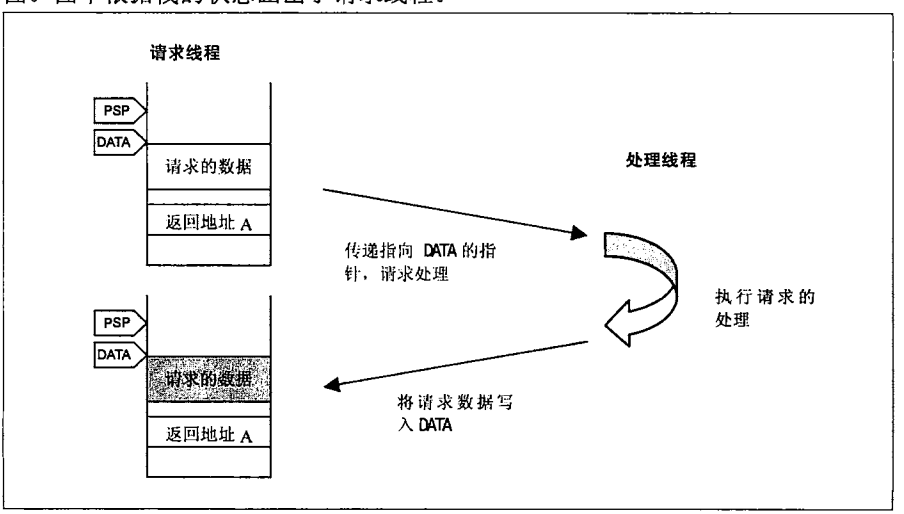


图 4-2 正常时应用程序的行为

156

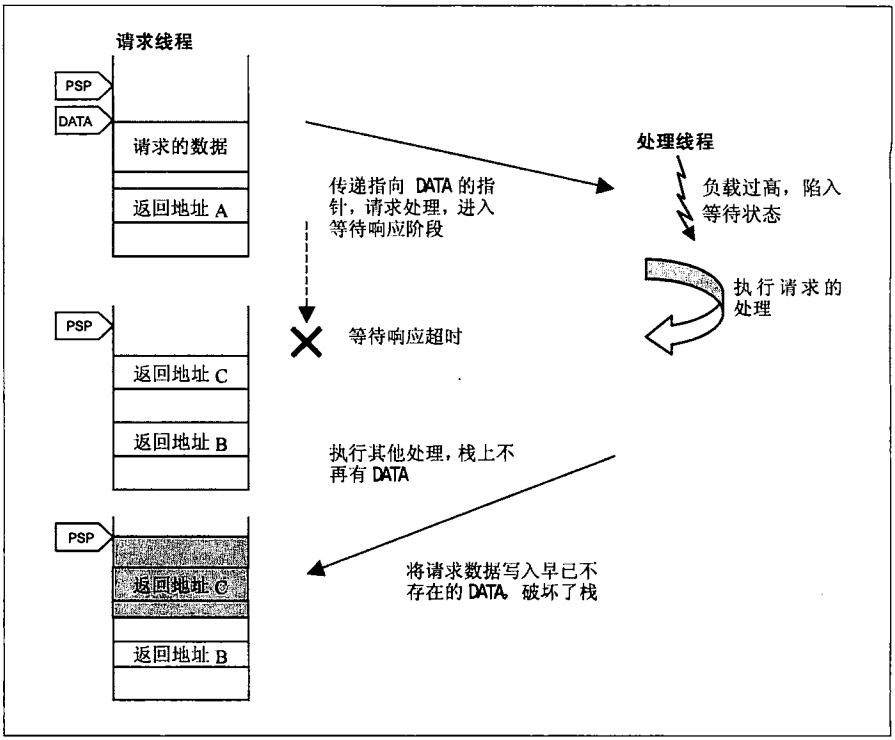


图 4-3 发生问题时应用程序的行为

总结

本 hack 讲述了栈破坏导致的问题。栈破坏一旦发生，就会出现无法获取 backtrace 信息、无法预料的变量破坏等问题，调试本身也会变得很困难。

157

反过来说，如果觉得 backtrace 不太对，可以怀疑是否栈被破坏了。

参考文献

- Intel® 64 and IA-32 Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm

——岛本裕志

数组非法访问导致内存破坏

本 hack 讲述引发 segmentation fault 的原因之一——错误的数组操作的调试方法。

数组的错误操作

错误地操作数组导致的典型 bug 之一就是缓冲区溢出，也就是说向已分配的内存空间之外写入数据。特别是如果这类 bug 发生在栈上的缓冲区中，就可能引发安全漏洞，因此出现了许多预防措施和应对措施，如通过指定缓冲区大小来编写安全的函数、源代码检查工具、编译器在构建时的报警等。即便如此，这种 bug 仍时有发生。此外，在计算数组的下标时，如果计算方法有误，就可能算出负下标，也可能会引起缓冲区溢出，本 hack 后半部分也会讲述这种 bug 的调试方法。

可怀疑是缓冲区溢出的情况

可以怀疑是缓冲区溢出的情况之一就是，即使指定了编译选项-g，利用 GDB 读入 core 并显示 backtrace 之后，栈帧中还是没有显示符号名，如下所示。通常，指定-g 选项后，各个栈帧都应显示出函数名。

```
(gdb) bt
#0 0x20656c62 in ?? ()
#1 0x72727563 in ?? ()
#2 0x08040079 in ?? ()
#3 0x0804948c in ?? ()
#4 0xb8008ff4 in ?? () from /lib/libc.so.6
#5 0xb802aca0 in ?? () from /lib/ld-linux.so.2
#6 0x080483c0 in main ()
```

158

“HACK#27 backtrace 无法正确显示”也讲过，在这种情况下 backtrace 是不可靠的。实际上，用 GDB 显示停止处的代码，也像下面这样无法显示。这是因为 backtrace 显示的内容很可能不是实际跟踪的内容。也就是说，代码突然跳转到或者调用了错误的地址 0x20656c62，导致了 segmentation fault 的发生。

```
(gdb) x/i 0x20656c62
0x20656c62:  Cannot access memory at address 0x20656c62
(gdb) x/i 0x72727563
```

```
0x72727563:  Cannot access memory at address 0x72727563
```

栈帧的#0、#1 显示的地址上很难放置程序、共享内存等。大多数 i386 架构的 Linux 发行版中，程序被定位到 0x08000000 地址附近，共享库被定位到 0xb0000000 之后的地址。在 x86_64 中，程序定位到 0x400000 或 0x0600000 附近，共享库被定位到 0x3000000000、0x2aaaaaaa0000 地址附近，具体情况由连接器决定。不同的使用环境下，与连接器的定位地址有关的选项也可能发生变化，因此记住在调试对象的环境中，标准情况下程序和共享库会被定位到什么地址，在阅读 backtrace 时就会很方便。

运行地址的改变

那么，是从哪里跳转到或调用了不存在的地址呢？在调查这个问题之前，先来整理一下改变程序运行地址的方法。方法基本可以分为 3 类。第 1 类就是直接指定地址并调用。C 语言中的 if 或 for 语句等进行条件判断时会用到这种方法，调用同一源代码内的函数时也会使用这种方法。第 2 类就是指定一块内存区域，其中保存了跳转地址（参见“**HACK#63　理解用 GOT/PLT 调用函数的原理**”）。第 3 类方法就是执行 ret 命令，用于函数结束时返回调用者函数。ret 命令将栈指针指向的位置的值作为跳转（返回）地址使用（参见“**HACK#9　调试时必需的栈知识**”）。

如果 bug 破坏了这些方法用到的值（被错误的地址覆盖），就可能跳转到错误的地址。但是，第 1 类方法使用的地址很难被破坏，这是因为第 1 类方法使用的地址一般保存在只读空间内。所以，尝试破坏该地址就会产生 segmentation fault。此时，core 文件中会记录这一瞬间的程序计数器的值，因此分析也比较容易。

159

相反，第 2、第 3 类方法使用的地址位于 GOT（Global Offset Table）或栈等可写的空间中，因此即使 bug 破坏其内容，也无法立即被检测到。之后，问题就会以本 hack 开头提到的 segmentation fault 的形式显露出来。

确定破坏跳转地址值的位置（栈破坏）

简单来说，确定破坏地址值的位置的方法，就是将错误地址当做数据，寻找复制该数据的地方。以本 hack 开头的栈跟踪为例，我们要寻找错误地写入 0x20656c62 这个数据的地方。

这种调查中，很重要的是需要怀疑数据是否为字符串的一部分，因为错误地将数

据写入地址的典型情况之一就是字符串复制。由于字符串的输入长度很难预测，若缓冲区过小，再加上对输入字符串的长度检查不完善，就可能发生这种状况。在本 hack 的例子中以此观点查看地址 0x20656c62，就会发现它是 ASCII 字符串“elb”。考虑到 x86_64 和 i386 架构为 little endian，因此可以认为是“…ble …”这个字符串的一部分被写入了。但是，连续出现 4 个字符代码对应的数字的可能性也不是没有，所以必须进一步验证这个假说是否成立。上一节说明的第 3 类方法是从栈中获取返回地址，因此先调查一下栈。

```
(gdb) x/30c $esp-15
0xbfc6f651:    100 'd' 105 'i' 110 'n' 103 'g' 32 ' ' 118 'v' 101 'e'103 'g'
0xbfc6f659:    101 'e' 116 't' 97 'a' 98 'b' 108 'l' 101 'e' 32 ' ' 99 'c'
0xbfc6f661:    117 'u' 114 'r' 114 'r' 121 'y' 0 '\0' 4 '\004'     8 '\b' -116 '\214'
0xbfc6f669:    -108 '\224'    4 '\004'     8 '\b' -12 '??' -1 '??' 4 '\004'
(gdb) p (char*)$esp-20
$9 = 0xbfc6f64c " building vegetable curry"
```

如上所示，栈指针指向的空间前后几乎都是字符串，因此将这种字符串写入栈上的错误位置的可能性很高。实际上引发该问题的源代码如下所示。“building…… curry”这个字符串是源代码中的 names 这个字符串的一部分，而使用 names 的是用 strcpy()复制 buf 的地方。此时就能立刻看出，原因就是 buf 只有 5 字节长，复制的字符串超过了这个长度（导致缓冲区溢出）。

160

```
char names[] = "book cat dog building vegetable curry";

void func(void)
{
   char buf[5];
   strcpy(buf, names);
}

int main(void)
{
   func();
   return EXIT_SUCCESS;
}
```

用图来表示则如图 4-4 所示。strcpy()写入字符串的空间，原本是用于保存返回 main()的地址的，所以 func()结束之后返回 main 时，就把“ble ”对应的 0x20656c62 当做返回地址使用，导致 segmentation fault。

本例中的地址是字符串的一部分，比较容易判别，而只是将数值错误地写入的情况会更难调查。即便如此，如果那个数值经常在程序中出现，就可以将调查范围缩小到使用该数值的部分。

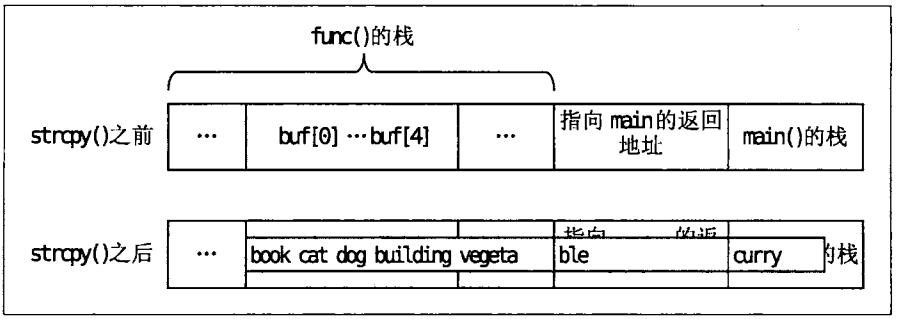


图 4-4 func()的栈

161

确定破坏跳转地址值的位置（GOT 破坏）

访问数据空间中静态分配的数组时，如果出现 bug，也会发生类似的现象。下面讨论程序在发生 segmentation fault 时，产生的 core 的 backtrace 如下所示。

```
(gdb) bt
#0 0x00000008 in ?? ()
#1 0x0000000a in ?? ()
#2 0x00000008 in ?? ()
#3 0x080483ca in main () at bufov2.c:19
```

本例中，backtrace 的#0 所示的地址也不存在。而且，其数值为 8，很难找出其原因，唯一的线索只有栈帧最后的 bufov2.c:19。我们对其进行反汇编，看看这个 0x080483ca 处执行了什么操作。

```
(gdb) disas 0x080483ca
...
0x080483be <main+64>:  movl  $0x80484a8,(%esp)
0x080483c5 <main+71>:  call  0x80482b0 <_init+56>
0x080483ca <main+76>:  mov   $0x0,%eax
...
```

可见，栈跟踪中显示的 0x080483ca，可以认为是前面的 call 指令放到栈上的。也就是说，从 0x080483c5 地址的 call 指令执行，到返回 0x080483ca 之间有可能发生

了问题。那么，被调用的 0x80482b0 处发生了什么呢？调查后可以发现，它跳转到了 0x80495b0 号地址中存储的地址（0x080482b6）。这样跟踪指令之后，就会发现 0x08048296 地址处的 jmp 指令访问的地址值为 0x8。跳转到该地址导致 segmentation fault，这样就能说得通了。

```
(gdb) disas 0x80482b0 0x80482c0
Dump of assembler code from 0x80482b0 to 0x80482c0:
0x080482b0 <_init+56>: jmp   *0x80495b0
0x080482b6 <_init+62>: push  $0x8
0x080482bb <_init+67>: jmp   0x8048290 <_init+24>
(gdb) x 0x80495b0
0x80495b0 <_GLOBAL_OFFSET_TABLE_+16>:  0x080482b6
(gdb) disas
0x080482b6 <_init+62>: push  $0x8
0x080482bb <_init+67>: jmp   0x8048290 <_init+24>
(gdb) disas 0x8048290 0x80482a0
Dump of assembler code from 0x8048290 to 0x80482a0:
0x08048290 <_init+24>: pushl 0x80495a4
0x08048296 <_init+30>: jmp   *0x80495a8
0x0804829c <_init+36>: add   %al,(%eax)
0x0804829e <_init+38>: add   %al,(%eax)
End of assembler dump.
(gdb) x 0x80495a8
0x80495a8 <_GLOBAL_OFFSET_TABLE_+8>:   0x00000008
```

162

这里是通过 PLT 调用库函数的代码。详情参见“HACK#63 理解用 GOT/PLT 调用函数的原理”。

那么，哪里写入了这个 8 呢？刚才也说过，这个数字太普通，因此很难找出，但是如 GDB 所示，0x80495a8 是 GOT 的一部分，通常的用户程序不会写入该区域。因此，很可能是错将 8 当做了要保存的地址写入了 GOT 区域。而弄错地址的情况（bug）也多种多样，很难一概而论，可以假设应当写入的地址就位于附近，像下面这样调查该地址附近的整体结构。

```
$ objdump -s bufov2
...
Contents of section .fini:
 8048484 5589e553 e8000000 005b81c3 13110000 U..S.....[......
```

```
 8048494 50e876fe ffff595b c9c3          P.v...Y[..
Contents of section .rodata:
 80484a0 03000000 01000200 54686973 20697320 ........This is
 80484b0 61206d65 73736167 650a00         a message..
Contents of section .eh_frame:
 80484bc 00000000                    ....
Contents of section .ctors:
 80494c0 ffffffff 00000000             ........
Contents of section .dtors:
 80494c8 ffffffff 00000000             ........
Contents of section .jcr:
 80494d0 00000000                    ....
Contents of section .dynamic:
 80494d4 01000000 24000000 0c000000 78820408 ....$.......x...
...
 8049594 00000000 00000000             ........
Contents of section .got:
 804959c 00000000                    ....
Contents of section .got.plt:
 80495a0 d4940408 00000000 00000000 a6820408 ................
 80495b0 b6820408                    ....
Contents of section .data:
 80495b4 00000000 00000000 cc940408 01000000 ................
 80495c4 02000000
...
```

163

由上可见，被破坏的地址 0x80495a8 位于.got.plt section 中，紧跟着就是保存用户的静态数据的.data section。这种情况下能想到的一种可能性就是，将本应写入.data section 的数据错误地写入了.got.plt section。地址错误大多数情况是由于错误的指针操作或错误的数组下标引起的。用这个观点来看看源代码。

```
int my_data[2] = {1, 2};

int calc_index(void)
{
   /* 该函数有 bug，返回错误的值 */
   return -7;
}
```

```
int main(void)
{
   int idx = calc_index();
   my_data[idx] = 0x0a;
   my_data[idx+1] = 0x08;
   printf("This is a message\n");
   return EXIT_SUCCESS;
}
```

164

源代码中有个静态数组 my_data，并有对该数组进行写入的操作，而其下标由 calc_index()负责计算。到这里就会想到，是不是 calc_index()返回了错误的下标呢？实际上，读一下源代码就会立即发现真的是 calc_index()返回了错误的值。实际情况下下标计算会更复杂些，可能并不会一看代码就能发现 bug。但是，如果能调查到怀疑某个函数有问题，然后多关注该函数内的处理并找出 bug，就会容易许多。

总结

本 hack 介绍了错误的数组操作引发内存破坏而导致 segmentation fault 的调试方法。确定内存内容被破坏的过程可以按照某种步骤进行，但找出引发破坏的地方，就必须依靠一定的感觉和经验。下面的**“HACK#29 利用监视点检测非法内存访问”**就介绍了利用 GDB 的 watch 命令在内存内容被破坏的瞬间就发现的方法。

——大和一洋

利用监视点检测非法内存访问

本 hack 通过实例，介绍监视点（watchpoint）的用法。监视点能在指定变量或地址的数据被访问时暂停程序运行。

监视点何时有效？

“HACK#28 数组非法访问导致内存破坏”中介绍了非法内存写入导致 segmentation fault 的例子。[HACK#28]的分析过程基本上是分析 core 文件，根据几个证据确定实施非法内存写入的地点。但是，如果调试对象程序就在手头，而且能立即复现 bug，那么利用 GDB 的监视点（watchpoint）可以更高效地确定 bug 所在。本 hack 介绍利用监视点进行调试的方法。

[HACK#28]中通过调查 core 文件，可以比较容易地确定被错误地写入的地址为0x80495a8。下面我们要执行生成 core 文件的程序，利用 watchpoint 检测进行错误写入的地方。当指定的变量或内存地址等被读取或写入时监视点会中断程序执行，因此调查中断位置，即可得知是否发生了预期之外的写入操作。

165

监视点的设置方法

输入下面的命令设置监视点。

```
(gdb) watch *0x80495a8
Hardware watchpoint 1: *134518184
```

直接指定地址而不是指定变量名或符号时，要在地址前面加上*，这与在 break 命令中指定地址是一样的。然后在该状态下执行程序。

```
(gdb) run
Starting program: /home/yamato/tmp/bufovrn.work/bufov2
Hardware watchpoint 1: *134518184
Hardware watchpoint 1: *134518184

Old value = 0
New value = -1208018240
_dl_relocate_object (l=0xb7ffc710, scope=0xb7ffc8c8, lazy=1,
    consider_profiling=0) at dl-reloc.c:268
268       ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
(gdb) x/i -1208018240
0xb7ff1ac0 <_dl_runtime_resolve>:     push   %eax
```

这里发生了向 0x80495a8 的写入操作，所以程序中断了运行，New value 右边的数值就是写入指定地址的新数值。但是，这次 New value 为十六进制数的 0xb7ff1ac0，这是包含 glibc 中的_dl_runtime_resolve 函数（获取库函数地址并设置到 GOT 中的函数）的地址。调查 core 时已知非法写入的数值为 0x8，因此可以判断，这次写入为正常的写入操作。我们继续执行程序。

```
(gdb) c
Continuing.
Hardware watchpoint 1: *134518184
Hardware watchpoint 1: *134518184

Old value = -1208018240
```

```
New value = 8
main () at bufov2.c:19
19        printf("This is a message\n");
```

166

上述 New value 的值为 8，正是我们要找的写入操作。不过显示出的源代码为 19 行的 printf 语句，这是监视点的中断位置，但引发程序停止的似乎并不是这个 printf()。这是因为监视点的中断与断点不同，是在访问数据之后发生的，因此会显示 C 源代码中的下一行。

上面将监视点设在了 GOT 区域内。关于 GOT 请参见“HACK#63 理解用 GOT/PLT 调用函数的原理”。

寻找问题原因

进一步分析问题似乎应该阅读汇编代码。

```
(gdb) p $pc
$4 = (void *) 0x80483be
(gdb) disas
Dump of assembler code for function main:
...
0x0804839a <main+28>:  call  0x8048374 <calc_index>
0x0804839f <main+33>:  mov   %eax,0xfffffffc(%ebp)
0x080483a2 <main+36>:  mov   0xfffffffc(%ebp),%eax
0x080483a5 <main+39>:  movl  $0xa,0x80495c0(,%eax,4)
0x080483b0 <main+50>:  mov   0xfffffffc(%ebp),%eax
0x080483b3 <main+53>:  movl  $0x8,0x80495c4(,%eax,4) ————①
0x080483be <main+64>:  movl  $0x80484a8,(%esp)
0x080483c5 <main+71>:  call  0x80482b0 <_init+56>
...
End of assembler dump.
```

停止执行的地址的前一个命令为①的 movl，它将 8 这个数值写入了某个地址，该地址具体来说就是 0x80495c4 + %eax*4。%eax 的值和地址计算结果如下所示。

```
(gdb) p $eax
$5 = -7
(gdb) p/x (0x80495c4 + $eax*4)
$6 = 0x80495a8
```

167

0x80495a8这个地址就是执行了非法写入的地址，可以判断这部分就是直接原因。相应的C语言源代码用下述方法获取。

```
(gdb) list *0x080483b3
0x80483b3 is in main (bufov2.c:18).
13

14    int main(void)
15    {
16       int idx = calc_index();
17       my_data[idx] = 0x0a;
18       my_data[idx+1] = 0x08;
19       printf("This is a message\n");
20       return EXIT_SUCCESS;
21    }
```

到这里，idx的值不正确这个假设就很容易成立了。实际调查后会发现，下面出现了错误的负值，确定bug位于calc_index()中。

```
(gdb) p idx
$10 = -7
```

如上所示，在list命令的参数中直接指定地址，就能显示该地址对应的源代码文件：行号及那一行包含的函数。

总结

本 hack 通过实例介绍了能在指定变量或地址的数据被访问时暂停程序执行的监视点（watchpoint）的用法。

——大和一洋

malloc()和free()发生故障

本hack介绍内存双重释放导致的bug和解决方法。

错误使用内存相关库函数引起的bug

应用程序，特别是C语言应用程序的bug中，最常见的就是内存相关库函数的错

168

误使用引发的 bug，如内存的双重释放、访问分配空间之外内存等 bug。

这些 bug 主要表现为内存操作函数的错误处理，也就是在 malloc()、free()之后发生 SIGSEGV。

```
$ gdb ./membug -c core
Core was generated by `./membug'.
Program terminated with signal 11, Segmentation fault.
#0 0x0000003b4867217c in _int_free () from /lib64/libc.so.6
(gdb) bt
#0 0x0000003b4867217c in _int_free () from /lib64/libc.so.6
#1 0x0000003b48675f2c in free () from /lib64/libc.so.6
#2 0x0000000000400534 in do_free () at membug.c:30
#3 0x0000000000400588 in main (argc=<value optimized out>,
   argv=<value optimized out>) at membug.c:39
```

从这段 backtrace 信息中可见，问题似乎出在执行内存释放的库函数 free()及其内部使用的函数_int_free()中，但实际上是使用 free()函数的应用程序的问题，而不是库函数 free()的问题。

另外，这种 malloc()、free()导致 SIGSEGV 的情况其实是运气很好的情况。

最危险的情况是：程序可能不会受到内存破坏的影响而继续运行。此时可能会产生以下结果。

- 会在完全没关系的地方产生 SIGSEGV。
- 使用被破坏的数据进行计算，产生错误的结果。

malloc()、free()产生 SIGSEGV 时，在怀疑库函数的 bug 之前，要检查使用方法是否有问题，特别是要仔细确认是否进行了双重释放。

利用 MALLOC_CHECK_进行调试

现在的 glibc 中有个方便的调试标志，可以利用环境变量进行调试。

```
$ man malloc
```

可以查看详细信息。

来看看效果。

169 设置环境变量 MALLOC_CHECK_，就可以发现内存操作的相关 bug。

下面的例子中检查了 double free，即双重释放。

```
$ env MALLOC_CHECK_=1 ./membug
*** glibc detected *** ./membug: double free or corruption (top): 0x00000000020b2010 ***
======= Backtrace: =========
/lib64/libc.so.6[0x3b48672832]
/lib64/libc.so.6(cfree+0x8c)[0x3b48675f2c]
./membug[0x400534]
./membug[0x400588]
/lib64/libc.so.6(__libc_start_main+0xf4)[0x3b4861e074]
./membug[0x400449]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:06 42927111      /data/mywork/mines/0002/membug
00600000-00601000 rw-p 00000000 08:06 42927111      /data/mywork/mines/0002/membug
020b2000-020d3000 rw-p 020b2000 00:00 0             [heap]
3b48200000-3b4821b000 r-xp 00000000 08:02 7057200   /lib64/ld-2.7.so
3b4841a000-3b4841b000 r--p 0001a000 08:02 7057200   /lib64/ld-2.7.so
3b4841b000-3b4841c000 rw-p 0001b000 08:02 7057200   /lib64/ld-2.7.so
3b48600000-3b4874d000 r-xp 00000000 08:02 7057202   /lib64/libc-2.7.so
3b4874d000-3b4894d000 ---p 0014d000 08:02 7057202   /lib64/libc-2.7.so
3b4894d000-3b48951000 r--p 0014d000 08:02 7057202   /lib64/libc-2.7.so
3b48951000-3b48952000 rw-p 00151000 08:02 7057202   /lib64/libc-2.7.so
3b48952000-3b48957000 rw-p 3b48952000 00:00 0
3b53a00000-3b53a0d000 r-xp 00000000 08:02 7057488   /lib64/libgcc_s-4.1.2-20070925.so.1
3b53a0d000-3b53c0d000 ---p 0000d000 08:02 7057488   /lib64/libgcc_s-4.1.2-20070925.so.1
3b53c0d000-3b53c0e000 rw-p 0000d000 08:02 7057488   /lib64/libgcc_s-4.1.2-20070925.so.1
7f8540000000-7f8540021000 rw-p 7f8540000000 00:00 0
7f8540021000-7f8544000000 ---p 7f8540021000 00:00 0
7f854738c000-7f854738e000 rw-p 7f854738c000 00:00 0
7f85473ae000-7f85473b0000 rw-p 7f85473ae000 00:00 0
7fff4f39b000-7fff4f3b0000 rw-p 7ffffffea000 00:00 0             [stack]
7fff4f3ff000-7fff4f400000 r-xp 7fff4f3ff000 00:00 0             [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0         [vsyscall]
Aborted (core dumped)
```

170 像这样，利用环境变量 MALLOC_CHECK_即可容易地发现应用程序中的双重释放 bug。

但也要知道，该方法并不适用所有情况。用 MALLOC_CHECK_没有检测出问题，并不

能说明应用程序没有 bug。

总结

本 hack 说明了 malloc()、free()等内存操作库函数表现出的问题，即内存的双重释放、非法区域释放等。

malloc()、free()等标准 C 库函数的内存操作函数上发生问题时，首先要怀疑应用程序是否发生了双重释放等 bug。此外，如果使用最新的 glibc，用环境变量 MALLOC_CHECK_进行调试会收到不错的效果。

内存双重释放导致的 bug 与内存泄漏并列，是很难发现真正原因的 bug 之一。

参考文献

- Manpage of MALLOC
  http://linux.die.net/man/3/malloc

——岛本裕志

应用程序停止响应（死锁篇）

本 hack 介绍 pthread_mutex_lock 死锁导致的程序停止响应的调试方法。

死锁的例子

下面用示例代码（astall.c）讲解一下停止响应的典型情况——死锁的调试步骤。该程序在获得了锁（mutex）的状态下调用函数 cnt_reset()，在该函数中再次试图获得同一个锁，导致了死锁。这种 bug 经常出现在设计不完善的程序中。

[astall.c]

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int cnt = 0;
```

```
void cnt_reset(void)
{
   pthread_mutex_lock(&mutex);
   cnt = 0;
   pthread_mutex_unlock(&mutex);
}

void* thr(void *arg)
{
   while(1) {
      pthread_mutex_lock(&mutex);
      if (cnt > 2)
         cnt_reset();
      else
         cnt++;
      pthread_mutex_unlock(&mutex);

      printf("%d\n",cnt);
      sleep(1);
   }
}

int main(void)
{
   pthread_t tid;

   pthread_create(&tid, NULL, thr, NULL);
   pthread_join(tid, NULL);

   return EXIT_SUCCESS;
}
```

该程序在运行中会突然停止工作，也就是陷入停止响应的状态。执行上述程序的结果如下，显示到3之后停止响应。

```
$ ./astall
1
2
3
```

<<这里期待它显示0，但等待片刻后什么都不显示>>

172

停止响应时的解决方法

一般情况下，程序停止响应时要先确认是失去控制还是在等待，用 ps 命令可以确认。ps 命令的输出结果中，左起第 3 列为进程状态，如果这里为 R，表示进程仍在执行。也就是说，失控的可能性很大。如果是 S，说明在睡眠（sleep）。如果预期之外的睡眠持续很长时间，大多数情况是陷入了死锁。那么，上例的 astall 的运行状态如何呢？ps 命令的确认结果如下，状态为 S，可以怀疑是陷入了死锁。

```
$ ps ax -L | grep astall
 2365  2365 pts/4    Sl+    0:00 ./astall
 2365  2366 pts/4    Sl+    0:00 ./astall
```

选项-L 可以显示所有线程。

接下来用 GDB attach 到这个进程上，调查哪里在睡眠（sleep）。

```
$ gdb -p `pidof astall`
...
(gdb) bt
#0  0xb7f47430 in __kernel_vsyscall ()
#1  0xb7f18bf7 in pthread_join () from /lib/tls/i686/cmov/libpthread.so.0
#2  0x08048634 in main () at astall.c:35
```

启动之后只输入 bt 命令，显示的并不是运行停止的那个线程，而是最初启动的线程的 backtrace。该线程在 pthread_join()处睡眠，是期待的结果，所以我们切到另一个线程执行 bt 命令。

```
(gdb) i thr
  2 Thread 0xb7db1b90 (LWP 29894)  0xb7f47430 in __kernel_vsyscall ()
  1 Thread 0xb7db26b0 (LWP 29893)  0xb7f47430 in __kernel_vsyscall ()
(gdb) thr 2
[Switching to thread 2 (Thread 0xb7db1b90 (LWP 29894))]#0  0xb7f47430 in __kernel_vsyscall ()
(gdb) bt
#0  0xb7f47430 in __kernel_vsyscall ()
#1  0xb7f1ed09 in __lll_lock_wait () from /lib/tls/i686/cmov/libpthread.so.0
#2  0xb7f1a114 in _L_lock_89 () from /lib/tls/i686/cmov/libpthread.so.0
```

173

```
#3 0xb7f19a42 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0
#4 0x08048576 in cnt_reset () at astall.c:10
#5 0x080485af in thr (arg=0x0) at astall.c:20
#6 0xb7f1850f in start_thread () from /lib/tls/i686/cmov/libpthread.so.0
#7 0xb7e957ee in clone () from /lib/tls/i686/cmov/libc.so.6
```

可以看出，#3 的 pthread_mutex_lock()处进入内核模式并睡眠。也就是说，其他线程在别的地方用 pthread_mutex_lock()获取了锁的情况下再次调用 pthread_mutex_lock()的可能性很大。

从上述 backtrace 可见，该线程在 astall.c:10 行调用的 pthread_mutex_lock()导致了死锁，但是这种问题还要确定之前在哪里调用了 pthread_mutex_lock()。一般而言，死锁都发生在多次调用加锁、解锁操作后发生。因此，即使设置了断点，要想手动确认死锁发生之前的操作也是很困难的。所以我们采用下面的 GDB 命令文件，自动记录死锁发生之前的所有操作。该命令文件会在 pthread_mutex_lock()和 pthread_mutex_unlock()被调用时显示 backtrace，并将其内容记录到 debug.log 中。

```
set pagination off                                    ①
set logging file debug.log
set logging overwrite                                 ②
set logging on
start
set $addr1 = pthread_mutex_lock
set $addr2 = pthread_mutex_unlock                     ③
b *$addr1
b *$addr2
while 1
   c
   if $pc != $addr1 && $pc != $addr2
      quit                                            ④
   end
   bt
end
```

174 上述命令文件中的①的设置表示，即使显示的信息的行数超过了终端的行数，也不要暂停显示，而是继续显示下去。②的意思是将屏幕显示的内容记录到文件中，set logging overwrite 表示覆盖同名文件。③在 pthread_mutex_lock()和

pthread_mutex_unlock()上设置断点。此外，这些地址在后面的④处也要用到，所以保存到了变量$addr1 和$addr2 中，以方便使用。调试对象程序执行后会死锁，因此结束该程序要在键盘上输入 Ctrl-C 组合键。④就是中断程序运行时退出 GDB 的处理。

将上述命令文件保存为 debug.cmd，并用以下方式执行。

```
$ gdb astall -x debug.cmd
…
<<大约显示60行>>
…
Breakpoint 2, 0xb7fc39d0 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0
#0  0xb7fc39d0 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0
#1  0x08048576 in cnt_reset () at astall.c:10
#2  0x080485af in thr (arg=0x0) at astall.c:20
#3  0xb7fc250f in start_thread () from /lib/tls/i686/cmov/libpthread.so.0
#4  0xb7f3f7ee in clone () from /lib/tls/i686/cmov/libc.so.6
<<这里信息输出停止，用Ctrl-C结束程序>>
```

查看上述处理生成的日志 debug.log，就能确认怎样导致的死锁。但是，日志中包含大量的无用信息，看起来不太方便，可以像下面这样用 grep、sed 或 Ruby 等命令整理一下。

```
$ cat debug.log | grep -A1 "^#0.*pthread_mutex_" | sed s/from\ .*$// | sed s/.*\ in\ //
pthread_mutex_lock ()
_dl_addr ()
--
pthread_mutex_unlock ()
_dl_addr ()
--
pthread_mutex_lock ()
thr (arg=0x0) at astall.c:18
--
pthread_mutex_unlock ()
thr (arg=0x0) at astall.c:23
--
pthread_mutex_lock ()
thr (arg=0x0) at astall.c:18
--
```

175

```
pthread_mutex_unlock ()
thr (arg=0x0) at astall.c:23
--
pthread_mutex_lock ()
thr (arg=0x0) at astall.c:18
--
pthread_mutex_lock ()
cnt_reset () at astall.c:10
```

从中可见，最后 astall.c:18 加锁之后，astall.c:10 再次加锁。

使用多个 mutex 时的调试方法

前面的例子在显示加锁和解锁时中断执行并显示了 backtrace。如果使用了多个 mutex，那么单靠这种方法是不够的，必须明确操作是针对哪个 mutex 进行的。由于 pthread_mutex_lock()和 pthread_mutex_unlock()的参数就是 mutex 的地址，所以用**“HACK#10 函数调用时的参数传递方法（x86_64 篇）”**和**“HACK#11 函数调用时的参数传递方法（i386 篇）”**介绍的方法调查其参数，即可明白是针对哪个 mutex 的操作。例如，在 i386 的环境下，将下述命令插入到前面命令文件的 bt 命令前后即可。

```
printf "## addr: %08x\n", *(int*)($esp+4)
```

总结

本 hack 介绍了 pthread_mutex_lock 死锁引发的程序停止响应的调试方法。

——大和一洋

应用程序停止响应（死循环篇）

本 hack 以 tcpdump 中实际存在的 bug 为例，介绍用户应用程序的死循环的分析方法。

下面介绍一下在测试网络协议 SCTP 安全性时的 bug。

SCTP 是 SIGTRAN 工作组定义的、在 IP 上传输电话网控制信号（SS7）的协议，
176
与 TCP 同属传输层协议。

我们实施了安全测试中最常用的 0 长度包收发测试。测试程序比较复杂，书中就不再给出了。我们采用了 RAW socket，可以自由改变 SCTP 包的内容。

测试程序将 SCTP DATA chunk 的 length 成员设置为 0，并发送 SCTP 包，然后在另一台机器上执行 tcpdump 检查包的内容，以确认发送的包是否与预期一致。

编译 tcpdump

tcpdump 采用了社区原始的 3.8.2 版。这就是 RHEL4 包含的 tcpdump 的基础版本。

```
[@target]# wget -t0 -c http://www.tcpdump.org/release/tcpdump-3.8.2.tar.gz
[@target]# tar zxvf tcpdump-3.8.2.tar.gz
[@target]# cd tcpdump-3.8.2
[@target tcpdump-3.8.2]# ./configure ; make
```

有些环境下会产生以下错误，只需修改 config.h 重新 make 即可（RHEL4 的 RPM 包已做了该修改，以免发生该错误）。

```
# make
tcpdump.o(.text+0x894): In function `main':
: undefined reference to `pcap_debug'
collect2: ld returned 1 exit status
make: *** [tcpdump] Error 1
# vi config.h
...
/* define if libpcap has pcap_debug */
//#define HAVE_PCAP_DEBUG 1       /* 注释掉该行 */
...
# make clean; make
```

检查包内容

检查包内容需要执行 tcpdump。

为了让 tcpdump 显示详细信息，习惯性地添加了-vvvX 选项。结果如下所示，出现了大量重复的输出，行为发生了异常。

177

```
[@target tcpdump-3.8.2]# ./tcpdump -ieth2 -vvvX sctp
...
```

```
00:21:12.747262 IP (tos 0x2,ECT(0), ttl 64, id 0, offset 0, flags [DF], proto
132, length: 56) 192.168.0.145.56934 > 192.168.0.155.49560: sctp
 1) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
 2) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
 3) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
 4) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
...
 2764) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
 2765) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
 2766) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
 2767) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
...
```

持续输出的大量“数字) [DATA] (B)(E)……”看上去像是循环。用 Ctrl-C 无法结束程序，只好用 Ctrl-Z 终止进程。

```
[1]+ Stopped          ./tcpdump -ieth2 -vvvX sctp
```

然后用 kill -9 将进程结束掉。

本次测试的前提是 length 为 0，恐怕这就是原因。查看源代码中对 SCTP DATA chunk 的 length 的处理部分应该就能明白，不过这里我们先不看代码，首先缩小调查范围。

确认不同选项下行为是否有变化

改变 tcpdump 的选项确认一下，结果只有-v 选项时工作正常，-vv 和-vvv 也能正常工作，但输出了下述信息。

```
[@target tcpdump-3.8.2]# ./tcpdump -ieth2 -vvv sctp
...
00:21:41.678060 IP (tos 0x2,ECT(0), ttl 64, id 0, offset 0, flags [DF], proto
132, length: 56) 192.168.0.145.13727 > 192.168.0.155.59671: sctp
     1) [DATA] (B)(E) [TSN: 2466762381] [SID: 0] [SSEQ 0] [PPID 0x0]
[Payload:bogus chunk length 0]
...
```

DATA chunk 的 length 输出了 0。这表明，测试程序发送了预期的 SCTP 包。

178

然后确认-X、-vX，现象没有发生，但设置-vvX 选项时问题复现了。这个信息在调

查源代码时十分有用。

选择断点

下面用 GDB 选择设置断点的地方。在 main()中设置断点也没问题，但分析不太方便，因此我们将断点设置在可能发生循环的地方。

```
[tcpdump-3.8.2/print-sctp.c]
 62 void sctp_print(
   ...
128  if (vflag < 2)      /* -v 的数量少于 2 时，直接 return */
129    return;           /* 因此在下面设置断点 */
     ...
156    case SCTP_DATA :  /* 在 DATA chunk 的情况下进入下面的处理 */
       ...
160     printf("[DATA] ");  /* 断点 */
       ...
196     printf("[Payload");

198     if (!xflag && !qflag) {  /* 如果没有-X 选项，则执行下述处理，*/
       ...                 /* 因此断点不能设置在这个 if 语句中 */
204           printf("bogus chunk length %u]",
205              htons(chunkDescPtr->chunkLength));
206           return;
207        }
       ...
212     } else
213        printf("]");
214      }
215     break;
   ...
```

由于连续输出[DATA]，因此我们将断点设置在 print-sctp.c 的 160 行。

给 tcpdump 加上调试选项重新编译，并复现 bug（下面省略了部分步骤）。

179

```
[@target tcpdump-3.8.2]# ./configure CFLAGS=-g
...
[@target tcpdump-3.8.2]# gdb ./tcpdump
...
```

```
(gdb) b print-sctp.c:160
Breakpoint 1 at 0x42d60b: file ./print-sctp.c, line 160.
(gdb) run -ieth2 -vvX sctp
Starting program: /root/nooiwa/tcpdump-3.8.2/tcpdump -ieth2 sctp -vvX
...
Breakpoint 1, sctp_print (bp=0x6da324 "K\210<\220 \b, bp2=Variable "bp2" is not
available.
) at ./print-sctp.c:160
160             printf("[DATA] ");
(gdb)
```

现象复现之后，用 gdb 在断点处暂停执行。

单步执行确认现象

单步执行时发现，同样的代码被反复执行，造成了循环。检查一下是不是 DATA chunk 的 length 的原因。

```
(gdb) s
...
145         chunkEnd = ((const u_char*)chunkDescPtr + EXTRACT_16BITS(&chunkDescPtr->
chunkLength));
(gdb) p chunkDescPtr
$1 = (const struct sctpChunkDesc *) 0x6da330
(gdb) p chunkDescPtr->chunkLength
$2 = 0
(gdb) s
...
151        nextChunk = (const void *) (chunkEnd + align);
(gdb) p nextChunk
$3 = (const void *) 0x6da330
(gdb) p chunkEnd
$4 = (const u_char *) 0x6da330 ""
(gdb) s
```

SCTP 包结构

SCTP 包结构如图 4-5 所示。

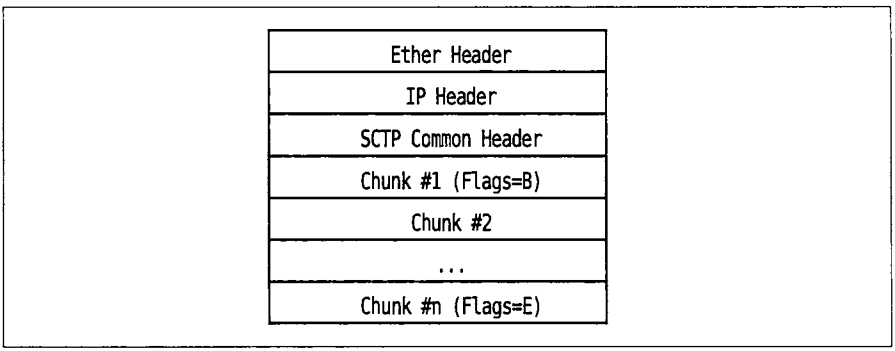


图 4-5 SCTP 包格式

SCTP 由两种数据构成：公共头部和 chunk。chunk 有多种，如果是 DATA chunk，SCTP 会检查 chunk 的长度，尽可能在包内放入多个 chunk。

Type 表明该 chunk 是个 DATA chunk，length 指示 DATA chunk 的长度。tcpdump 首先查看 Chunk#1，接下来要查看 Chunk #2，但指向 Chunk#1 的指针只前进了 length。指针前进 length 本应指向 Chunk#2 的开头，但这次 length 为 0，因此仍然留在 Chunk#1 的开头。

因而陷入了死循环。大量的输出信息“[数字) [DATA] (B)(E)…”中，数字为包中的 chunk 数量，(B)(E)如图 4-6 所示为 Flags，分别为第 1 个 chunk “(B)eginning”和最后一个 chunk“(E)nding”。这表示包中该 chunk 为第 1 个 chunk，也是最后一个 chunk，即不存在下一个 chunk（Chunk#2）。

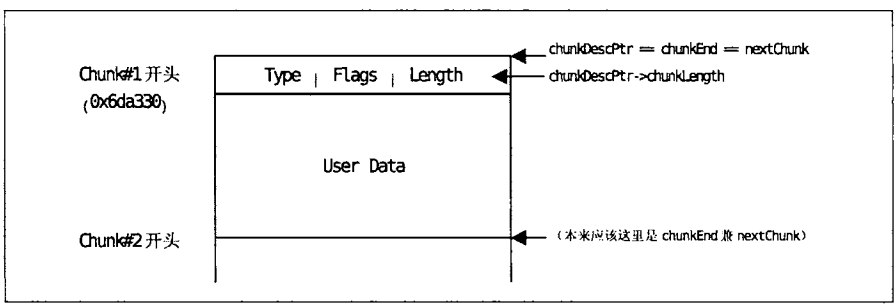


图 4-6 SCTP DATA chunk 格式

确认更高的版本

原因已经明确了，而修正方法也考虑了很多种。

181

更高版本中该 bug 有可能已经被修正，因此用 tcpdump 3.9.8 试了一下，可以正常运行了。

```
[@target tcpdump-3.9.8]# ./tcpdump -ieth2 -vvX sctp
...
00:16:51.581458 IP (tos 0x2,ECT(0), ttl 64, id 0, offset 0, flags [DF], proto
SCTP (132), length 56) 192.168.0.145.51816 > 192.168.0.155.10295: sctp
      1) [Bad chunk length 0]
      0x0000:  4502 0038 0000 4000 4084 b7c3 c0a8 0091  E..8..@.@.......
...
```

如果想确认补丁内容，可以到 tcpdump 项目的 cvs 上查看。它提供了匿名 cvs，也给出了 checkout 的方法。

```
# cvs -d :pserver:tcpdump@cvs.tcpdump.org:/tcpdump/master login
Logging in to :pserver:tcpdump@cvs.tcpdump.org:2401/tcpdump/master
CVS password:                              <- 输入 anoncvs
# cvs -d :pserver:tcpdump@cvs.tcpdumo.org:/tcpdump/master checkout tcpdump
cvs checkout: Updating tcpdump
...
```

可以从 log 或 diff 的内容中查看补丁。本次 bug 的修改可以用以下命令查看。

```
# cd tcpdump/
# cvs log print-sctp.c
# cvs diff -N -u -p -r 1.17 -r1.18 print-sctp.c
```

总结

本 hack 以 tcpdump 为例，介绍了用户应用程序死循环的分析方法。首先确定范围，然后在适当的地方设置断点，就能轻松地解决掉问题。

这次的例子输出了大量相同的信息，因此能立即注意到死循环。而有时也有什么信息也不输出，只是不返回到提示符的情况，此时可以用 top 命令、vmstat 命令确认 CPU 使用率等。

参考文献

- tcpdump/libpcap 项目

http://www.tcpdump.org/

182

- [RFC2960] Stream Control Transmission Protocol
  http://www.ietf.org/rfc/rfc2960.txt

——大岩尚宏

关于 SCTP 协议

SCTP 并不是传输 SS7 的专用协议，还可以作为普通协议使用。与 TCP 相比，它有以下特长。

- SCTP 拥有 TCP、UDP 两种风格，可灵活应用于可靠性优先的通信和传输效率优先的通信。
- UDP 风格在同一端口下，可以用一个 socket 管理多个连接。
- 用 socket 管理多个 IP 地址，或者动态添加或删除 IP 地址（路径），支持多重主目录（multihome）环境。除主通信 IP 地址外，协议内实现了用 heartbeat 进行监视，一旦主 IP 地址无法通信，就能自动切换到其他路径。
- 连接切断等事件，由 SCTP 以异步方式通知用户应用程序。TCP 中为了实现导通确认，必须发送数据包，而 SCTP 则不需要这种监视。

第5章

实践内核调试

hack #33~#42

本章介绍内核故障的调试方法，包括 kernel panic（NULL 指针引用、链表破坏、竞态条件等）、内核失去响应（kernel stall，包括死循环、自旋锁、信号量、实时进程等）、运行缓慢、CPU 负载过高等非正常状况的调试方法。

kernel panic（空指针引用篇）

本 hack 以实际存在的导致内核产生 Oops 的问题为例，介绍内核调试的方法。

我们在使用 LTP（Linux Test Project）的测试程序进行 IPv6 网络负载测试时，发生了 kernel panic。当时控制台输出了如下信息。

```
Unable to handle kernel NULL pointer dereference at 0000000000000160 RIP:
<ffffffffa0130ac2>{:ipv6:icmpv6_ send+1235}
PML4 226fd067 PGD 0
Oops: 0000 [1] SMP
CPU 0
Modules linked in: ah6 deflate twofish serpent aes blowfish des sha256 crypto_null af_key
i2c_dev i2c_core 8021q md5 ipv6 ide_dump scsi_dump diskdump zlib_deflate dm_mirror
dm_multipath dm_mod button battery ac joydev uhci_hcd ehci_hcd hw_random tg3 e100 mii
floppy ext3 jbd ata_piix libata sd_mod scsi_mod
Pid: 6574, comm: ping6 Not tainted 2.6.9-prep
RIP: 0010:[<ffffffffa0130ac2>] <ffffffffa0130ac2>{:ipv6:icmpv6_send+1235}
RSP: 0018:00000100351f3918 EFLAGS: 00010216
RAX: ffffffffa0158eb0 RBX: 0000000000000000 RCX: 0000000000000001
RDX: 000000000000060c RSI: 000001002808487f RDI: ffffffffa0158eb0
RBP: 00000000000004d0 R08: 00000100351f37d8 R09: 0000000000000002
```

```
R10: 0000000000000000 R11: 0100000000000000 R12: 0000010032cbd180
R13: 0000000000000040 R14: 000001001f76a058 R15: 0000010026a72c00
FS:  0000002a959b5e20(0000) GS:ffffffff8050cf00(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: 0000000000000160 CR3: 0000000000101000 CR4: 00000000000006e0
Process ping6 (pid: 6574, threadinfo 00000100351f2000, task 000001003e4a2030)
Stack: 00000000000004d0 0000000000000202 0000000000000000 0000010026a72f58
       0000000200000000 00000100370c3d00 9c05000000000002 003a00000000007c
       0000010032cbd180 0000010000000000
Call Trace:<ffffffffa013f56f>{:ipv6:xfrm6_output+135}
       <ffffffff802cfcdd>{ip_generic_getfrag+66}
       <ffffffffa011de71>{:ipv6:ip6_push_pending_frames+798}
       <ffffffffa012fd00>{:ipv6:rawv6_sendmsg+2324}
       <ffffffffa0048106>{:jbd:journal_dirty_metadata+391}
       <ffffffff802a9883>{sock_sendmsg+271}
       <ffffffff801381c8>{release_console_sem+369}
       <ffffffff80238bc8>{do_con_write+7903}
       <ffffffff8022f2b6>{vt_ioctl+61}
       <ffffffff80227a41>{tty_ldisc_try+60}
       <ffffffff8013560e>{autoremove_wake_function+0}
       <ffffffff802ab1f3>{sys_sendmsg+463}
       <ffffffff8012370f>{do_page_fault+575}
       <ffffffff802ac381>{release_sock+16}
       <ffffffff8018baf8>{sys_ioctl+924}
       <ffffffff8011026a>{system_call+126}

Code: 48 8b 9b 60 01 00 00 48 85 db 74 07 f0 ff 83 d0 00 00 00 48
RIP <ffffffffa0130ac2>{:ipv6:icmpv6_send+1235} RSP <00000100351f3918>
CR2: 0000000000000160
```

184

这段信息第1行指出，是NULL指针访问（实际地址是0x160）导致的Oops发生。Oops信息的阅读方法请参见**“HACK#15 Oops信息的解读方法”**。

建立复现程序

继续调查发现，只要执行LTP中的特定测试，就能100%地发生Oops。因此，我们将这个测试的内容进一步简化，利用下面的脚本成功复现。

```
$ cat reproducer.sh
```

```
#!/bin/sh                                                          185
IFACE=$1
CONTENT="add fd00:1:1:1::2 fd00:1:1:1::1 ah 1000 -m tunnel \
        -A hmac-sha1 \"beef_fish_pork_salad\" ; \
        spdadd fd00:1:1:1::2 fd00:1:1:1::1 any -P out \
        ipsec ah/tunnel/fd00:1:1:1::2-fd00:1:1:1::1/use ; \
        spdadd fd00:1:1:1::1 fd00:1:1:1::2 any -P in \
        ipsec ah/tunnel/fd00:1:1:1::1-fd00:1:1:1::2/use ;"
ip -f inet6 addr add fd00:1:1:1::2/64 dev $IFACE
echo $CONTENT | setkey -c
sleep 5
ping6 -I $IFACE fd00:1:1:1::1 -c 1 -s 1500
```

这段脚本创建 IPsec 隧道，并在其中使用 ping6 命令发送比 IPsec 隧道的 MTU 更大的包。如下所示，指定网络接口执行该脚本就会发生 Oops。脚本执行需要超级用户权限。

```
$ sudo ./reproducer.sh eth1
```

用内核转储查看哪里发生了 NULL 指针访问

该问题产生了 Oops，因此采用了获取内核崩溃转储并进行调试的办法。首先从开头的 Oops 信息中可以得知，执行 icmpv6_send()函数的偏移量 1235 字节时发生了 panic，因此先对该函数进行反汇编，看看究竟发生了什么事。

```
crash> disas icmpv6_send
No symbol "icmpv6_send" in current context.
```

输入 crash 命令后发生了错误，说找不到符号。icmpv6_send()是在 ipv6.ko 这个可加载模块中定义的，因此不加载该模块就无法显示。

```
crash> mod -s ipv6
     MODULE       NAME   SIZE    OBJECT FILE
ffffffffa015d1B0 ipv6   284512  /lib/modules/2.6.9-prep/kernel/net/ipv6/ipv6.ko
```

这样就能看到 icmpv6_send()了。

```
crash> disas icmpv6_send                                           186
Dump of assembler code for function icmpv6_send:
0xffffffffa01305ef <icmpv6_send+0>:    push   %r15
0xffffffffa01305f1 <icmpv6_send+2>:    mov    $0xffffffff80543930,%rax
```

```
...
0xffffffffa0130a91 <icmpv6_send+1186>:  callq  0xffffffff802ba11e <net_ratelimit> —①
0xffffffffa0130a96 <icmpv6_send+1191>:  test   %eax,%eax
0xffffffffa0130a98 <icmpv6_send+1193>:  je     0xffffffffa0130bef <icmpv6_send+1536>
0xffffffffa0130a9e <icmpv6_send+1199>:  mov    $0xffffffffa01413ba,%rdi
0xffffffffa0130aa5 <icmpv6_send+1206>:  xor    %eax,%eax
0xffffffffa0130aa7 <icmpv6_send+1208>:  callq  0xffffffff80138413 <printk>
0xffffffffa0130aac <icmpv6_send+1213>:  jmpq   0xffffffffa0130bef <icmpv6_send+1536>
0xffffffffa0130ab1 <icmpv6_send+1218>:  mov    0x30(%r12),%rbx
0xffffffffa0130ab6 <icmpv6_send+1223>:  mov    $0xffffffffa0158eb0,%rdi ————②
0xffffffffa0130abd <icmpv6_send+1230>:  callq  0xffffffff8030e7d0 <_read_lock> ——③
0xffffffffa0130ac2 <icmpv6_send+1235>:  mov    0x160(%rbx),%rbx ————————④
0xffffffffa0130ac9 <icmpv6_send+1242>:  test   %rbx,%rbx
0xffffffffa0130acc <icmpv6_send+1245>:  je     0xffffffffa0130ad5 <icmpv6_send+1254>
0xffffffffa0130ace <icmpv6_send+1247>:  lock   incl 0xd0(%rbx)
...
```

问题发生在 icmpv6_send+1235，是④的位置。这条指令从 RBX 寄存器表示的地址处偏移 0x160 字节，将该位置的内容保存到 RBX 寄存器中。从 Oops 消息中的寄存器转储内容可知，RBX 为 0，因此④的指令写成 C 语言就是

```
RBX = *(0x160 + 0x0)
```

这就是 Oops 的原因。那么我们来看看这个位置在内核源代码上对应哪一部分，要点位于①调用的 net_ratelimit()函数。这里 net_ratelimit()被替换成 LIMIT_NETDEBUG()宏，在 icmpv6_send()内有 3 个地方调用了 LIMIT_NETDEBUG()。

```
[include/net/sock.h]
#define LIMIT_NETDEBUG(x) do { if (net_ratelimit()) { x; } } while (0)
```

187

net_ratelimit()是内核的网络栈代码中经常使用的函数，可以像下面这样与printk()配合使用显示信息。

```
if (net_ratelimit())
  printk(...);
```

用 printk()显示信息需要消耗一定的开销。网络代码可能成为外部恶意用户的 DoS 攻击对象，因此不能每次必定执行 printk()，而是要将其控制在某种频率之下，其原理就是 net_ratelimit()。在内核网络代码中，阅读反汇编后的代码时以 net_ratelimit()为标记，就十分易懂了。

另一个需要注意的地方是③处的_read_lock()。前面的②处设置了传递给

read_lock()的参数，用crash命令查看一下这个参数是什么。

```
crash> sym 0xffffffffa0158eb0
ffffffffa0158eb0 (d) addrconf_lock
```

也就是说，③处执行的代码是read_lock(&addrconf_lock);。

总结以上内容可知，LIMIT_NETDEBUG()调用附近有个 read_lock (&addrconf_lock)，这样就能知道位于下面的位置。

```
[net/ipv6/icmp.c]
void icmpv6_send(struct sk_buff *skb, int type, int code, _ _u32 info,
            struct net_device *dev)
{
...
      if (len < 0) {
            LIMIT_NETDEBUG(————————————————————⑤
                  printk(KERN_DEBUG "icmp: len problem\n"));
            goto out_dst_release;
      }

      idev = in6_dev_get(skb->dev);
...
```

与反汇编结果的①对应的就是⑤。看看in6_dev_get()就会发现正在寻找read_lock()。该函数被声明为 inline，因此在反汇编代码上并不表现为函数调用，而是在icmpv6_send()函数内直接展开。 188

```
[include/net/addrconf.h]
static inline struct inet6_dev *
in6_dev_get(struct net_device *dev)
{
      struct inet6_dev *idev = NULL;
      read_lock(&addrconf_lock); ——————————————⑥
      idev = dev->ip6_ptr; ————————————————————⑦
      if (idev)
            atomic_inc(&idev->refcnt);
      read_unlock(&addrconf_lock);
      return idev;
}
```

可知与反汇编结果的③对应的地方是⑥。另外，我们看看⑦用到的 net_device 结构的 ip6_ptr 成员的偏移量。

```
crash> hex
output radix: 16(hex)
crash> struct -o net_device.ip6_ptr
struct net_device {
  [0x160] void *ip6_ptr;
}
```

偏移量为 0x160 字节。就是它，Oops 发生的地方，也就是说反汇编中的④对应的地方就是这里的⑦。

```
mov 0x160(%rbx),%rbx
```

偏移量 0x160 是 net_device 结构的成员 ip6_ptr，那么 RBX 指向的地址就是 in6_dev_get()的参数 dev。Oops 信息的寄存器转储中，RBX 值为 0，可知问题发生时 dev 指针为 NULL。传递给 in6_dev_get()的参数为 skb->dev，因此 skb->dev 为 NULL 就是直接原因。

根据源代码调查处理内容

进一步进行源代码跟踪，可知问题发生时的调用流程如图 5-1 所示。

189

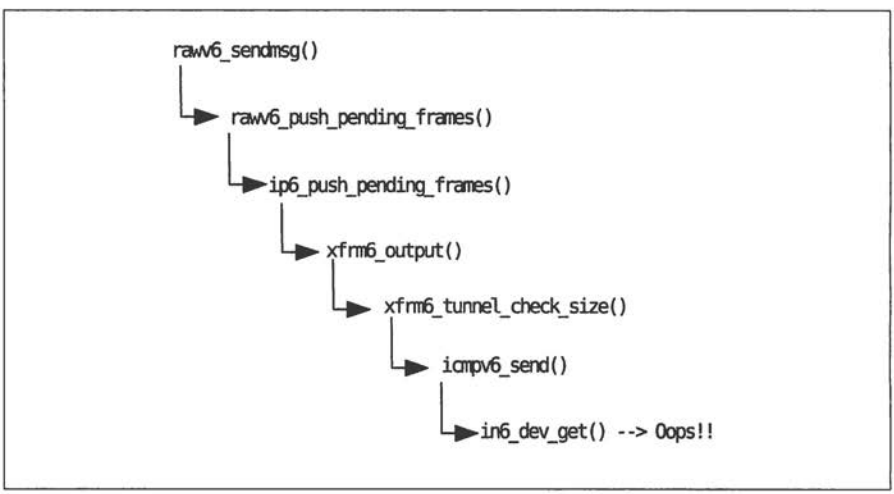


图 5-1 Oops 发生之前的调用流程

阅读xfrm6_tunnel_check_size()函数就能明白，由于试图发送的包长度超过了IPsec隧道的MTU，因此它返回了ICMPv6包过大的错误。

```
[net/ipv6/xfrm6_output.c]
static int xfrm6_tunnel_check_size(struct sk_buff *skb)
{
...
        if (skb->len > mtu) {
        /* 如果包长度超过IPsec隧道的MTU，就返回ICMP错误 */
                icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, skb->dev);
                ret = -EMSGSIZE;
        }
...
}
```

总结一下，ping6命令后调用了icmpv6_send()，但表示icmpv6_send()试图送信的设备skb->dev为NULL，导致Oops发生。再进一步，此时正在试图向IPsec隧道发包，正在建立socket缓冲区（skb），因此skb->dev还未设置，仍然处于NULL的状态。超出MTU大小就应该给发包方返回错误，这段代码似乎漏掉了这一点。

检查社区的修改历史

像Linux内核等开源软件的bug通常已被社区修改了，因此在Linus Torvals的git树上搜索相关修改，就发现了下面这段补丁。该问题已被社区发现，在内核版本2.6.12中改正。

190

```
commit 180e42503300629692b513daeb55a6bb0b51500c
Author: Herbert Xu <herbert@gondor.apana.org.au>
Date:  Mon May 23 13:11:07 2005 -0700

    [IPV6]: Fix xfrm tunnel oops with large packets

    Signed-off-by: Herbert Xu <herbert@gondor.apana.org.au>
    Acked-by: Hideaki YOSHIFUJI <yoshfuji@linux-ipv6.org>
    Signed-off-by: David S. Miller <davem@davemloft.net>

diff --git a/net/ipv6/xfrm6_output.c b/net/ipv6/xfrm6_output.c
index 601a148..6b98677 100644
```

```
--- a/net/ipv6/xfrm6_output.c
+++ b/net/ipv6/xfrm6_output.c
@@ -84,6 +84,7 @@ static int xfrm6_tunnel_check_size(struct sk_buff *skb)
            mtu = IPV6_MIN_MTU;

      if (skb->len > mtu) {
+           skb->dev = dst->dev;
            icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, skb->dev);
            ret = -EMSGSIZE;
      }
```

总结

本 hack 通过实际发生的 NULL 指针导致的 Oops 的典型范例，介绍了根据内核崩溃转储进行调试的方法。从中可见，简化复现程序是非常重要的。复现程序越简单，分析范围就越小，解决问题就越快。

参考文献

- Linux Test Project
  http://ltp.sourceforge.net/
- Linus Torvals 的 git 树
  http://git.kernel.org/?p=linux/kernel/git/torvals/linux-2.6.git;a=summary

——安部东洋

191

kernel panic（链表破坏篇）

本 hack 介绍利用测试程序调试链表破坏的方法。

我们写了个测试程序，以复现之前遇到过的链表破坏，该 bug 在 SMP 环境下发生。本 hack 以此为基础介绍一下链表破坏，操作系统为 2.6.9 内核的发行版。

内核的链表

复现用测试程序使用的链表为内核中实现的双向链表 `list_head`，这是最简单、最普通的链表。它适合小规模且简单的管理，用于文件系统、内存管理和网络等。

复现测试程序包含了 include/linux/list.h 头文件，其中定义了链表操作函数和宏。表 5-1 列出了代表性的函数和宏。

表 5-1 链表操作函数和宏

链表操作函数和宏	内容
list_add	向开头添加一项
list_del	从链表中删除一项
list_empty	检查链表是否为空
list_entry	获取链表项（由链表指针获取链表项的结构）
list_for_each	遍历链表项

复现测试程序的内容

复现程序是个内核模块，下面是部分代码。操作链表的 3 个内核线程通过模块的 insmod 启动[注1]。

第 1 个线程 list_add_thread()向链表中添加 10000 个链表项。为复现 bug，需要尽量多地添加项目，所以这里添加了 10000 个。

```
#include <linux/list.h>
static int list_add_thread(void *data)
{
   int i;
   do {
      spin_lock(&trouble.lock); ——————————————————————①
      for (i = 0; i < 10000; i++) {     /* 创建10000个链表项 */
         struct k_entry *entry;

         entry = kmalloc(sizeof(struct k_entry),GFP_ATOMIC);
         INIT_LIST_HEAD(&entry->list);   /* 初始化链表 */
         list_add(&entry->list, &trouble.list); /* 添加项目到trouble链表 */
      }
      spin_unlock(&trouble.lock); ——————————————————————②
      msleep(200);
   } while (!kthread_should_stop());
```

192

注 1：测试程序可以在本书的支持页面（http://www.oreilly.co.jp/books/9784873114040/）上下载。

```
    return 0;
}
```

第2个线程list_release_thread()将所有非空链表中的链表项删除。

```
static int list_release_thread(void *data) ——————————————— ③
{
    do {
        spin_lock(&trouble.lock);
        while (!list_empty(&trouble.list)){  /* 检查链表是否为空 */
            struct k_entry *entry;

            entry = list_entry(trouble.list.next, struct k_entry, list);
            list_del(&entry->list);          /* 从链表中取出项 */
            kfree(entry);                    /* 释放链表项 */
        }
        spin_unlock(&trouble.lock); ——————————————————————— ④
        msleep(100);
    } while (!kthread_should_stop());

    return 0;
}
```

第3个线程list_del_thread()在链表非空时删除一个链表项。

193

```
static int list_del_thread(void *data)
{
    do {
        if (!list_empty(&trouble.list)){  /* 确认链表是否为空 /* ——————————⑤
            struct k_entry *entry;

            spin_lock(&trouble.lock); ——————————————————————⑥
            entry = list_entry(trouble.list.next, struct k_entry, list);
            list_del(&entry->list);
            kfree(entry);
            spin_unlock(&trouble.lock); ——————————————————————⑦
        }
        msleep(1);
    } while (!kthread_should_stop());
```

```
    return 0;
}
```

为了能执行 rmmod，最后放了个 kthread_should_stop()。

这段代码用 spin_lock()保护链表，看似没有问题，但一 insmod 该模块就会 panic 并产生转储。

分析转储

下面是获取的转储的 backtrace。

```
Unable to handle kernel paging request at 0000000000100108 RIP: ———————— ⑧
<ffffffff80162bb6>{kfree+168}
PML4 0
Oops: 0000 [1] SMP
CPU 1
Modules linked in: trouble_list …   /* 复现测试程序的模块*/
...
Pid: 4825, comm: list_del Not tainted 2.6.9
RIP: 0010:[<ffffffff80162bb6>] <ffffffff80162bb6>{kfree+168}
RSP: 0018:000001007c753ee8 EFLAGS: 00010002
RAX: 0000000000000001 RBX: 0000000000000000 RCX: 000001000000c000
RDX: 0000000000000000 RSI: 0000000000000008 RDI: 0000000000100100
RBP: ffffffffa0196d40 R08: 0000000000000003 R09: 0000000000000040
R10: 0000000000000001 R11: 0000000000000001 R12: 0000010061ed7e18
R13: 00000000fffffffc R14: 0000010061ed7e08 R15: ffffffff8014b4f8
FS:  0000000000000000(0000) GS:ffffffff8050d300(0000) knlGS:0000000000000000
CS:  0010 DS: 0018 ES: 0018 CR0: 000000008005003b
CR2: 0000000000100108 CR3: 00000000016e8000 CR4: 00000000000006e0
Process list_del (pid: 4825, threadinfo 000001007c752000, task 0000010059f9a7f0)
Stack: 0000000000000006 0000000000000282 0000000000000000 ffffffffa01960e0
       0000010061ed7e18 ffffffffa0196123 0000000000000000 ffffffff8014b4cf
       ffffffffffffffff 0000010061ed7e08
Call Trace:<ffffffffa01960e0>{:trouble_list:list_del_thread+0}
       <ffffffffa0196123>{:trouble_list:list_del_thread+67}
       <ffffffff8014b4cf>{kthread+200} <ffffffff80110f53>{child_rip+8}
       <ffffffff8014b4f8>{keventd_create_kthread+0} <ffffffff8014b407>{kthread+0}
       <ffffffff80110f4b>{child_rip+0}
```

194

通常要根据反汇编的结果检查 kfree()+168 的位置，看看它是在访问什么变量时发生的 panic。但这次我们查看访问指针地址⑧，这个地址有点特别，是 100108(0x100100+0x8)。这就是 LIST_POISON。

LIST_POISON

Linux 的 2.6 内核中，list_del()不会将链表项的 next、prev 成员置成 NULL，而是赋为 LIST_POISON。

```
[include/linux/list.h]
#define LIST_POISON1  ((void *) 0x00100100)
#define LIST_POISON2  ((void *) 0x00200200)

static inline void list_del(struct list_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = LIST_POISON1;
        entry->prev = LIST_POISON2;
}
```

这是为了在错误地访问 list_del()删除过的链表项或使用未初始化的链表项时，能够发现这种问题，而故意进行了污染。

⑧的 100108 的原因是使用了 kfree()内 trouble.list.next(0x100100)的偏移量 0x8。

195

链表破坏的原理

这个 bug 的原理如图 5-2 所示。

链表为空时，用 list_entry()得到的链表项为 0x100100，然后把它当做用 kmalloc()获得的正常链表项使用，会引发非法访问。

正常链表和本例的链表如图 5-3 所示。

list_release_thread()处理结束后，链表为空。panic 时，也就是说链表为空的状态下执行 list_del_thread()时，由于试图从空链表中强行获取链表项并执行 list_del()，导致链表的 next 和 prev 被 LIST_POISON 污染。

下面是在该转储中访问全局的 trouble 链表的结果。同样是非法访问。

```
crash> struct list_head trouble
struct list_head {
  next = 0x100100,
  prev = 0x200200
}
crash> list trouble
ffffffffa0196d40
100100
list: invalid kernel virtual address: 100100 type: "list entry"
crash>
```

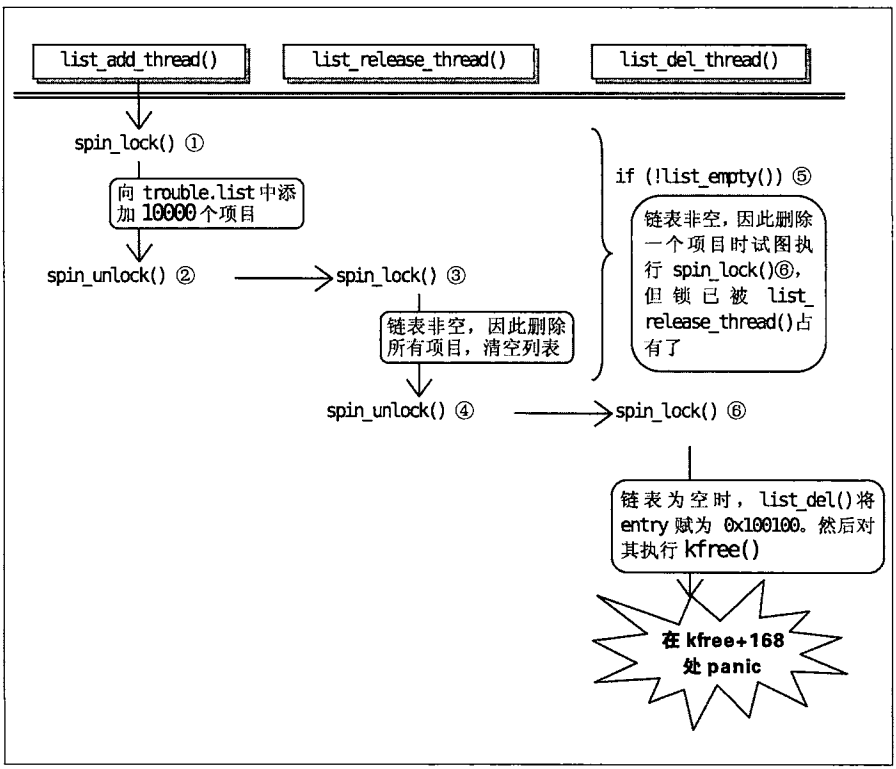


图 5-2 链表破坏的原理（1）

196

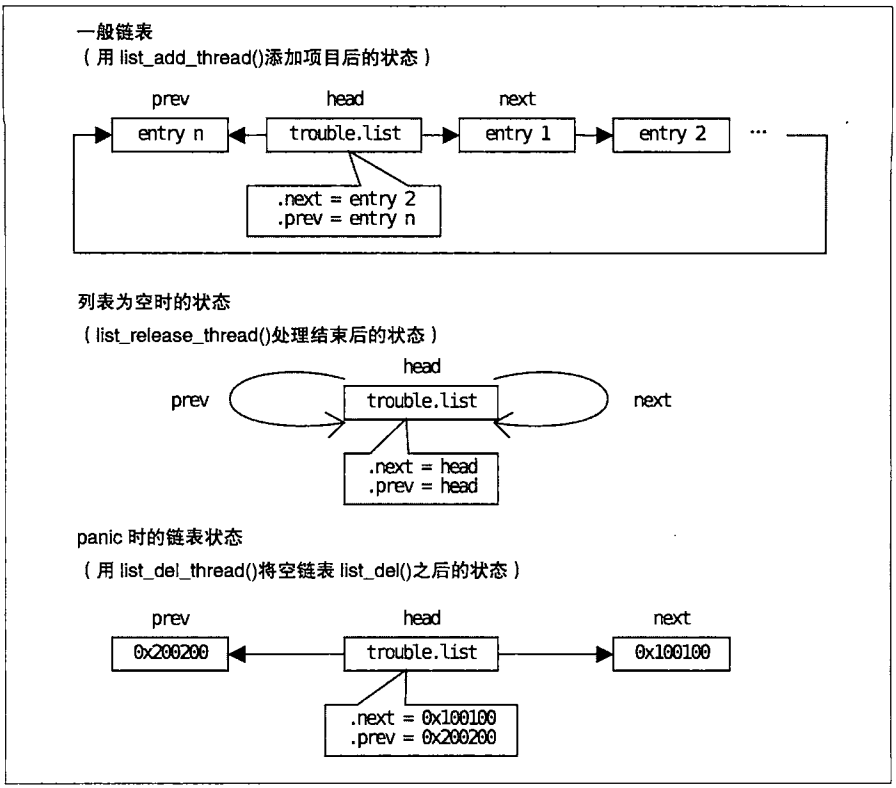


图 5-3 链表破坏的原理（2）

Debug memory allocations

刚才用复现测试程序弄清楚了原因，但实际的 bug 是很困难的。list_del()造成的链表破坏可以通过 LIST_POISON 发现，除此之外，还有个 Debug memory allocations 调试功能，可以用于检测已分配的内存区域和双重释放等。kmalloc()/kfree()可以检查已分配的内存（slab cache）。

要使用该功能，需要在 make menuconfig 中启用 Kernel hacking -> Debug memory allocations(CONFIG_DEBUG_SLAB=y)，再重新编译内核。

下面是在启用了 Debug memory allocations 的内核上执行复现程序的结果。BUG()引发了 panic，在此之前输出了信息。

197

```
kfree_debugcheck: bad ptr ffffffffa0196d40h. ——————————————— ⑨
```

```
---------- [cut here ] --------- [please bite here ] ---------
Kernel BUG at slab:1862                /* 意思是kernel/slab.c的1862行*/
invalid operand: 0000 [1] SMP
CPU 0
Modules linked in: trouble_list ...
...
Pid: 4380, comm: list_del Not tainted 2.6.9-42.28AXinode
RIP: 0010:[<ffffffff80162b3a>] <ffffffff80162b3a>{kfree_debugcheck+436}
RSP: 0018:0000010077eafed8 EFLAGS: 00010012
RAX: 0000000000000030 RBX: ffffffffa0196d40 RCX: ffffffff803e8d68
...
Call Trace:<ffffffff80163833>{kfree+29} <ffffffffa01960e0>{:trouble_list:list_de
l_thread+0}
     <ffffffffa0196132>{:trouble_list:list_del_thread+82}
     <ffffffff8014b4cf>{kthread+200} <ffffffff80110f53>{child_rip+8}
     <ffffffff8014b4f8>{keventd_create_kthread+0} <ffffffff8014b407>{kthread+0
}
     <ffffffff80110f4b>{child_rip+0}
...
```

kfree_debugcheck()是个内存检查函数，只在 CONFIG_DEBUG_SLAB=y 时由 kfree()调用。⑨的指针是 trouble 变量的地址。该消息的意思是，slab cache 之外（trouble 是个全局变量，不是 kmalloc()或 kmem_cache_alloc()分配的内存）的内存区域被访问了。

链表破坏的修改方法

本 hack 讨论的链表破坏可以用互斥处理解决。链表是否为空的检查也应当在 spin_lock()之内进行。下面是补丁。

```
--- trouble_list.c       2009-01-14 23:18:27.000000000 +0900
+++ trouble_list.c.new   2009-01-14 23:19:59.000000000 +0900
@@ -63,15 +63,15 @@ static int list_release_thread(void *dat
 static int list_del_thread(void *data)
 {
      do {
+          spin_lock(&trouble.lock);
           if (!list_empty(&trouble.list)){
                struct k_entry *entry;
```

198

```
-                spin_lock(&trouble.lock);
                 entry = list_entry(trouble.list.next, struct k_entry, list);
                 list_del(&entry->list);
                 kfree(entry);
-                spin_unlock(&trouble.lock);
             }
+            spin_unlock(&trouble.lock);
             msleep(1);
         } while (!kthread_should_stop());
```

应用该补丁之后就不再发生 panic，可以正常运行了。

总结

本 hack 介绍了链表破坏导致的 panic。只要出现 0x100100 或者 0x200200，就可以猜想产生了链表破坏。

LIST_POISON 由 list_del()设置，而 list_del_init()会将链表项初始化为 NULL，不会赋为 LIST_POISON。因此，该方法虽然不是万能钥匙，却是 Linux 内核调试的技术之一。

本 hack 还介绍了 Debug memory allocations。使用该功能，即使不是被 LIST_POISON 等污染的内存区域，也可以检测到双重释放、slab cache 的非法使用等。

类似的功能有检查内存页面的 Page alloc debugging（CONFIG_DEBUG_PAGEALLOC）。

（这次链表破坏的例子来自于 2.6.15 版之前在内核的 kernel/posix-timers.c 的 clock_was_set()中实际发生的链表破坏 bug。）

——大岩尚宏

kernel panic

本 hack 根据内核源代码确认操作系统的行为，根据竞态条件编写复现用的测试程序。

从转储中弄清楚原因后编写的补丁是否能改正 bug，是否还有其他应当考虑的地方，以及内核源代码中看上去像是 bug 的地方实际上是否有问题，要确认这些，都需要设法复现。

但是，那些极其罕见的 bug 复现就相当困难了，内核和用户程序都必须执行过 bug 发生的路径，并且时机必须合适，才能复现。

本 hack 在 2.6.9 版内核的发行版中，发现 inode 的代码中可能包含 bug，以此为例讲解一下 bug 的修改方法。使用机器的内存为 2GB，交换空间为 2GB。

199

修改流程

bug 修改的主要流程如图 5-4 所示。

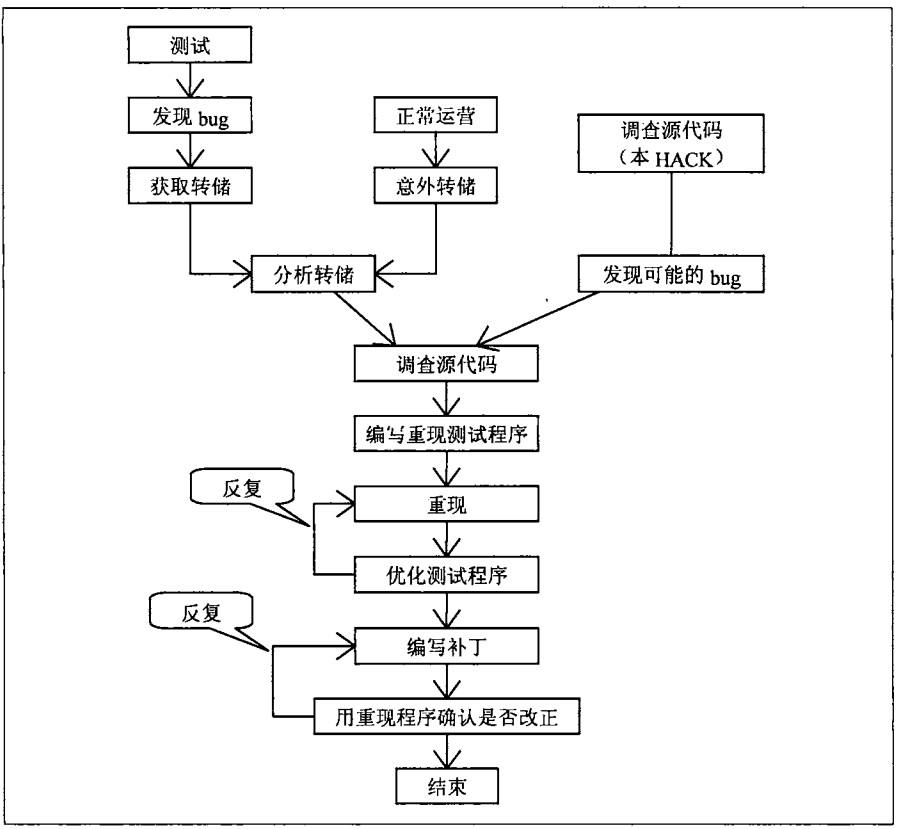


图 5-4 修改 bug 的流程

本 hack 按照以下流程讲述从源代码中发现可能存在的 bug 时的处理方法。

1. 可能存在的 bug。

2. 调查源代码。
3. 确认路径1。
3-1. 用WARN_ON()确认。
4. 确认路径2。
4-1. 调整vfs_cache_pressure参数。
5. 复现bug。
5-1. 用正常内核执行复现测试。
200
5-2. 用mdelay()提高复现率。
6. 确认社区的修改历史。

1.可能存在的bug

Linux内核中有inode这个概念。inode就是管理普通文件、符号链接、目录等信息的缓存。

本次发现的可能的bug位于fs/inode.c的generic_forget_inode()，它临时对inode_lock执行解锁。

```
static void generic_forget_inode(struct inode *inode)
{
   struct super_block *sb = inode->i_sb;

   if (!hlist_unhashed(&inode->i_hash)) {
      if (!(inode->i_state & (I_DIRTY|I_LOCK)))
         list_move(&inode->i_list, &inode_unused);
      inodes_stat.nr_unused++;
      spin_unlock(&inode_lock);
      if (!sb || (sb->s_flags & MS_ACTIVE))
         return;
      write_inode_now(inode, 1);
      spin_lock(&inode_lock);
      inodes_stat.nr_unused--;
      hlist_del_init(&inode->i_hash);
   }
   list_del_init(&inode->i_list);
   inode->i_state|=I_FREEING;
   inodes_stat.nr_inodes--;
```

用iput()的atomic_dec_and_lock()获取锁

在这里解锁

加锁

设置I_FREEING标志

由于nr_inodes不是原子操作，

```
                                                    所以要在锁保护中做减操作
    spin_unlock(&inode_lock);                       解锁
    if (inode->i_data.nrpages)
      truncate_inode_pages(&inode->i_data, 0);
    clear_inode(inode);
    destroy_inode(inode);
}
```

像这样在中途解锁之后，其他 CPU 就有可能获得 inode_lock。如果启用了 CONFIG_PREEMPT，那么等待获取 inode_lock 的进程可能会自行切换，其他 CPU 获得 inode_lock 的可能性就更高了。

201

2. 调查源代码

调查源代码，并将互斥的原理总结如图 5-5。

进程 X 和 Y 互斥后，可能会发生 panic 或其他故障。为证实这一点，下面来编写测试程序。

首先要调查源代码，从某种程度上确认假说是否成立。

3. 确认路径 1

考虑图 5-5 中的进程 X。许多路径都会调用 iput()，而 iput()会调用 generic_drop_inode()。由于数量很多，全部确认相当花时间，因此首先调查一下 iput()是什么函数。iput()给 inode 加上个标志表示已释放，并从管理列表中删除，以便将其释放。然后执行 clear_inode()、destroy_inode()删除该 inode。

inode 是种缓存，可以加速对同一文件的访问，因此反复进行文件创建、删除操作的话，某个 inode 一开始被创建就应该会被 iput()释放。这里用 WARN_ON()确认路径 1 是否会被执行。

202

3-1. 用 WARN_ON()确认

WARN_ON()是内核内部的宏，可以显示栈跟踪以引起注意。这里用它来确认函数调用路径（尽管本来并非用于该目的），输出的信息可以从视觉上确认，这也是个优点。这样就能看到，它是只在特定时机发生，还是频繁发生。

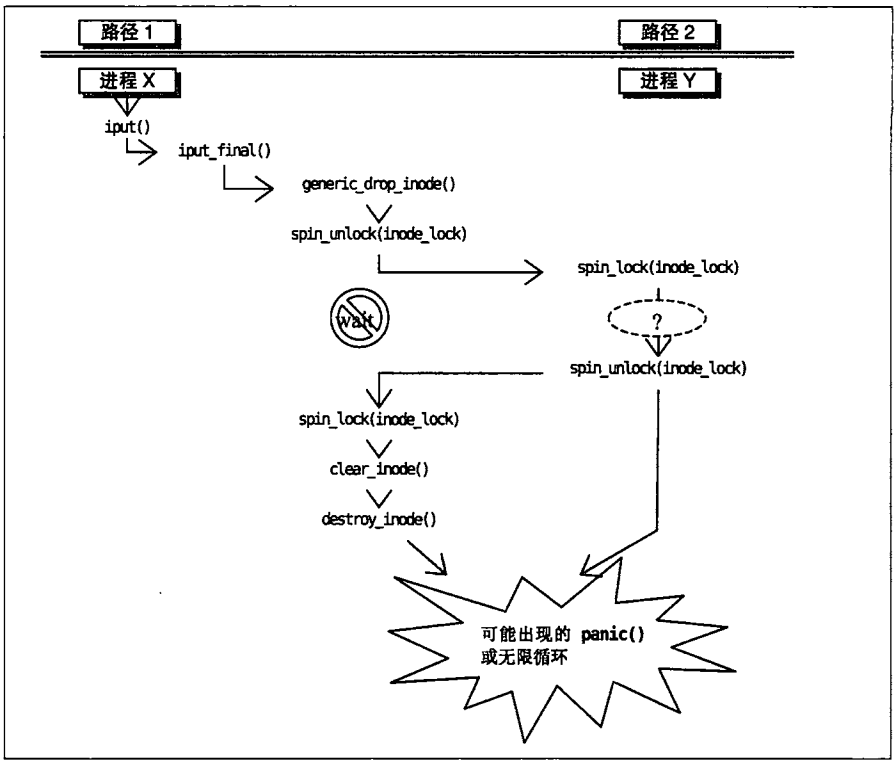


图 5-5 互斥的原理

应用下面的补丁后，重新编译内核。

```
--- fs/inode.c.orig    2005-06-18 04:48:29.000000000 +0900
+++ fs/inode.c 2008-10-13 21:47:40.000000000 +0900
@@ -1035,6 +1036,7 @@ static void generic_forget_inode(struct
                list_move(&inode->i_list, &inode_unused);
        inodes_stat.nr_unused++;
        spin_unlock(&inode_lock);
+       WARN_ON(1);
        if (!sb || (sb->s_flags & MS_ACTIVE))
                return;
        write_inode_now(inode, 1);
```

启动该内核，用 stress 加上 I/O 负载。关于 stress 请参见 **"HACK#56 OOM Killer 的行为和原理"**。inode 依赖于文件数量，因此将文件大小设小一些，设置为 10MB，文件系统为 ext3。

```
# stress --hdd 1 --hdd-bytes 10M
```

控制台上并没有 WARN_ON()的输出，也就是说，路径 1 并没有被执行。内存富裕的情况下，即使创建新文件，也只会增加 ext3 的 inode。ext3 的 inode 数量可以用以下命令确认。

```
# cat /proc/slabinfo | grep ext3_inode_cache
```

在内存不足的状态下，没有充裕的缓存，inode 应该会被释放。因此，用下面的 stress 选项增加内存负载。执行环境为 2GB 内存、2GB 交换区，8 个进程各占用 500MB 内存。

```
# stress --hdd 1 --hdd-byptes 10M --vm 8 --vm-bytes 500M --vm-keep
```

执行后，输出有以下两种形式。这样就能确信，上述命令执行了 generic_forget_inode()。

203

```
Badness in generic_forget_inode at fs/inode.c:1038

Call Trace:<ffffffff801932c8>{generic_drop_inode+152} <ffffffff80190446>{prune_dcache+806}
     <ffffffff80190999>{shrink_dcache_memory+20} <ffffffff8016541f>{shrink_slab+188}
     <ffffffff80166705>{try_to_free_pages+348} <ffffffff8015ed0f>{__alloc_pages+527}
     <ffffffff8015dfb2>{get_zeroed_page+26} <ffffffff80167fdd>{pte_alloc_map+47}
     <ffffffff8016a9bc>{handle_mm_fault+226} <ffffffff80123720>{do_page_fault+520}
     <ffffffff8030dc3e>{thread_return+0} <ffffffff8030dc96>{thread_return+88}
     <ffffffff80110d9d>{error_exit+0}

Badness in generic_forget_inode at fs/inode.c:1038

Call Trace:<ffffffff801932c8>{generic_drop_inode+152} <ffffffff80190446>{prune_dcache+806}
     <ffffffff80190999>{shrink_dcache_memory+20} <ffffffff8016541f>{shrink_slab+188}
     <ffffffff80166705>{try_to_free_pages+348} <ffffffff8015ed0f>{__alloc_pages+527}
     <ffffffff8015c4a2>{generic_file_buffered_write+413}
     <ffffffff8015cd19>{__generic_file_aio_write_nolock+731}
     <ffffffff8015cfb7>{generic_file_aio_write_nolock+32}
     <ffffffff8015d081>{generic_file_aio_write+126}
     <ffffffffa005af01>{:ext3:ext3_file_write+22}
     <ffffffff8017a291>{do_sync_write+173} <ffffffffa0062dfb>{:ext3:ext3_unlink+410}
     <ffffffff80135646>{autoremove_wake_function+0} <ffffffff801898cb>{sys_unlink+313}
     <ffffffff801941ad>{dnotify_parent+34} <ffffffff8017a38c>{vfs_write+207}
```

<ffffffff8017a474>{sys_write+69} <ffffffff80110276>{system_call+126}

调查源代码就会发现，卸载（unmount）处理也会调用 iput()。这是因为卸载文件系统时，文件会执行 sync，相关的缓存就会被全部删除。

因此在挂载的文件系统上写入大量文件，然后删除，再卸载。

```
# mount /dev/sda10 /mnt/10
# cd /mnt/10
# stress --hdd 5 --hdd-bytes 10M --hdd-noclean -t 5
# rm -rfv *
# cd -
# umount /mnt/10
```

执行 umount 后，控制台显示内容如下。

204

```
Badness in generic_forget_inode at fs/inode.c:1038

Call Trace:<ffffffff801932c8>{generic_drop_inode+152} <ffffffffa004d80f>{:jbd:journal_destroy+557}
    <ffffffff80135646>{autoremove_wake_function+0} <ffffffff80135646>{autoremove_wake_function+0}
    <ffffffffa00643dd>{:ext3:ext3_put_super+38} <ffffffffa00643b7>{:ext3:ext3_put_super+0}
    <ffffffff8017f855>{generic_shutdown_super+198} <ffffffff80180677>{kill_block_super+13}
    <ffffffff8017f776>{deactivate_super+95} <ffffffff80195177>{sys_umount+925}
    <ffffffff80182a44>{sys_newstat+17} <ffffffff80110d9d>{error_exit+0}
    <ffffffff80110276>{system_call+126}
```

这样就能确认，刚才的 stress 命令和 umount 命令执行后，路径 1 被执行了。

4. 路径 2 的确认

下面调查一下，执行路径 2 的进程 Y 属于什么情况。这里进程 X 中的 iput()对 inode_lock 加锁之后，对 inode_unused 链表进行了操作。因此，请注意 fs/inode.c 的 prune_icache()，这是因为 prune_icache()也对 inode_lock 和 inode_unused 链表进行了操作。

接下来根据源代码确认一下 prune_icache()路径。调查的结果如图 5-6 所示。

在没有空闲内存的状态下试图分配内存时，__alloc_page()就会将无用的 inode 一起释放掉。此时，似乎会执行 prune_icache()，这是因为，Linux 的结构决定了文件删除或内存释放时，释放处理不会立即被执行，只有在内存不足时才会进行释放处理。

还有一条由 kswapd 开始的路径（图 5-7）。

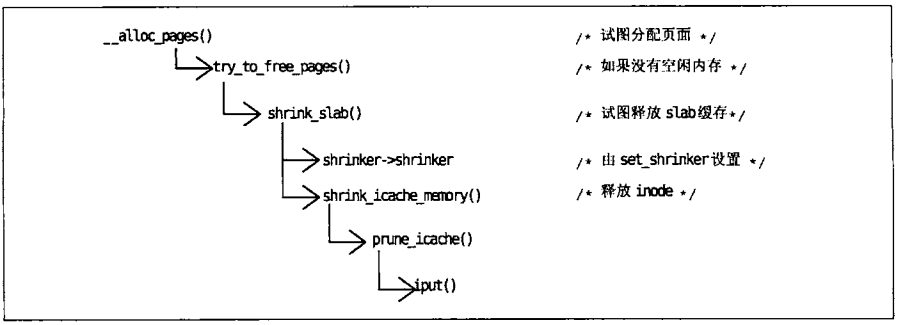
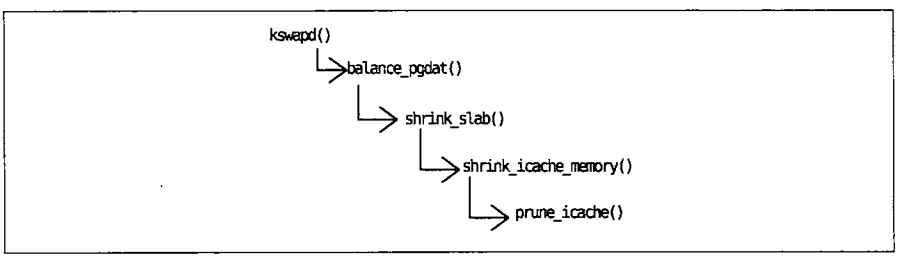


图 5-6 确认路径 2



205

图 5-7 确认路径 2(2)

由于是 kswapd()，所以只需消耗内存，让交换区工作应该就可以。prune_icache()中也会调用 iput()。如果互斥处理不完善，就可能产生双重释放进程 X 已释放的内存的问题。

接下来重新编译内核，同样利用 WARN_ON()确认 prune_icache()是否被执行，同时删除刚才 generic_forget_inode()中的 WARN_ON(1)。补丁如下。

```
--- fs/inode.c  2008-10-14 22:47:19.000000000 +0900
+++ fs/inode.c.next    2008-10-14 22:30:04.000000000 +0900
@@ -443,6 +443,7 @@ static void prune_icache(int nr_to_scan)
           if (inode_has_buffers(inode) || inode->i_data.nrpages) {
                 __iget(inode);
                 spin_unlock(&inode_lock);
+                WARN_ON(1);
                 if (remove_inode_buffers(inode))
                       reap += invalidate_inode_pages(&inode->i_data);
                 iput(inode);
@@ -1035,7 +1036,6 @@ static void generic_forget_inode(struct
```

```
                list_move(&inode->i_list, &inode_unused);
        inodes_stat.nr_unused++;
        spin_unlock(&inode_lock);
-       WARN_ON(1);
        if (!sb || (sb->s_flags & MS_ACTIVE))
                return;
        write_inode_now(inode, 1);
```

接下来要给这个内核增加负载，但执行时间过长途中就会被 OOM Killer 中断。__alloc_pages()会尽可能释放 inode 等未使用的缓存，但如果无法分配足够的内存，__alloc_pages()就会执行 OOM Killer，因此要让内存负载分出强弱来。下面的脚本 vm_stress.sh 在后台执行，20 秒内给内存增加负载，然后休息 2 秒。

206

```
# cat vm_stress.sh
#!/bin/bash

while [ 0 ] ; do
        stress --vm 8 --vm-bytes 500M --vm-keep --timeout 20
        sleep 2
done
# ./vm_stress.sh &
# stress --hdd 1 --hdd-bytes 10M
```

然而并没有出现 WARN_ON()信息，也就是说路径 2 未通过。那调整一下参数试试看。

4-1. 调整参数 vfs_cache_pressure

从 shrink_icache_memory()的代码可见，sysctl_vfs_cache_pressure 的值会导致 return 值变化。sysctl_vfs_cache_pressure 是 sysctl 的参数，可以通过 /proc/sys/vm/vfs_ cache_pressure 改变。增大该值，可以增加 shrink_icache_memory()释放的 inode 数。

通常，shrink_icache_memory()只会处理未使用的 inode 数 nr_unused。nr_unused 可以用 crash 在运行中的系统上确认。

```
crash> struct -o inodes_stat_t inodes_stat
struct inodes_stat_t {
  nr_inodes = 1658,
  nr_unused = 15,
  dummy = {0, 0, 0, 0, 0}
```

```
}
crash>
```

nr_unused为系统此刻的值，在有负载的情况下会不断增加。将vfs_cache_pressure设置到100以上，那么shrink_icache_memory()开始执行时，即使未使用inode数等于nr_unused，prune_icache()执行期间增加的未使用的inode也能被释放。所以我们将vfs_cache_pressure设置为5000（50倍）。

```
# echo 5000 > /proc/sys/vm/vfs_cache_pressure
```

执行vm_stress.sh后，WARN_ON()输出了信息。

207

```
Badness in prune_icache at fs/inode.c:446

Call Trace:<ffffffff80192759>{shrink_icache_memory+309} <ffffffff8016541f>{shrink_slab+188}
       <ffffffff80166705>{try_to_free_pages+348} <ffffffff8015ed0f>{__alloc_pages+527}
       <ffffffff8016a48e>{do_no_page+651} <ffffffff8016aa4f>{handle_mm_fault+373}
       <ffffffff80123720>{do_page_fault+520} <ffffffff8030dc3e>{thread_return+0}
       <ffffffff8030dc96>{thread_return+88} <ffffffff8016da00>{do_mmap_pgoff+1581}
       <ffffffff801eb99d>{__up_write+20} <ffffffff80110d9d>{error_exit+0}
```

但是，kswapd模式的信息并没有输出，那再加一些进程试试。准备4个测试I/O负载的分区，每个分区上用4个stress进程执行I/O。

执行脚本inode.sh后，kswapd模式的信息也出现了（inode.sh脚本请从本书的支持页面（http://www.oreilly.co.jp/books/9784873114040/）下载。

```
# ./vm_stress.sh &
# ./inode.sh

Badness in prune_icache at fs/inode.c:446

Call Trace:<ffffffff80192759>{shrink_icache_memory+309}
       <ffffffff8016541f>{shrink_slab+188}
       <ffffffff80166a39>{balance_pgdat+538}
       <ffffffff80166c63>{kswapd+252}
       <ffffffff80135646>{autoremove_wake_function+0}
       <ffffffff80135646>{autoremove_wake_function+0}
       <ffffffff80110f53>{child_rip+8}
       <ffffffff80166b67>{kswapd+0}
```

这样就可以确认，stress 增加内存负载和 I/O 负载，或者调整 vfs_cache_pressure 之后，就会通过路径 2。

5. 复现 bug

接下来使用刚才建立的脚本，在没有添加 WARN_ON()的正常内核上看看 bug 是否能发生。我们同时执行通过路径 1 的 umount 命令和通过路径 2 的复现测试程序 vm_stress.sh 和 inode.sh。两条路径都是瞬间走完的，所以故障也不可能立即发生。

208

5-1. 在正常内核上进行复现试验

在不包含 WARN_ON()的正常内核上执行脚本几个小时，bug 都没有复现。

为了让复现更容易些，我们添加了 mdelay()。mdelay()的意思是不进行调度，只进行延迟。这样就能延长 spin_unlock()到 spin_lock()的时间，进程 X 和进程 Y 互斥的时机也就容易捕获了。

内核中还有个 msleep()睡眠函数，但它会导致进程调度，所以这里不用它。

用 mdelay()提高复现率

用下面的修改程序试试看。mdelay()延迟了 50 毫秒。

```
--- fs/inode.c.orig    2008-10-16 20:47:38.000000000 +0900
+++ fs/inode.c  2008-10-16 20:39:42.000000000 +0900
@@ -443,6 +443,7 @@ static void prune_icache(int nr_to_scan)
            if (inode_has_buffers(inode) || inode->i_data.nrpages) {
                  __iget(inode);
                  spin_unlock(&inode_lock);
+                 mdelay(50);
                  if (remove_inode_buffers(inode))
                        reap += invalidate_inode_pages(&inode->i_data);
                  iput(inode);
@@ -1035,6 +1036,7 @@ static void generic_forget_inode(struct
                  list_move(&inode->i_list, &inode_unused);
            inodes_stat.nr_unused++;
            spin_unlock(&inode_lock);
+           mdelay(50);
            if (!sb || (sb->s_flags & MS_ACTIVE))
                  return;
```

```
        write_inode_now(inode, 1);
```

我们还进一步改进了复现测试程序，调整了睡眠时间、I/O 大小、内存负载的比例等（最终的复现程序可以在本书的支持页面（http://www.oreilly.co.jp/books/9784873114040/）下载）。

利用加入了 mdelay()的内核和改进后的测试程序，几分钟后 bug 就复现了。下面就是获得转储的 backtrace。umount 命令调用 generic_drop_inode()时发生了 panic。

209

```
crash> bt
PID: 4733  TASK: 100139c27f0     CPU: 2  COMMAND: "umount"
 #0 [10042975b40] start_disk_dump at ffffffffa01a336d
 #1 [10042975b70] try_crashdump at ffffffff8014bd01
 #2 [10042975b80] do_page_fault at ffffffff80123978
 #3 [10042975c00] find_get_pages_tag at ffffffff8015b24c
 #4 [10042975c60] error_exit at ffffffff80110d9d
    [exception RIP: __writeback_single_inode+643]
    RIP: ffffffff80199088 RSP: 0000010042975d18 RFLAGS: 00010246
...
    R13: 000001013ab46800 R14: 000001013a187d78 R15: 0000010042975d58
    ORIG_RAX: ffffffffffffffff CS: 0010 SS: 0018
 #5 [10042975d10] __writeback_single_inode at ffffffff80198f96
 #6 [10042975d50] write_inode_now_err at ffffffff80199204
 #7 [10042975db0] generic_drop_inode at ffffffff80193388
 #8 [10042975dd0] journal_destroy at ffffffffa007a80f
 #9 [10042975e50] ext3_put_super at ffffffffa00913dd
#10 [10042975e80] generic_shutdown_super at ffffffff8017f855
#11 [10042975ea0] kill_block_super at ffffffff80180677
#12 [10042975eb0] deactivate_super at ffffffff8017f776
#13 [10042975ed0] sys_umount at ffffffff80195217
#14 [10042975ef0] sys_newstat at ffffffff80182a44
#15 [10042975f50] error_exit at ffffffff80110d9d
...
```

6. 确认社区的修改历史

应用内核 2.6.12 的补丁（backport）后，这个 bug 就被修改了。

确认即使加入 mdelay()也不会复现，是很重要的。

```
commit 991114c6fa6a21d1fa4d544abe78592352860c82
Author: Alexander Viro <aviro@redhat.com>
Date:  Thu Jun 23 00:09:01 2005 -0700

    [PATCH] fix for prune_icache()/forced final iput() races

commit 4a3b0a490d49ada8bbf3f426be1a0ace4dcd0a55
Author: Jan Blunck <jblunck@suse.de>
Date:  Sat Feb 10 01:44:59 2007 -0800

    [PATCH] igrab() should check for I_CLEAR
```

210

这个补丁在解锁之前设置了 I_WILL_FREE 标志。要释放 inode 时，如果发现 I_WILL_FREE 标志，就不再释放。I_WILL_FREE 标志表示已准备释放，这样两个进程就不会释放同一个 inode 了。

简单地考虑，只要不在途中解锁就没问题，但图 5-8 的函数调用中包含进程调度，所以必须进行解锁，因此只能这样修改。

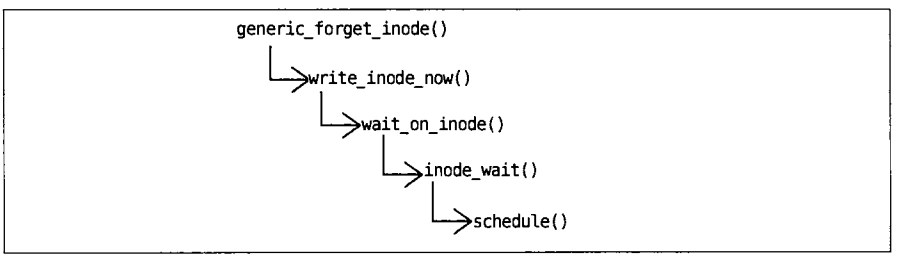


图 5-8 改正后的调用顺序

这次偶然地找到了已改好的补丁，如果要自己修改，理解其中的原理是很重要的。

总结

逐步确认源代码，然后按照 bug 发生的路径在操作系统上执行操作，最终复现了 bug。编写复现测试程序，还能验证补丁（backport）和自行编写的补丁是否正确。此外，这也是个灵活运用 WARN_ON(1)和 sysctl 参数的例子。根据情况和操作系统的行为，适当地应用就好。

内核停止响应（死循环篇）

本 hack 以实际发生的操作系统停止响应的问题为例，介绍内核的调试方法。

某天接到了一个报告，说操作系统停止响应了。报告说，给实时进程用 kill 命令发送 core dump 信号，几十秒后操作系统似乎仍然没有响应。出问题的操作系统是个基于内核版本 2.6.9 的发行版。

211

询问问题发生的详细状况

首先，尽可能收集、整理信息是很重要的。进一步详细地询问后，得知在以下情况下发生了问题。

- 进程调度策略为 SCHED_FIFO。
- 多线程。
- 用 kill 命令发送 SIGSEGV、SIGABRT 等能引发 core dump 的信号。

问题在上述条件全部满足的情况下发生。也许还有其他报告者没注意到的条件，也不能保证不满足这些条件，bug 一定不会发生，但这些的确是有效的信息。

接下来问了问这个实时进程都执行了什么处理，原来它是个提供网络服务的服务器端应用程序，由多个子线程通过 select()系统调用接收客户端的请求。此外，运行 select()时还指定了 timeout 参数。

另外，即使在系统负载较低时，复现频率也很高。

编写复现程序

如果问题能在手头的环境中复现，通向问题的解决之路就非常近了。尽管不是所有问题都能复现，我仍然尽力尝试各种复现步骤，并编写复现程序。这个 bug 用下面的复现程序成功地复现了。出于说明的目的，错误处理等细节部分就省略了。

```
$ cat segfault.c
#include <unistd.h>
#include <pthread.h>
#include <time.h>
```

```
#include <sys/types.h>
#include <signal.h>

void thread (void* buf)
{
   struct timespec tm = { 0, SLEEP_NSEC };
   do {
        nanosleep(&tm, NULL);
   } while (1);
}

int main (void)
{
   pthread_t thr;

   for (i=0; i< NUM_THRAEDS; i++)
        pthread_create( &thr, NULL, thread, NULL );
   /* 稍稍释放CPU，让子线程运行 */
   sleep(2);
   /* 给自己发送core dump信号 */
   kill(0, SIGSEGV);
   return 0;
}
```

212

要改变 NUM_THREADS、SLEEP_NSEC，可以在编译时用 gcc 的-D 选项传递参数。让这些可能成为关键的参数可以被简单地改变，测试时就很方便了。用 chrt 命令执行该程序，就成功地复现了 bug。chrt 命令是 RedHat 系列发行版中的一个命令，可以改变调度策略。将调度策略改成实时，需要 root 权限，因此执行命令时要加上 sudo 命令。我设置的参数 SLEEP_NSEC 为 100 毫秒，NUM_THREADS 为 CPU 个数+1。报告说会停止响应几十秒，而这个程序会导致永久停止响应。

```
$ gcc -DSLEEP_NSEC=10000000 -DNUM_THREADS=3 -lpthread -o segfault segfault.c
$ sudo chrt -f 99 ./segfault
```

用各种条件尝试复现程序

复现成功后，稍稍改变一下条件看看能否复现。我的常用做法是在同一环境中只把内核改成新版本。此外还尝试了一些其他方法，这样就能更详尽地了解发生条件。

- 仅在 SCHED_FIFO 时发生。
- 线程数大于等于 CPU 数+1 时发生。
- 故意访问 NULL 指针时并不会发生。
- 复现频率为 100%，能立即发生。
- 与系统负载没有关系。
- 停止响应后仍然能响应外部的 ping。
  →但是反应速度变慢。
- 无法接收键盘操作。
- 用户空间似乎没有工作
  →不接受任何命令。
- 新版本内核上不发生。
- 旧版本内核上也能发生。

接下来以这些信息为提示，开始分析问题。

分析内核转储

由于这是个停止响应的问题，所以用 watchdog 获取了内核崩溃转储。有关 watchdog 的内容请参见 **“HACK#22 死机时利用 IPMI watchdog timer 获取崩溃转储”**。

首先确认各线程在做什么。

```
crash> ps | grep segfault
  PID   PPID  CPU     TASK        ST   %MEM  VSZ    RSS   COMM
...
  3817  3781  1    1007debf7f0    RU   0.0   24108  528   segfault
> 3818  3781  0    1007cec5030    RU   0.0   24108  528   segfault
> 3819  3781  1    1007dddb030    IN   0.0   24108  528   segfault
```

任务 PID:3817 为父线程，就是执行 kill()系统调用的线程，可见子线程就是各个 CPU 的当前任务。当前任务也有必要调查，不过首先来看看父线程在做什么。

ps命令的运行结果中，最左列标有“>”的就是当前任务。

```
crash> bt 3817
PID: 3817  TASK: 1007debf7f0     CPU: 1  COMMAND: "segfault"
```

214

```
#0 [100780b3c28] schedule at ffffffff8030d7b4
#1 [100780b3d00] sys_sched_yield at ffffffff80134802
#2 [100780b3d20] do_coredump at ffffffff80184e02
#3 [100780b3e10] get_signal_to_deliver at ffffffff801433a1
#4 [100780b3e50] do_signal at ffffffff8010f6fb
#5 [100780b3f50] ptregscall_common at ffffffff801105df
  RIP: 0000003200e2e829 RSP: 0000007fbffffab8 RFLAGS: 00000206
  RAX: 0000000000000000 RBX: 0000000000000000 RCX: ffffffffffffffff
  RDX: 0000000000000000 RSI: 000000000000000b RDI: 0000000000000000
  RBP: 0000007fbffffaf0  R8: 0000007fbffff8e0  R9: 0000002a9557b190
  R10: 0000007fbffffa01 R11: 0000000000000206 R12: 00000000004007d0
  R13: 0000007fbffffbc0 R14: 0000000000000000 R15: 0000000000000000
  ORIG_RAX: 000000000000003e CS: 0033 SS: 002b
```

do_coredump()被调用表明父线程在执行 core dump，其中执行了 sched_yield()，释放了 CPU。查看内核代码后证实其行为的确如此（图 5-9）。

然后再查看 task_struct 及运行队列中残留的时间戳信息，可知一旦释放了 CPU，就无法再获得。进程接收到致命的信号，于是创建 core dump 后就结束了，在这途中释放了 CPU。

接下来调查一下独占 CPU 的当前任务在做什么。下面省略了 watchdog 引发崩溃转储的处理部分。

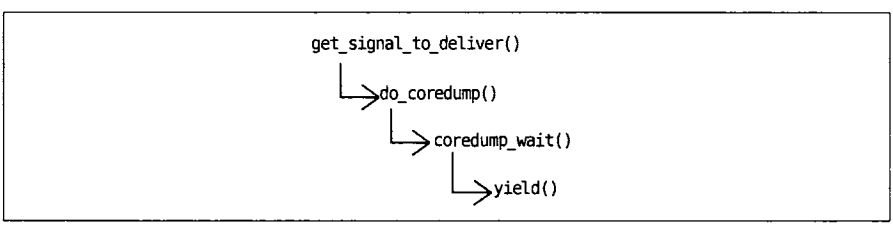


图 5-9 确认线程行为

```
crash> bt -a
PID: 3818  TASK: 1007cec5030     CPU: 0  COMMAND: "segfault"
...
#16 [100778b1e10] get_signal_to_deliver at ffffffff801433c9
#17 [100778b1e50] do_signal at ffffffff8010f6fb
#18 [100778b1f50] ptregscall_common at ffffffff801105df
  RIP: 000000320170bab5 RSP: 0000000040a001a0 RFLAGS: 00000202
  RAX: fffffffffffffdfc RBX: 0000000000000000 RCX: ffffffffffffffff
```

```
    RDX: 0000000000000002 RSI: 0000000000000000 RDI: 00000000004008d0
    RBP: 0000000040a001d0  R8: 0000000040a00960  R9: 0000000040a00960
    R10: 0000000040a00101 R11: 0000000000000202 R12: 0000003201706080
    R13: 000000320170d1c0 R14: 0000000000000000 R15: 000000320170d1c0
    ORIG_RAX: 00000000000000db CS: 0033 SS: 002b

PID: 3819  TASK: 1007dddb030     CPU: 1  COMMAND: "segfault"
...
 #3 [100778b3ef0] schedule_timeout at ffffffff8030e1ad
 #4 [100778b3f50] nanosleep_restart at ffffffff8030e335
 #5 [100778b3f80] system_call at ffffffff8011026a
    RIP: 000000320170bab5 RSP: 00000000414011a0 RFLAGS: 00000202
    RAX: 00000000000000db RBX: ffffffff8011026a RCX: ffffffffffffffff
    RDX: 0000000000000002 RSI: 0000000000000000 RDI: 00000000004008d0
    RBP: 00000000414011d0  R8: 0000000041401960  R9: 0000000041401960
    R10: 00000000414019f0 R11: 0000000000000202 R12: 0000000000000000
    R13: 000000320170d1c0 R14: 0000000000000000 R15: 000000320170d1c0
    ORIG_RAX: 00000000000000db CS: 0033 SS: 002b
```

PID:3818 线程在处理信号。PID:3819 似乎正在执行 nanosleep()，nanosleep()等部分系统调用中一旦发生信号，就会先处理信号，再继续剩下的处理。通过观察内核转储，就可以了解问题发生时各个线程都在运行什么。

加入调试代码进行分析

从 backtrace 来看，当前任务中似乎没有奇怪的地方。只看转储也看不出什么东西，于是在内核源代码中加入 printk()，看看究竟是什么处理花费了较多的时间。我们从分析转储的过程中明白了各个线程正在运行的操作，因此可以高效率地添加调试代码。结果就发现，下面的信号导致了死循环。

nanosleep()被信号中断后，就会停止睡眠，试图处理信号（图 5-10）。信号处理结束后，就执行 nanosleep_restart()，继续残留的睡眠。但是，这次无法处理该信号，导致像图 5-11 那样反复出现睡眠中断和信号处理失败。

信号被发送到进程后，进程内的所有线程都会被设置 TIF_SIGPENDING 标志，并将信号信息放到线程间共享的队列中。nanosleep()中断的原因就是设置了这个 TIF_SIGPENDING。相反，信号处理失败的原因是父线程已开始信号接收处理，从线程间共享队列中已获得了信号信息。

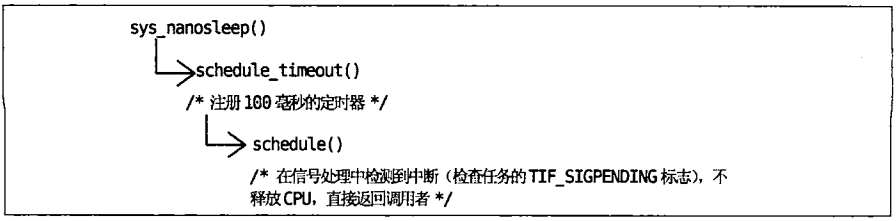


图 5-10 nanosleep()被信号中断

216

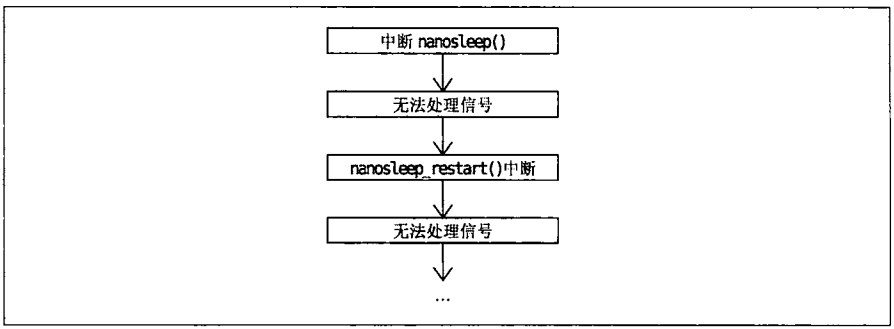


图 5-11 子线程死循环

通常，应当给开始信号接收处理的父线程发送 SIGKILL，以终止所有子线程，但此处在执行该操作之前，子线程就失控了。观察整个流程后发现，问题似乎出在 TIF_SIGPENDING 标志上。尽管应当处理的信号信息已被父线程取走，却仍通知子线程去接收信号。

TIF_SIGPENDING 标志在 recalc_sigpending_tsk()内被清除。recalc_sigpending_tsk()检查当前的信号接收状况，因此几乎所有信号接收处理都会执行该函数。调用流程如图 5-12。

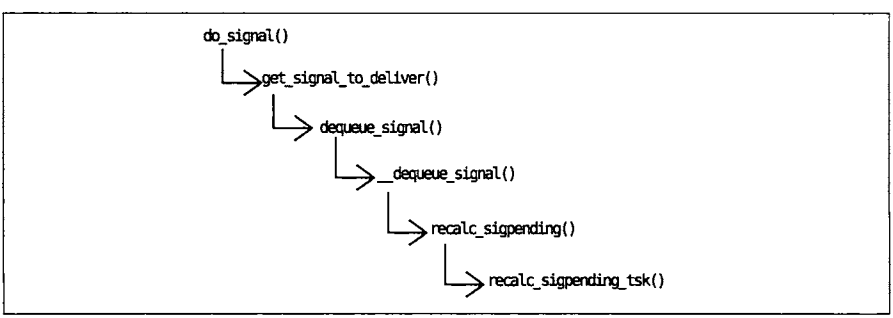


图 5-12 信号接收处理的调用流程

217

recalc_sigpending_tsk()函数如下所示。

[kernel/signal.c]

```
fastcall void recalc_sigpending_tsk(struct task_struct *t)
{
      if (t->signal->group_stop_count > 0 ||
         PENDING(&t->pending, &t->blocked) ||
         PENDING(&t->signal->shared_pending, &t->blocked))
            set_tsk_thread_flag(t, TIF_SIGPENDING);
      else
            /* 清除TIF_SIGPENDING */
            clear_tsk_thread_flag(t, TIF_SIGPENDING);
}
```

这次的复现程序没有改变信号掩码，因此由于 TIF_SIGPENDING 会被清除，所以 group_stop_count 必须在 0 以下。这个 group_stop_count 的含义为信号发送处理中（也就是说 kill()系统调用过程中）从属于该进程的线程数。实际上，这个计数器在 do_coredump()中被清除。

[fs/exec.c]

```
int do_coredump(long signr, int exit_code, struct pt_regs * regs)
{
...
      coredump_wait(mm);

      /*
       * Clear any false indication of pending signals that might
       * be seen by the filesystem code called to write the core file.
       */
      current->signal->group_stop_count = 0;
...
```

而且仔细观察可以发现，这个计数器的清除时机位于父线程释放 CPU 的 coredump_wait()函数之后。调试到这里就能明白，停止响应的原因是 TIF_SIGPENDING 标志，清除该标志所需的 group_stop_count 被设置为 0 的时机有问题。

218

检查社区的修改历史

我们发现，复现程序在新版本内核上不会发生问题。类似于“HACK#33 kernel panic（空指针引用篇）”，搜索 Linus Torvals 的 git 树，发现了下面的补丁。

```
[PATCH] do_coredump() should reset group_stop_count earlier
commit bb6f6dbaa48c53525a7a4f9d4df719c3b0b582af
```

笔者的环境中应用该补丁后，问题就解决了。修改内容为执行 coredump_wait()之前将 group_stop_count 清零。虽然该补丁解决了问题，但同时我也应用了下面这个清理补丁。该清理补丁删除了笔者有疑问的 do_coredump()中调用 yield()的代码。

```
[PATCH] coredump_wait() cleanup
commit 2384f55f8aa520172c995965bd2f8a9740d53095
```

总结

分析故障之前，划分原因是非常重要的。本 hack 讲述了几个重点。

- 认真询问故障发生时的情况。
- 在自己的环境上复现。
- 试验各种各样的条件。

另外，在实际的分析中，通过分析 kernel dump，调查问题发生时内核在做什么，可以高效率地添加调试代码。之后要确认 git 的修改历史，只有弄清楚原因后，才能知道应该移植哪个补丁。

参考文献

- Linus Torvals 的 git 树
  http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

——安部东洋

219

内核停止响应（自旋锁篇之一）

本 hack 介绍单纯的自旋锁死锁时形成的转储的阅读方法。

本 hack 中，我们来看看内核中的自旋锁死锁时的转储。所用的 Linux 内核为 2.6.9 版本。

复现

下面的 spinlock_stall.c 是个自旋锁死锁的模块。

```
# cat spinlock_stall.c
#include <linux/module.h>
#include <linux/kthread.h> // for kthread_run()
#include <linux/delay.h> // for msleep()

DEFINE_SPINLOCK(lock1);
DEFINE_SPINLOCK(lock2);
int thread2_flag;

static int spinlock_stall_thread1(void *data)
{
   spin_lock(&lock1);
   while(1){
      if (thread2_flag == 1) break;
      msleep(200);
   }
   spin_lock(&lock2);
   return 0;
}
static int spinlock_stall_thread2(void *data)
{

   spin_lock(&lock2);
   thread2_flag = 1;
   spin_lock(&lock1);
   return 0;
}
static int __init spinlock_init(void)
{
   struct task_struct *kthread1;
   struct task_struct *kthread2;

   spin_lock_init(&lock1);
   spin_lock_init(&lock2);

   kthread1 = kthread_run(spinlock_stall_thread1, NULL, "spinlock1");
```

220

```
    kthread2 = kthread_run(spinlock_stall_thread2, NULL, "spinlock2");

    return 0;
}
static void __exit spinlock_exit(void) { return; }

module_init(spinlock_init);
module_exit(spinlock_exit);
# cat Makefile
obj-m := spinlock_stall.o
```

将该模块加载到内核中后，thread1 和 thread2 就会产生死锁。make 和 insmod 的执行方法如下所示。make 的-C 选项可以指定现在运行中的内核的源代码目录。

```
# ls
Makefile spinlock_stall.c
# make -C /usr/srclinux M='pwd' modules
...
# insmod spinlock_stall.ko
```

insmod 之后就会停止响应，然后用 IPMI watchdog 获取转储。

分析转储

来看看转储的 backtrace。

```
crash> bt -ta
PID: 13417  TASK: 1007de867f0     CPU: 0   COMMAND: "spinlock1"
...
--- <IRQ stack> ---
 [    1006ddb9e58] apic_timer_interrupt at ffffffff80110bf5
  [exception RIP: .text.lock.spinlock+2]
...
 [    1006ddb9f00] msleep at ffffffff801407f9
 [    1006ddb9f10] spinlock_stall_thread1 at ffffffffa01cf02e
 [    1006ddb9f20] kthread at ffffffff8014b6c3
 [    1006ddb9f50] child_rip at ffffffff80110f47
 [    1006ddb9f58] keventd_create_kthread at ffffffff8014b6ec
 [    1006ddb9fc8] kthread at ffffffff8014b5fb
...
```

221

```
PID: 13418  TASK: 1007dd88030     CPU: 1  COMMAND: "spinlock2"
          START: smp_call_function_interrupt at ffffffff8011c5f0
...
--- <IRQ stack> ---
 [    1006e613e58] call_function_interrupt at ffffffff80110b69
  [exception RIP: .text.lock.spinlock+5]
...
 [    1006e613f10] spinlock_stall_thread2 at ffffffffa01cf055
 [    1006e613f20] kthread at ffffffff8014b6c3
 [    1006e613f50] child_rip at ffffffff80110f47
 [    1006e613f58] keventd_create_kthread at ffffffff8014b6ec
 [    1006e613fc8] kthread at ffffffff8014b5fb
...
```

自旋锁处于 busy wait 状态（不进行任务调度，陷入死循环），这点从 backtrace 上可以直观地看出来。存在“.text.lock.spinlock”的话，大多数情况可以判断为在自旋锁处停止响应（“.text.lock.spinlock”在不同内核版本下可能不一样，2.6.28 内核中显示为“_spin_lock”）。

总结

自旋锁引起的死锁可以从转储中立即被发现，但是现在的 Linux 内核中几乎没有这种 bug。大多数情况为对内核做出修改，或是在创建驱动程序、模块等时犯下的简单错误。

内核停止响应（自旋锁篇之二）

本 hack 以 NMI watchdog timeout 发生时的内核调试方法为例进行说明。

我们在拿到新硬件之后，进行了连续的 kdump 测试。结果，控制台上偶然会显示以下的信息，表明 kdump 失败了。使用的操作系统为基于 2.6.18 版内核的发行版。

222

```
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14NMI Watchdog detected LOCKUP on CPU 0
CPU 0
Modules linked in:
Pid: 1, comm: swapper Not tainted 2.6.18-prep #6
RIP: 0010:[<ffffffff80063b7c>]  [<ffffffff80063b7c>] .text.lock.spinlock+0x2/0x30
```

①

```
RSP: 0000:ffffffff8040fd00  EFLAGS: 00000086 ──────────────────────────── ②
RAX: ffff8100014cdf00 RBX: ffffffff803af380 RCX: 0000000000000000
RDX: ffffffff80407f00 RSI: ffffffff8040fd48 RDI: ffffffff803af3bc
RBP: 0000000000000000 R08: 0000000000000003 R09: 000000000079e321
R10: 0000000000000096 R11: 0000000000000086 R12: 0000000000000000
R13: 000000000000000e R14: ffffffff803af3bc R15: ffffffff8040fd48
FS:  0000000000000000(0000) GS:ffffffff80397000(0000) knlGS:0000000000000000
CS:  0010 DS: 0018 ES: 0018 CR0: 000000008005003b
CR2: 00002aaaae176000 CR3: 0000000001001000 CR4: 0000000000006e0
Process swapper (pid: 1, threadinfo ffff810008d00000, task ffff8100019fd7a0)
Stack:  ffffffff800b5efb 0000000000000086 0000000000000000 ffffffff8040fd48
 0000000000000000 000000000000000e ffffffff8040feb8 0000000000000001
 ffffffff8006b3bf 0000000300000000 ffff8100084734c0 ffffffff8040fd70
Call Trace:
 <IRQ>  [<ffffffff800b5efb>] __do_IRQ+0x47/0x105
 [<ffffffff8006b3bf>] do_IRQ+0xe7/0xf5 ──────────────────────────── ③
 [<ffffffff8005c615>] ret_from_intr+0x0/0xa
 [<ffffffff8002e007>] __wake_up+0x38/0x4f
 [<ffffffff800107be>] handle_IRQ_event+0x1b/0x58 ──────────────────── ④
 [<ffffffff8000d276>] ide_intr+0x11f/0x1df
 [<ffffffff800b69bd>] note_interrupt+0x13a/0x227 ──────────────────── ⑤
 [<ffffffff800b5f7b>] __do_IRQ+0xc7/0x105
 [<ffffffff8006b3bf>] do_IRQ+0xe7/0xf5 ──────────────────────────── ⑥
 [<ffffffff8005c615>] ret_from_intr+0x0/0xa
 [<ffffffff80159597>] vgacon_cursor+0x0/0x1a5
 [<ffffffff80011cd5>] __do_softirq+0x53/0xd5
 [<ffffffff8005d2fc>] call_softirq+0x1c/0x28
 [<ffffffff8006b53c>] do_softirq+0x2c/0x85
 [<ffffffff8005cc8e>] apic_timer_interrupt+0x66/0x6c
 <EOI>  [<ffffffff80159597>] vgacon_cursor+0x0/0x1a5
 [<ffffffff8008e71e>] vprintk+0x29e/0x2ea
 [<ffffffff8005a5bb>] cache_alloc_refill+0x106/0x186
 [<ffffffff8008e7bc>] printk+0x52/0xbd
 ...
```

223

从第1行可知NMI watchdog检测出了死锁。NMI watchdog显示出死锁情况所在的CPU的信息。关于NMI watchdog请参见**"HACK#23 用NMI watchdog在死机时获取崩溃转储"**。该现象是在/proc/sysrq-trigger引发panic之后，启动dump kernel的过程中发生的。NMI watchdog虽然能在检测出死锁之后获取转储，但这次是kdump的问题，因此无法转储。那么首先从这个信息进行调试吧。NMI watchdog

显示的信息的阅读方法基本上与 Oops 信息的相同。关于 Oops 信息请参见“HACK#15 Oops 信息的解读方法”。

分析信息

首先简单地确认一下问题发生时的情况。信息中的①是正在运行的代码，看见“.text.lock.spinlock”就知道正在获取自旋锁。接下来看看②的 EFLAGS，可知 IF 标志（中断标志）已被清除，也就是在禁止中断的状态下获取自旋锁。

在 x86 和 x86_64 架构中，IF 标志是 EFLAGS 寄存器的第 9 个比特。00000086 中清除了 IF 标志，因此是禁止中断的状态。如果是 00000286，即设置了 IF 标志，就是允许中断的状态。

说点题外话，大部分发行版的 kdump 的 dump kernel 的启动参数中都加入了以下选项。

```
maxcpus=1 irqpoll
```

第 1 个表示 dump kernel 在单处理器（Uniprocessor）上运行，第 2 个表示启用 IRQ 轮询。详细的内核启动参数请参见内核源代码树中的 Documentation/kernel-parameters.txt。这次第 1 个参数特别重要，因为死锁并不是多个 CPU 的竞态条件，而是同一个 CPU 试图多次获取同一个锁而发生的。也就是说，该信息中显示的栈跟踪中包含了全部必要的信息。

我们来看看栈跟踪。③是个 do_IRQ()调用，可知是在 IRQ 中断处理程序运行中发生了死锁。do_IRQ()这个子程序会根据接收的 IRQ 中断启动相应的中断处理程序。do_IRQ()函数只会获取一个锁，就是每个 IRQ 号都有的中断描述符（struct irq_desc *desc）的锁 desc->lock。估计是这个 desc->lock 引起了死锁。

224

根据⑥，以前也调用过这个 do_IRQ()，可能是在 IRQ 中断中又发生了 IRQ 中断。⑥取得 desc->lock，但③并不知道这一点，试图获取同一个 desc->lock 时发生了死锁。这不禁让人怀疑，会是如此简单的问题吗？平时不用 kdump 时可不会发生这种死锁。如果只是因为连续接收 IRQ 中断就导致了死锁，那么操作系统就完全不能用了。跟平时有什么区别吗？于是注意到另一个启动选项：irqpoll，轮询 IRQ……似乎有点问题。

关于 irqpoll 选项

有时会发生在 IRQ 中断时找不到相应的中断处理程序的情况。有时是设备的固件 bug 引起的，但是在使用 kdump 的环境中，这种情况经常发生。可能导致 panic 的是在设备尚未关闭、dump kernel 加载相应的驱动程序之前，设备上发生中断。我们经常会看到下面这种信息。

```
irq X: nobody cared (try booting with the "irqpoll" option)
```

这就是找不到 IRQ 中断对应的处理程序时显示的信息。这种现象多次发生，就会导致该 IRQ 被禁用。为了避免这个问题，许多发行版都给 dump kernel 的启动参数中加上了 irqpoll 选项。

以 irqpoll 为重点跟踪源代码

请注意⑤的 note_interrupt()函数。这个函数用于检查接收的 IRQ 中断是否被正确处理了。如果找不到相应的中断处理程序，通常会像刚才那样只显示一条信息，但设置了 irqpoll 选项时，行为会稍有变化。

225

[kernel/irq/handle.c]

```
fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
{
   struct irq_desc *desc = irq_desc + irq;
...
      /*
       * 在desc->lock 已加锁的情况下，do_IRQ()调用了 note_interrupt()
       */
      spin_lock(&desc->lock);————————————————————————⑦
      if (!noirqdebug)
         note_interrupt(irq, desc, action_ret, regs);
...
```

[kernel/irq/spurious.c]

```
void note_interrupt(unsigned int irq, struct irq_desc *desc,
                irqreturn_t action_ret, struct pt_regs *regs)
{
...
   if (unlikely(irqfixup)) {
      /* Don't punish working computers */
      if ((irqfixup == 2 && irq == 0) || action_ret == IRQ_NONE) {
```

```
            int ok = misrouted_irq(irq, regs);
            if (action_ret == IRQ_NONE)
               desc->irqs_unhandled -= ok;
         }
      }
   ...
```

指定 irqpoll 选项后，irqfixup 的值为 2，misrouted_irq()函数就会被调用。misrouted_irq()函数为了检查相应的中断处理程序是否注册了错误的 IRQ 号，会搜索其他 IRQ 号上注册的处理程序并调用之。如果找到正确的处理程序，该函数就返回 1。来看看 misrouted_irq()的内容。

[kernel/irq/spurious.c]

```
static int misrouted_irq(int irq, struct pt_regs *regs)
{
   for (i = 1; i < NR_IRQS; i++) {
      struct irq_desc *desc = irq_desc + i;
      struct irqaction *action;

      if (i == irq) /* 只检查其他IRQ号 */
         continue;
      ...
      /* 调用其他IRQ号上注册的中断处理程序 */
      ...
      while ((desc->status & IRQ_PENDING) && action) {
         /*
          * 对上述处理时发生的IRQ中断中未处理的部分
          * (pending的中断)进行处理
          */
         working = 1;
         spin_unlock(&desc->lock); ——————————————————⑧
         handle_IRQ_event(i, regs, action);
         spin_lock(&desc->lock);
         desc->status &= ~IRQ_PENDING;
      }
   ...
```

226

请看注释部分，在处理处于 pending 状态的中断时会调用 handle_IRQ_event()函数。请看开头的死锁信息④，该函数的名字显示在栈跟踪中，这是不正常的。⑧明明已经为 desc->lock 解锁，但这跟⑦的 note_interrupt()调用时加锁的那个 desc->lock 并不是一个锁。下面来看看 handle_IRQ_event()。

```
[kernel/irq/handle.c]
irqreturn_t handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                             struct irqaction *action)
{
...
   if (!(action->flags & IRQF_DISABLED))
      local_irq_enable_in_hardirq();
...
```

227

似乎 action->flags 中如果没有 IRQF_DISABLED 标志，就允许中断。如果是与其他设备共享 IRQ 的设备，这里就会出问题，敏锐的人应该已经注意到问题了。流程如图 5-13 所示。

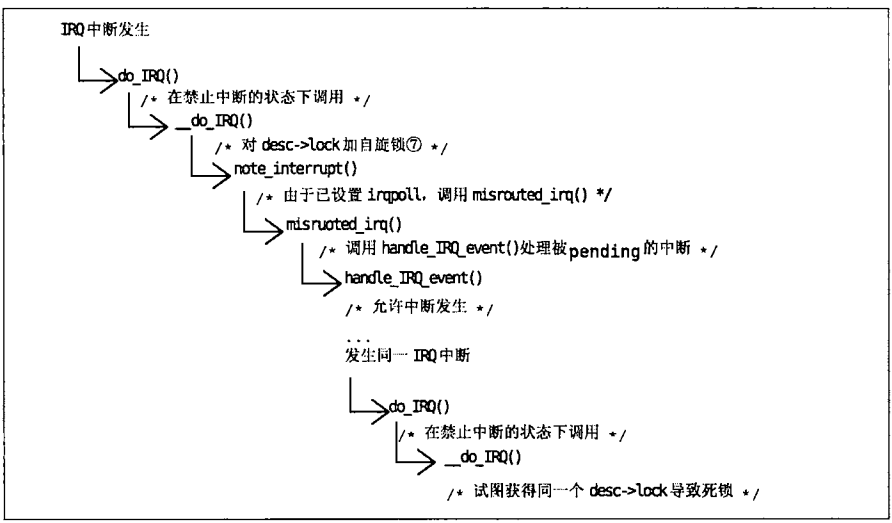


图 5-13 引发死锁的流程

检查社区的修改历史

分析到这里，问题的原因就很清楚了。在已获得 desc->lock 的情况下调用 note_interrupt()有问题。寻找其他 IRQ 处理程序时其实并不需要获取当前中断的描述符的锁。

社区是如何看待这个问题的？检查修改历史后发现了两个补丁，其修改内容和我想的一样。

```
commit f72fa707604c015a6625e80f269506032d5430dc
```

```
Author: Pavel Emelianov <xemul@openvz.org>
Date:   Fri Nov 10 12:27:56 2006 -0800

    [PATCH] Fix misrouted interrupts deadlocks

commit b42172fc7b569a0ef2b0fa38d71382969074c0e2
Author: Linus Torvalds <torvalds@woody.osdl.org>
Date:   Wed Nov 22 09:32:06 2006 -0800

  Don't call "note_interrupt()" with irq descriptor lock held
```

228

总结

本 hack 以单 CPU 下的死锁为例，介绍了利用 NMI watchdog 给出的内核故障信息进行调试的方法。此外，有时可以像这次的问题一样，从内核启动参数寻找原因。

参考文献

- Miracle Linux “Linux 的 110” [注2]
  http://www.miraclelinux.com/support/?q=node/246

——安部东洋

内核停止响应（信号量篇）

本 hack 介绍实际发生的信号量导致死锁的问题的解析方法。

问题内容

在某个以 Linux 2.6.9 内核为基础的发行版上测试内核时，出现了用户进程无响应的问题。为了确认情况执行了 ps 命令，结果 ps 命令也没有响应。

此外，其他进程的运行似乎没有问题。

- 基于 Linux 内核 2.6.9 的发行版。

注 2：“Linux 的 110” 是 Miracle Linux 的支持页面，提供面向开发者的技术支持资料。 ——译者注

- CPU x86_64。
- 内存 8GB。

收集崩溃转储

为了确认、分析出问题的进程的详细情况，我们先收集了崩溃转储。

下面介绍由崩溃转储分析问题的详细方法。

检查进程情况

首先看看无响应的 ps 命令的情况。

利用 crash 的 ps 命令看看无响应的进程（ps 命令）的情况。

```
crash> ps | grep ps
   2943  2596  2    1022f85d170   UN  0.0   5408  1036  ps
```

229

pid 为 2943，状态为 UN（UNINTERRUPTABLE）。考虑一下该进程无响应的可能原因，有以下几种情况。

- 等待磁盘 I/O。
- 等待某个事件，但并不是普通的睡眠。
- 也不是 busy loop。

检查进程的 backtrace

接下来检查一下 backtrace 信息，看看是如何到达该状态（UNINTERRUPTABLE）的。确定现象是执行了哪些处理而发生的，对于分析问题十分重要。

利用 crash 的 bt 命令输出 backtrace 信息。

```
crash> bt 2943
PID: 2943  TASK: 1022f85d170     CPU: 2  COMMAND: "ps"
 #0 [10226357c88] schedule at ffffffff805537d7 ———————————— ②
 #1 [10226357db0] __down_read at ffffffff80554bbf ———————————— ①
 #2 [10226357df0] access_process_vm at ffffffff801413ca ———————————— ③
 #3 [10226357e70] proc_pid_cmdline at ffffffff801bb0e5
 #4 [10226357eb0] proc_info_read at ffffffff801bb5e0
```

```
#5 [10226357ef0] vfs_read at ffffffff8018613e
#6 [10226357f20] sys_read at ffffffff801863dc
#7 [10226357f80] no_syscall_entry_trace at ffffffff8010e539
  RIP: 0000002a95827232 RSP: 0000007fbfffe640 RFLAGS: 00010202
  RAX: 0000000000000000 RBX: ffffffff8010e539 RCX: 0000000000000000
  RDX: 00000000000007ff RSI: 0000007fbfffddf0 RDI: 0000000000000006
  RBP: 0000000000000000  R8: 0000000000000000  R9: 0000000000000000
  R10: 00000000746f6f72 R11: 0000000000000246 R12: 0000000000000000
  R13: 0000000000000006 R14: 00000000005464a0 R15: 0000000000000000
  ORIG_RAX: 0000000000000000 CS: 0033 SS: 002b
```

从这段 backtrace 信息可以看出，ps 命令的进程调用了__down_read()（①），然后进一步调用 schedule()（②）。可以认为，它在等待只读信号量。

死锁

从信号量操作等互斥处理的结构上看，必然存在发生死锁的可能性。这里等待的信号量一直不返回，可以认为发生了死锁。

230

虽然知道是在等待信号量，但并不知道是在等待什么信号量，因此也弄不清是哪个处理上发生的问题，因此应该调查一下它在试图操作什么信号量。下面我们来看看调用了 down_read()的函数 access_process_vm()（③）。

看看源代码。

```
kernel/ptrace.c:
int access_process_vm(struct task_struct *tsk, unsigned long addr, void *buf, int len,
int write)
{
      struct mm_struct *mm;
      struct vm_area_struct *vma;
      struct page *page;
      void *old_buf = buf;

      mm = get_task_mm(tsk);
      if (!mm)
            return 0;

      down_read(&mm->mmap_sem); ————————————————————————④
```

```
        /* ignore errors, just check how much was sucessfully transfered */
        while (len) {
        ...
        }

        up_read(&mm->mmap_sem); ——————————————————————————— ⑤
        mmput(mm);

        return buf - old_buf;
}
```

从这个函数的处理来看，主循环 while()前后通过信号量对 mm->mmap_ sem 进行了互斥控制（④和⑤）。

这是对进程的内存结构（struct）的处理的保护，是个读取类信号量的请求，可能发生死锁的情况就是其他进程已获取了写入类的信号量，但是系统处于空闲状态，似乎没有持有信号量并处于繁忙状态的进程。

231 不过，无法分配读取信号量，应该跟信号量的写入有关系。那么来看看其他进程是否持有写入信号量。

通常，访问某个内存结构的只有拥有该内存结构的进程本身，几乎不会有访问其他进程的内存结构的情况。

从 backtrace 信息可以看出，ps 命令通过 proc 文件系统进行访问。

那么，试图访问 proc 文件系统的进程有哪些？

看上去像在读取/proc/pid/cmdline。

检查一下系统调用的参数。详细情况请参见 AMD64 的 ABI（Application Binary Interface）。

```
RAX=0 => read 系统调用
RDI=6 => fd=6
RSI   => ptr
RDX   => size
```

因此，只要弄清 fd=6 的文件是什么就行了。

利用 crash 的 files 命令查看 ps 命令的进程打开的文件，看看 fd=6 是什么文件。

```
crash> files 2943
PID: 2943  TASK: 1022f85d170     CPU: 2  COMMAND: "ps"
ROOT: /   CWD: /root
 FD     FILE         DENTRY         INODE        TYPE PATH
  0   1022e3e5e40    102244026a8    1022e4813a0  CHR /dev/pts/1
  1   1022e3e5e40    102244026a8    1022e4813a0  CHR /dev/pts/1
  2   1022e3e5e40    102244026a8    1022e4813a0  CHR /dev/pts/1
  3   1022ea1a2c0    1022dcaca48    1022957d320  REG /proc/uptime
  4   1022ea1ab80    10224edfde8    1022957d098  REG /proc/meminfo
  5   1022e621e80    100cff563f0    100cff51d00  DIR /proc/
  6   1022e621200    10224402b30    1022dad0a78  REG /proc/2639/cmdline
```

该内存结构属于 pid=2639 这个进程，那么来看看 pid=2639 进程的信息。这里要用到 crash 的 ps 和 bt 命令。

```
crash> ps | grep 2639
  2639  2554  3   1022e9920f0   UN  0.0  31248   696 MYAPL
```

232

```
crash> bt 2639
PID: 2639  TASK: 1022e9920f0     CPU: 3  COMMAND: "MYAPL"
 #0 [10229a65b88] schedule at ffffffff805537d7
 #1 [10229a65cb0] __down_read at ffffffff80554bbf
 #2 [10229a65cf0] do_page_fault at ffffffff80121fcc
 #3 [10229a65e20] error_exit at ffffffff8010f0dd
    [exception RIP: copy_user_generic+178]
    RIP: ffffffff802d1632  RSP: 0000010229a65ed8  RFLAGS: 00010202
    RAX: 0000000000000000  RBX: 0000000000000001  RCX: 0000000000000001
    RDX: 0000000000000001  RSI: 0000010225f44000  RDI: 0000002a95c018f0
    RBP: 0000000000000001   R8: 00000000fffffffa   R9: ffffffff806bcd10
    R10: 0000000000000001  R11: 0000000000000001  R12: 0000000000000001
    R13: 0000000008000201  R14: 0000000000000001  R15: 0000010225f44000
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0000
 #4 [10229a65ee0] mincore_vma at ffffffff80177a9a
 #5 [10229a65f40] sys_mincore at ffffffff80177bf3
 #6 [10229a65f80] no_syscall_entry_trace at ffffffff8010e539
    RIP: 0000002a958323f9  RSP: 0000007fbffff3c0  RFLAGS: 00010206
```

```
RAX: 000000000000001b RBX: ffffffff8010e539 RCX: 000000000000000c
RDX: 0000002a95c018f0 RSI: 0000000000001000 RDI: 0000000000400000
RBP: 0000000000000000  R8: 0000000000020711  R9: 0000002a95c008e0
R10: 0000000000000003 R11: 0000000000000206 R12: 0000000000000000
R13: 0000000000000000 R14: 0000000000400000 R15: 0000000000000002
ORIG_RAX: 000000000000001b CS: 0033 SS: 002b
```

这里找到的进程 MYAPL 就是最初失去响应的用户进程。根据 backtrace 信息，无响应的 ps 命令的进程也在等待读取类的信号量。另外，这个信号量操作似乎是页面错误（page fault）后的操作。

异常处理中发生死锁的一种情况就是，在获取了某个互斥对象的情况下再次尝试获取该互斥对象。

检查一下页面错误引发的异常处理的流程。

```
mm/mincore.c:
asmlinkage long sys_mincore(unsigned long start, size_t len,
        unsigned char __user * vec)
{
    int index = 0;
    unsigned long end;
    struct vm_area_struct * vma;
    int unmapped_error = 0;
    long error = -EINVAL;

    down_read(&current->mm->mmap_sem);
...
        /* Here vma->vm_start <= start < vma->vm_end. */
        if (end <= vma->vm_end) {
            if (start < end) {
                error = mincore_vma(vma, start, end, &vec[index]);
                if (error) goto out;
            }
            error = unmapped_error;
            goto out;
        }
...
out:
    up_read(&current->mm->mmap_sem);
```

233

```
        return error;
}
```

查看代码后发现，sys_mincore()中获得内存结构的信号量之后，mincore_vma()被调用了。从 backtrace 中可以看到，这个 mincore_vma()中发生了页面错误异常。

似乎这就是问题的原因。也就是说，在已获取信号量的情况下，在异常处理中再次试图获取同一信号量，导致了死锁的发生。

但是，只有读取类的信号量操作是不会发生死锁的，所以必然在某个地方存在写入类的信号量。

来看看进程的详细情况。再次运行 crash 的 ps 命令。这里关注一下用户进程 MYAPL，会发现 MYAPL 是个线程。

```
crash> ps
...
  2639  2554  3   1022e9920f0   UN  0.0  31248   696 MYAPL
  2640  2554  3   1022e045810   UN  0.0  31248   696 MYAPL
  2641  2554  1   1022e9b4170   UN  0.0      0     0  MYAPL
  2642  2554  0   1022e959850   UN  0.0  31248   696 MYAPL
...
```

234

显示一下各个线程的 backtrace。

```
crash> bt 2640
PID: 2640  TASK: 1022e045810     CPU: 3  COMMAND: "MYAPL"
 #0 [10228c13cb8] schedule at ffffffff805537d7
 #1 [10228c13de0] __down_read at ffffffff80554bbf
 #2 [10228c13e20] do_page_fault at ffffffff80121fcc
 #3 [10228c13f50] error_exit at ffffffff8010f0dd
    RIP: 0000002a957e8793  RSP: 00000000407ff780    RFLAGS:   00010206
    RAX: 0000002a9599b540  RBX: 0000002a95a00020    RCX:      000000000000000c
    RDX: 0000000000002001  RSI: 0000002a95a00768    RDI:      0000000000001000
    RBP: 0000002a95a000b8   R8: 000000000001e711    R9:       0000002a95a008e0
    R10: 0000000000000003  R11: 0000000000000100    R12:      0000002a95a028f0
    R13: 0000000000002015  R14: 0000000000002011    R15:      0000000000000000
    ORIG_RAX: ffffffffffffffff  CS: 0033  SS: 002b
crash> bt 2641
PID: 2641  TASK: 1022e9b4170     CPU: 1  COMMAND: "MYAPL"
```

```
 #0 [1022924bab8] schedule at ffffffff805537d7
 #1 [1022924bbe0] __down_read at ffffffff80554bbf
 #2 [1022924bc20] do_futex at ffffffff8014fd91
 #3 [1022924bd30] sys_futex at ffffffff8015010f
 #4 [1022924bd90] do_exit at ffffffff8013b879
 #5 [1022924be00] get_signal_to_deliver at ffffffff80145512
 #6 [1022924be50] do_signal at ffffffff8010d958
 #7 [1022924bf50] retint_signal at ffffffff8010eb76
    RIP: 0000002a957ea2a4 RSP: 0000000040fff810 RFLAGS: 00000246
    RAX: 0000000000004011 RBX: 0000002a9599b540 RCX: 0000000000004010
    RDX: 0000002a9599b5d8 RSI: 00000000005031d0 RDI: 00000000005031c0
    RBP: 0000000000000000  R8: 00000000005091e0  R9: 00000000005061d0
    R10: 0000000000001001 R11: 0000000000001000 R12: 0000002a9567300b
    R13: 0000002a956799c0 R14: 0000000000001000 R15: 0000000000000003
    ORIG_RAX: ffffffffffffffff CS: 0033 SS: 002b
crash> bt 2642
PID: 2642  TASK: 1022e959850     CPU: 0  COMMAND: "MYAPL"
 #0 [10228c15d78] schedule at ffffffff805537d7
 #1 [10228c15ea0] __down_write at ffffffff80554b1b
 #2 [10228c15ee0] sys_mprotect at ffffffff8017aa8b
 #3 [10228c15f80] no_syscall_entry_trace at ffffffff8010e539
    RIP: 0000002a95832309 RSP: 00000000417ff780 RFLAGS: 00000297
    RAX: 000000000000000a RBX: ffffffff8010e539 RCX: 0000000000000004
    RDX: 0000000000000003 RSI: 0000000000024000 RDI: 0000002a95d00000
    RBP: 0000000000000003  R8: 00000000ffffffff  R9: 0000000000000000
    R10: 0000000000004022 R11: 0000000000000217 R12: 0000000000001000
    R13: 0000002a956799c0 R14: 0000000000024000 R15: 0000002a95d00000
    ORIG_RAX: 000000000000000a CS: 0033 SS: 002b
```

235

pid=2642的线程调用down_write()，请求写入类的信号量。可以认为，与这个down_write()结合操作导致了死锁。

根据以上内容，可以推测问题发生流程如图5-14所示。

陷入死锁状态后，访问出问题的进程的内存结构的所有进程都会陷入等待信号量的睡眠状态。但是，对其他进程并没有影响。

死锁的原因在于mincore()系统调用中，sys_mincore()在持有读取信号量的状态下，执行了可能会引发页面错误异常的处理（即copy_to_user()）。

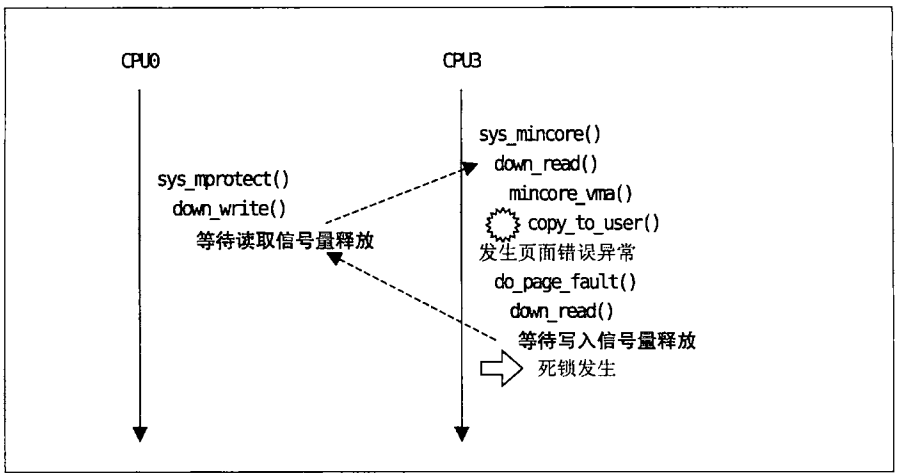


图 5-14 问题发生流程

复现测试

我们编写了一个复现用的程序。

查看内核代码，寻找对 mmap_sem 执行 down_write()的处理。结果发现在 mmap()的处理中执行了 down_write()。

这次的问题是 mincore()系统调用和 mmap()系统调用的冲突，满足竞态条件。

编写的程序代码如下所示。

236

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <pthread.h>

#define PAGESIZE    (4096)

void *th(void *p)
{
   unsigned char *ptr;

   for (;;) {
        ptr = mmap(NULL, PAGESIZE, PROT_READ|PROT_WRITE,
```

```
                MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
        *ptr = 0;
        munmap(ptr, PAGESIZE);
    }

    return NULL;
}

void do_mincore(void)
{
    unsigned char *vec;

    vec = mmap(NULL, PAGESIZE, PROT_READ|PROT_WRITE,
          MAP_SHARED|MAP_ANONYMOUS, -1, 0);

    for (;;) {
        if (mincore(vec, PAGESIZE, vec) < 0)
            perror("mincore");
    }

    munmap(vec, PAGESIZE);
}

int main(int argc, char **argv)
{
    pthread_t tid;
```

237

```
    pthread_create(&tid, NULL, th, NULL);

    do_mincore();

    return 0;
}
```

执行这个复现程序后，问题就发生了。

解决问题

首先确认一下社区有没有解决该问题的方法。

用复现程序在社区的 2.6.9 版内核中复现了该问题，而在最新的内核中该问题无法复现。因此，社区提供的内核应该已经改正了该问题。

查看 mm/mincore.c 的修改历史，找到了该问题的修改补丁。

```
commit 2f77d107050abc14bc393b34bdb7b91cf670c250
Author: Linus Torvalds <torvalds@woody.osdl.org>
Date:   Sat Dec 16 09:44:32 2006 -0800
  Fix incorrect user space access locking in mincore()

  Doug Chapman noticed that mincore() will doa "copy_to_user()" of the
  result while holding the mmap semaphore for reading, which is a big
  no-no. While a recursive read-lock on a semaphore in the case of a page
  fault happens to work, we don't actually allow them due to deadlock
  schenarios with writers due to fairness issues.

  Doug and Marcel sent in a patch to fix it, but I decided to just rewrite
  the mess instead - not just fixing the locking problem, but making the
  code smaller and (imho) much easier to understand.

  Cc: Doug Chapman <dchapman@redhat.com>
  Cc: Marcel Holtmann <holtmann@redhat.com>
  Cc: Hugh Dickins <hugh@veritas.com>
  Cc: Andrew Morton <akpm@osdl.org>
  Signed-off-by: Linus Torvalds <torvalds@osdl.org>
```

238

将这个补丁移植到测试中的内核上并利用复现程序测试，该问题就不再发生了。

总结

本 hack 介绍了内核信号量的死锁案例及其分析方法。分析的要点如下。

- 确认问题的详细情况。
- 从崩溃转储中收集分析数据。
- 用复现程序复现问题。

参考文献

- AMD64 Application Binary Interface
  http://www.x86-64.org/documentation/abi.pdf

——岛本裕志

实时进程停止响应

转储可以证实实时用户程序停止响应。

一般的进程停止响应的调试方法已在**"HACK#32 应用程序停止响应(死循环篇)"**中介绍了。本 hack 介绍实时进程停止响应时的调试方法。实时进程停止响应，会占据 CPU，导致系统整体停止响应，引发重大故障。本 hack 用的复现程序虽然简单，问题本身却是用户实际咨询的问题。内核版本为 2.6.9。

实时进程就是优先级比正常进程更高的进程。只要没有同级别或更高优先级的实时进程，并且自身没有主动睡眠（抢占式，preemption），就会一直使用 CPU。

首先要复现问题

首先要复现问题。下面是个仅执行死循环的 shell 脚本 `loop.sh`。`chrt` 命令可以将指定进程的调度策略设置为实时，99 是优先级，实时的优先级数字就是最大数字 99。

239

*为了获取转储，需要事先启用 IPMI watchdog（详情参见 **"HACK#22 死机时利用 IPMI watchdog timer 获取崩溃转储"**）。这次的现象是用户应用程序停止响应，因此无法用 NMI watchdog 获取转储。*

```
# cat loop.sh
#!/bin/bash
while [ 0 ] ; do
   ;
done
# chrt 99 ./loop.sh &
# ps aux
USER  PID %CPU %MEM   VSZ   RSS  TTY STAT START TIME COMMAND
...
root 28333 100  0.0 64624 1028 pts/0 R    23:10 4:48 /bin/bash ./loop.sh
       /* ^^^CPU使用率为100% */
...
```

死循环会持续消耗 CPU，因此 CPU 使用率为 100%，这样就能占据一个 CPU。本次的机器有两个 CPU，因此再执行一个 `loop.sh`，这样就能占据两个 CPU。

```
# chrt 99 ./loop.sh
```

这样 loop.sh 就占据了两个 CPU，系统停止了响应，键盘也无法输入。然后用 IPMI watchdog 的超时获取转储。

再查看 backtrace

实际发现故障时，首先要进行这一步。由于已获得了转储，先看看开头的信息。最先引起注意的是 COMMAND（进程）。

```
# crash vmlinux vmcore
...
     PANIC: ""
      PID: 4223
   COMMAND: "loop.sh"   /* panic 时运行的进程名 */
      TASK: 1007e3087f0 [THREAD_INFO: 10036144000]
       CPU: 0
...
```

240

可知 panic 时运行的进程是 loop.sh。

查看 backtrace（e1000 篇）

接下来用 bt 命令看看 backtrace。

```
crash> bt
PID: 4223  TASK: 1007e3087f0     CPU: 0  COMMAND: "loop.sh"
...
 #8 [ffffffff8045e3b0] error_exit at ffffffff80110d91
    [exception RIP: panic+211]
    RIP: ffffffff8013797a RSP: ffffffff8045e468 RFLAGS: 00010086
...
    ORIG_RAX: ffffffffffffffff CS: 0010 SS: 0000
 #9 [ffffffff8045e460] panic at ffffffff80137966
#10 [ffffffff8045e480] e1000_alloc_rx_buffers_ps at ffffffffa009bb77
#11 [ffffffff8045e4f0] sock_def_write_space at ffffffff802ad1d4
#12 [ffffffff8045e540] ipmi_wdog_pretimeout_handler at ffffffffa01c4892
#13 [ffffffff8045e550] ipmi_smi_watchdog_pretimeout at ffffffffa01ad927
#14 [ffffffff8045e570] handle_flags at ffffffffa01b51a0
#15 [ffffffff8045e590] smi_event_handler at ffffffffa01b54ca
...
```

panic()之前调用的函数好像是e1000驱动程序的函数。对此再进一步调查一下。首先用dis命令看看e1000_alloc_rx_buffers_ps at ffffffffa009bb77。

```
crash> dis e1000_alloc_rx_buffers_ps
...
0xffffffffa009baf4 <e1000_alloc_rx_buffers_ps+381>:  add      $0x12,%edi
0xffffffffa009baf7 <e1000_alloc_rx_buffers_ps+384>:  callq    0xffffffff802adb53
<alloc_skb> ——————————❶
0xffffffffa009bafc <e1000_alloc_rx_buffers_ps+389>:  test     %rax,%rax
...
0xffffffffa009bb70 <e1000_alloc_rx_buffers_ps+505>:  inc      %ebx
0xffffffffa009bb72 <e1000_alloc_rx_buffers_ps+507>:  callq    0xffffffff80120294
<dma_map_single>——————————❷
0xffffffffa009bb77 <e1000_alloc_rx_buffers_ps+512>:  mov      0x18(%rsp),%rdx
...
```

反汇编中的地址为e1000_alloc_rx_buffers_ps+512，实际上寄存器中的内容是下一个指令，因此panic时执行的是前一条指令，应当是上一行e1000_alloc_rx_buffers_ps+507❷的call指令，它调用了dma_map_single()，那我们来看看源代码。反汇编❶对应于源代码中的①，e1000_alloc_ rx_buffers_ps+507❷对应于下面的②。

241

```
[drivers/net/e1000/e1000_main.c]
e1000_alloc_rx_buffers_ps()
...
      skb = netdev_alloc_skb(netdev,
                    adapter->rx_ps_bsize0 + NET_IP_ALIGN); ——————————— ①

      if (unlikely(!skb)) {
...
      buffer_info->skb = skb;
      buffer_info->length = adapter->rx_ps_bsize0;
      buffer_info->dma = pci_map_single(pdev, skb->data, ——————————— ②
                                    adapter->rx_ps_bsize0,
                                    PCI_DMA_FROMDEVICE);
...

[include/asm-generic/pci-dma-compat.h]
static inline dma_addr_t
pci_map_single(struct pci_dev *hwdev, void *ptr, size_t size, int direction)
```

```
{
        return dma_map_single(hwdev == NULL ? NULL : &hwdev->dev, ptr, size, (enum
dma_data_direction)direction);
}
```

pci_map_single()被定义为 inline，因此在转储的 backtrace 中并不会显示为 call 0xffff...<pci_map_single>。pci_map_single()调用了 dma_map_single()。

```
[arch/x86_64/kernel/pci-gart.c]
dma_addr_t dma_map_single(struct device *dev, void *addr, size_t size, int dir)
{
        unsigned long phys_mem, bus;

        BUG_ON(dir == DMA_NONE);

        if (swiotlb)
...
```

242

的确，调用路径为 e1000_alloc_rx_buffers_ps() -> BUG() -> panic()。这种情况下 dma_map_single+16❸ 应当残留在 backtrace 中，但这次没有，因此应该不是在 e1000_alloc_rx_buffers_ps()中发生了 panic。

```
crash> dis dma_map_single
0xffffffff80120294 <dma_map_single>:     push  %rbp
0xffffffff80120295 <dma_map_single+1>:   cmp   $0x3,%ecx
0xffffffff80120298 <dma_map_single+4>:   mov   %rdi,%rbp
0xffffffff8012029b <dma_map_single+7>:   mov   %rdx,%rdi
0xffffffff8012029e <dma_map_single+10>:  push  %rbx
0xffffffff8012029f <dma_map_single+11>:  push  %rax
0xffffffff801202a0 <dma_map_single+12>:  jne   0xffffffff801202ae
<dma_map_single+26>
0xffffffff801202a2 <dma_map_single+14>:  ud2a
0xffffffff801202a4 <dma_map_single+16>:  mov   $0x70,%dh ————————————❸
0xffffffff801202a6 <dma_map_single+18>:  xor   0xffffffffffffffff(%rax),%al
...
```

backtrace 中还有 sock_def_write_space()，但并没有调用 panic()的路径。

确认 backtrace（IPMI 篇）

backtrace 的#12 中有个 ipmi_wdog_pretimeout_handler()，调查一下。

```
crash> dis ipmi_wdog_pretimeout_handler
0xffffffffa01c486f <ipmi_wdog_pretimeout_handler>:          push   %rax
0xffffffffa01c4870 <ipmi_wdog_pretimeout_handler+1>: cmpb   $0x0,19226(%rip)
# 0xffffffffa
01c9391
0xffffffffa01c4877 <ipmi_wdog_pretimeout_handler+8>: je     0xffffffffa01c48e3
0xffffffffa01c4879 <ipmi_wdog_pretimeout_handler+10>:movzbl 19216(%rip),%eax
# 0xffffffffa
01c9390
0xffffffffa01c4880 <ipmi_wdog_pretimeout_handler+17>:       cmp    $0x1,%al
0xffffffffa01c4882 <ipmi_wdog_pretimeout_handler+19>:       jne
0xffffffffa01c4892
0xffffffffa01c4884 <ipmi_wdog_pretimeout_handler+21>:       mov
$0xffffffffa01c4e6f,%rdi
0xffffffffa01c488b <ipmi_wdog_pretimeout_handler+28>:       xor    %eax,%eax
0xffffffffa01c488d <ipmi_wdog_pretimeout_handler+30>:callq  0xffffffff801378a7
<panic> ————————————❹
0xffffffffa01c4892 <ipmi_wdog_pretimeout_handler+35>:cmp    $0x2,%al
...
```

❹调用了 panic()，对应于下面源代码中的④。

243

```
[drivers/char/ipmi/ipmi_watchdog.c]
static void ipmi_wdog_pretimeout_handler(void *handler_data)
{
     if (preaction_val != WDOG_PRETIMEOUT_NONE) {
          if (preop_val == WDOG_PREOP_PANIC)
               panic("Watchdog pre-timeout"); ——————————————— ④-1
          else if (preop_val == WDOG_PREOP_GIVE_DATA) {
               spin_lock(&ipmi_read_lock);
               data_to_read = 1;
...
```

ipmi_wdog_pretimeout_handler()调用了 panic()（④-1），panic()中有个参数为字符串“Watchdog pre-timeout”。下面用 log 命令查看一下内核中的日志缓存。

```
crash> log
...
Kernel panic - not syncing: Watchdog pre-timeout ———————————— ④-2
---------- [cut here ] --------- [please bite here ] ---------
Kernel BUG at panic:75
invalid operand: 0000 [1] SMP
...
Call Trace:<IRQ> <ffffffffa009bb77>{:e1000:e1000_alloc_rx_buffers_ps+512}
  <ffffffff802ad1d4>{sock_def_write_space+18}
  <ffffffffa01c4892>{:ipmi_watchdog:ipmi_wdog_pretimeout_handler+35}
  <ffffffffa01c486f>{:ipmi_watchdog:ipmi_wdog_pretimeout_handler+0}
  <ffffffffa01ad927>{:ipmi_msghandler:ipmi_smi_watchdog_pretimeout+53}
  <ffffffffa01b51a0>{:ipmi_si:handle_flags+87}
  <ffffffffa01b54ca>{:ipmi_si:smi_event_handler+490}
  <ffffffffa01b58ad>{:ipmi_si:smi_timeout+72}
  <ffffffffa01b5865>{:ipmi_si:smi_timeout+0}
  <ffffffff80140115>{run_timer_softirq+356} <ffffffff8013c7c8>{__do_softirq+88}
  <ffffffff8013c871>{do_softirq+49} <ffffffff80110bf5>{apic_timer_interrupt+133}
   <EOI>
...
```

这里显示了源代码的字符串 Watchdog pre-timeout（④-2），这个转储似乎是 IPMI watchdog 中发生了 panic。

下面总结一下目前了解的情况。

1. 这次的现象为 IPMI watchdog 超时中发生了 panic。而且，据此可以推断内核或用户应用程序停止响应的可能性很高。

244

2. 由于是 IPMI watchdog，所以中断并没有被禁止。

确认运行中的进程

停止响应的原因大致可分为 4 种，内核和用户应用程序分别有死锁和死循环两种。

再次调查 e1000 的函数，并没有发现死循环或可能导致停止响应的代码，于是开始在应用程序中寻找停止响应的原因。首先用 ps 命令查看运行中的命令。

```
crash> ps
   PID   PPID  CPU     TASK      ST  %MEM    VSZ   RSS  COMM
```

```
...
> 4223  4046  0   1007e3087f0   RU  0.1  53532  1168 loop.sh
> 4224  4046  1   10057b147f0   RU  0.1  53532  1168 loop.sh
crash>
```

运行中的进程为 loop.sh。下面收集该进程的信息，task 命令可以查看优先级和进程调度策略。

```
crash> task 4223 | grep prio
  prio = 0,
  static_prio = 120,
  rt_priority = 99,
crash> task 4224 | grep prio
  prio = 0,
  static_prio = 120,
  rt_priority = 99,
crash> task 4223 | grep policy -w
  policy = 2,
crash> task 4224 | grep policy -w
  policy = 2,
```

两个 loop.sh 的优先级都是 99，可知是轮转调度（round robin）中的实时进程，来看看它们占用了多少 CPU。应用程序可以用 utime 查看，utime 的结果是在用户空间消耗的 CPU 时间（即 time(1)命令的“user”值）。调度器会更新该值（查看内核空间 CPU 占用的命令为 stime）。

245

```
crash> task 4223 | grep utime
  utime = 50303,
crash> task 4224 | grep utime
  utime = 50252,
```

用 ps 命令查看 utime 很方便。下面是用 ps 命令查看 CPU 消耗时间的结果，与其他进程比较一下。

```
crash> ps -t
...
PID: 4044  TASK: 10037d25030     CPU: 1  COMMAND: "sshd"
    RUN TIME: 00:27:31
  START TIME: 266
   USER TIME: 203
```

```
 SYSTEM TIME: 283
PID: 4046  TASK: 100793aa7f0     CPU: 1  COMMAND: "bash"
    RUN TIME: 00:27:30
  START TIME: 267
   USER TIME: 53
 SYSTEM TIME: 30
...
PID: 4223  TASK: 1007e3087f0     CPU: 0  COMMAND: "loop.sh"
    RUN TIME: 00:00:55
  START TIME: 1862
   USER TIME: 50303
 SYSTEM TIME: 2628
PID: 4224  TASK: 10057b147f0     CPU: 1  COMMAND: "loop.sh"
    RUN TIME: 00:00:54
  START TIME: 1863
   USER TIME: 50252
 SYSTEM TIME: 2651
...
```

utime 是 USER TIME，单位为毫秒。这次 IPMI watchdog 的 pretimeout 为 30，timeout 为 90，因此停止响应 60 秒之后发生了 panic，而 watchdog 守护进程的 interval 为 10 秒，因此在停止响应之后 50～60 秒之间发生了 panic。sshd、bash 连 1 秒的时间都没有，而 loop.sh 光是在用户空间就占用了 50 秒，因此并没有进行进程调度，loop.sh 一直在运行，也就是说，可以判断为死循环。

246

根据内核转储分析应用程序就到此为止了。要分析应用程序方面的原因，需要用到其他方法，如利用 strace（参见 **“HACK#43 使用 strace 寻找故障原因的线索”**）、GDB（**“HACK#5 调试器（GDB）的基本用法（之一）”**）进行复现等。另外，loop.sh 执行之后立即停止响应，因此 watchdog 运行了 55 秒，如果是长时间持续运行的进程，就要再想其他办法，如跟其他进程进行比较等。如果是死循环，其 utime 应该是个不自然的值，但是如果在循环中进行了进程调度，就不能过于信赖这个值。

总结

本 hack 介绍了在内核转储中查看 utime 的结果，发现实时进程死循环的例子。

实际上，同样的现象发生时，e1000 的函数在 backtrace 中反复出现，因此客户曾询问是否在 e1000 中发生了 panic。这次为了复现该问题，故意增加网络负载，故

意让e1000函数在backtrace中显示。像这次的例子，栈中残留了以前的信息，在这种状态下获取转储后，可能会显示跟backtrace完全没有关系的符号。

——大岩尚宏

运行缓慢的故障

本hack介绍在内核版本升级时发生的MTD设备写入速度缓慢的调试实例。

内核版本升级之后的异常

这里想介绍的是将某个MTD设备(闪存)中的操作系统的内核由2.6.9升级到2.6.18之后发生的故障。用下述命令向MTD设备写入128KB数据，以前只需几秒就可以完成，这次几分钟也没能结束，而且也没有输出任何错误信息。

```
# dd if=a.dat of=/dev/mtd0 bs=131072 count=1
```

列举问题原因并排查

首先，为了理解发生了什么问题，我们确认了以下两点。这是因为在问题发生时，只要确认了这两点，很多情况下都能获得与问题有关的线索。

247

- 画面上有没有显示错误信息。
- /var/log/messages、/var/log/syslog中有没有进程、守护进程或内核的错误信息、警告信息。

但是笔者确认之后，发现并没有这些信息。话虽如此，什么信息也没有也是一种信息。也就是说，进程或内核并没有认识到执行的处理产生了错误。接下来，笔者考虑是否发生了以下情况。

a. 某处陷入死循环（busy loop）。
b. 发生某种繁忙，导致反复重试。
c. 进程在等待某个事件（等待信号、正在睡眠等）。
d. 死锁。
e. SIGSTOP等导致进程停止。

根据表 5-2 的 CPU 使用率、进程状态，可以大致地判断问题原因。

表 5-2 根据 CPU 使用率和进程状态判断问题

可能性	CPU 使用率	进程状态
a.	高（几乎为 100%）	R
b.	可能为各种情况	R 或者 S
c.	低（几乎为 0）	S 或者 D
d.	低（几乎为 0）	S 或者 D
e.	0	T

我用 top 命令检查了 CPU 使用率，结果发现 CPU 使用率几乎为 0。接下来用 ps 命令查看进程（dd）的状态，结果如下所示，进程处于中断禁止的睡眠状态（左起第 3 列为进程状态）。

```
# ps ax | grep dd
25921 pts/3  D+  0:00 dd if /dev/mtd0 of a.dat bs 131072 count 1
```

248

反复执行几次该命令，结果一直是 D 状态。也就是说，可以怀疑是上述 c.或 d.的情况。这两种情况（如死锁）无论在进程代码还是在内核代码中，都有可能发生。那么，接下来调查一下引起该问题的原因，究竟是进程（dd）的代码还是内核的代码。

我们用 GDB attach 到进程上获取 backtrace。

```
# gdb -p `pidof dd`
...
(gdb) bt
#0 0x000000309eec0e60 in __write_nocancel () from /lib64/libc.so.6
#1 0x0000000000401fa7 in iwrite (fd=1, buf=0x10169000 "", size=512) at dd.c:782
#2 0x000000000040200b in write_output () at dd.c:808
#3 0x0000000000403425 in main (argc=<value optimized out>, argv=<value optimized out>)
at dd.c:1294
```

本来，只有检查了上述所有函数的详细行为之后才能做出判断，但这个 backtrace 中并不包含死锁或执行睡眠的函数（pthread_mutex_lock()、wait()、sleep()等函数），而且 dd 这个进程显然是个单线程进程，可以认为引发死锁的可能性非常低。

```
(gdb) i thr
  1 Thread 46912496307920 (LWP 10070) 0x000000309eec0e60 in __write_nocancel () from
```

```
/lib64/libc.so.6
```

此外，我们还用另一种方法调查问题出在进程中还是内核中，那就是使用 strace，strace 工具可以跟踪进程执行的系统调用。对于运行中的进程，也可以使用下述方法用 strace 进行跟踪。关于 strace 的详细内容请参见 **"HACK#43 使用 strace 寻找故障原因的线索"**。

```
# strace -t -p `pidof dd`
Process 10070 attached - interrupt to quit
14:49:14 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512) = 512
14:49:14 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512) = 512
<<在该状态下暂停 13 秒>>
14:49:27 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512) = 512
14:49:27 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512
<<在该状态下暂停 13 秒>>
) = 512
...
<<之后 read()和write()反复被调用>>
```

249

从上述跟踪结果可以看出，write 系统调用写入 512B 内容需要花费大约 13 秒，直观上来看这段时间太长了。硬盘写入的吞吐量大约是几 MBps 到几十 MBps，即便是现在使用的 MTD 设备，写入也有几百 KBps。因此，512B 的写入本应在毫秒级别完成。另外应当注意，尽管花费了这么长时间，系统调用本身却是成功的。

从上述 GDB 和 strace 的调查结果可知，与进程的代码相比，内核内部出现问题的可能性更大，进一步说，write 系统调用上出现问题的可能性更大。

进程处于 D 状态（禁止中断的睡眠状态）时，如上例用 GDB 或 strace attach 之后，有可能发生 GDB 或 strace 似乎停止响应的状况，这是因为这些命令启动后会等待被 attach 的进程进入跟踪状态，但进程由于在内核内部发生死锁等原因，已处于无法执行的状态，因此无论多久也无法进入跟踪状态。此时，内核内部引发问题的可能性更大，因此可以跳过 GDB 或 strace 这一步，直接用下面的 crash 进行调查更好。

接下来用 crash 命令看看内核内部的情况。从 strace 的调查结果可知，write 系统调用(c)被调用后，又调用了 MTD 设备写入函数 mtd_write(B)。

```
crash> bt -t 4313
PID: 4313  TASK: ffff81022760d040 CPU: 1  COMMAND: "dd"
```

```
        START: thread_return (schedule) at ffffffff80061f29
[ffff81021b073cf0] schedule_timeout at ffffffff80062839
[ffff81021b073d10] process_timeout at ffffffff8009409f
[ffff81021b073d28] inval_cache_and_wait_for_operation at ffffffff883d31d6
[ffff81021b073d40] msleep at ffffffff80094768 ———————————— (A)
[ffff81021b073d50] inval_cache_and_wait_for_operation at ffffffff883d3332
[ffff81021b073d88] default_wake_function at ffffffff8008986e
[ffff81021b073da0] __wake_up at ffffffff8002e04d
[ffff81021b073de0] do_write_oneword at ffffffff883d5ad8
[ffff81021b073e30] cfi_intelext_write_words at ffffffff883d6e05
[ffff81021b073e80] mtd_write at ffffffff883b34ef ———————————— (B)
[ffff81021b073eb0] do_mmap_pgoff at ffffffff8000dc60
[ffff81021b073f10] vfs_write at ffffffff80016233
[ffff81021b073f40] sys_write at ffffffff80016b00 ———————————— (C)
[ffff81021b073f80] system_call at ffffffff8005c116
...
```

250

继续往下看会发现，(A)的 msleep()执行了睡眠操作。要想了解这里为何睡眠，就要阅读代码。阅读代码后发现，这个 msleep()是由名为 inval_cache_and_wait_for_operation()的函数调用的。inval_cache_and_wait_for_operation()函数在 do_write_oneword()函数中写入一个字（该芯片下一个字为两个字节）之后被调用，等待写入完成。另外，调查内核 2.6.9 的相应部分后发现并不存在该函数，而是用其他算法等待写入完成。

调查源代码

```
for (;;) {
    status = map_read(map, cmd_adr); ———————————————————— ①
    if (map_word_andequal(map, status, status_OK, status_OK)) ———— ②
      break;

    if (!timeo) { ———————————————————————————————————— ③
      map_write(map, CMD(0x70), cmd_adr);
      chip->state = FL_STATUS;
      return -ETIME;
    }

    /* OK Still waiting. Drop the lock, wait a while and retry. */
```

```
        spin_unlock(chip->mutex);
        if (sleep_time >= 1000000/HZ) { ——————————————————④
            /*
             * Half of the normal delay still remaining
             * can be performed with a sleeping delay instead
             * of busy waiting.
             */
            msleep(sleep_time/1000); ——————————————————⑤
            timeo -= sleep_time;
            sleep_time = 1000000/HZ;
        } else { ——————————————————⑥
251         udelay(1);
            cond_resched();
            timeo--;
        }
        spin_lock(chip->mutex);

        if (chip->state != chip_state) {
            /* Someone's suspended the operation: sleep */
            "省略"
        }
    }
```

①读取闪存设备写入是否完成的信息，再由②判断写入是否完成。如果写入完成，就离开 for 循环并返回。③是超时判断，如果等待很长时间也没写完，就返回错误并结束函数。查看源代码后发现，该函数调用的 do_write_oneword()在发生错误时会显示内核信息。而 dd 命令执行过程中并没有显示这种内核信息，可以认为这里不可能超时。

④和⑥的代码块是等待写入完成的处理，执行④还是执行⑥是由 sleep_time 变量的值决定的。sleep_time 在⑤处除以 1000，并作为 msleep()的参数使用，因此给 sleep_time 传递的睡眠时间应该是以微秒为单位的。该系统中 HZ 为 1000，因此④处的 sleep_time 判断是否在 1000 微秒以上。根据睡眠时间进行分支的原因是 Linux 睡眠（定时器）的精确度就是 1000 微秒（1 毫秒）。也就是说，睡眠时间超过 1 毫秒，用 msleep()等基本上能在经过了指定的时间后唤醒进程。但如果不足 1 毫秒，就只能用 busy loop 函数和 udelay()等调整时间。

那么，这里 sleep_time 被设置为多少呢？如上述代码开头所示，它被设置为

chip_op_time 的 1/2。调查代码后发现，chip_op_time 来源于保存闪存芯片信息的结构 struct flchip 中的 word_write_time 成员，进一步调查代码发现，word_write_time 成员的初始值为 50000（单位为微秒）。也就是说，sleep_time 的值（即睡眠时间）就是它的 1/2，25 毫秒。

252

假说提出和验证

可以考虑的是，1 个字（2B）的数据写完之后，①处设备返回的讯息仍在写入，因此⑤处执行睡眠 25 毫秒。每次的睡眠时间非常短，但写入 128KB 却需要睡眠 64×1024 次，折合成时间大约为 20 分钟，这与几分钟都写不完的事实是吻合的。

那么，这里真的有必要等待 25 毫秒吗？说起来，在内核 2.6.9 中瞬间就能完成写入，因此等待时间再短些应该也没问题。从闪存设备的 datasheet 来看，1 个字的平均写入时间为 10 微秒，最多只需 200 微秒，这样看来，写入 1 个字只需花费 10 微秒，之后却要等待 25 毫秒后才会确认它是否写入完成。

另外，从 MTD 设备初始化时显示的内核信息可知，闪存设备的型号为 M50FW080。

```
kernel: Found: ST M50FW080
```

此外，包括此设备在内的许多闪存设备的 datasheet 都可以在厂商主页上找到。

为验证这个假说，我们将 word_write_time 的初始值设置为比最大写入时间稍长，即 256 微秒（sleep_time 就是其 1/2，为 128 微秒）并编译内核。编译后的内核中用 dd 命令写入时，只需几秒钟就可以写完，与内核 2.6.9 基本相同。因此，原因就是等待时间的初始值过长。

总结

本 hack 介绍了内核版本升级时发生的 MTD 设备写入速度降低的调试案例。

253

CPU 负载过高的故障

我们曾经遇到过一个故障，一旦在使用 VLAN 的网络上进行 TCP 通信，网卡利用硬件计算校验和的功能就无法正常工作。以此为例介绍一下调试方法。

Intel 的网络设备拥有计算 TCP 包的校验和的功能，以降低 CPU 的负载。Linux 内

核首先判断网络设备是否支持此功能，支持则通过硬件计算校验和，内核（软件）就不再计算。我们发现，一般的网络通信可以正常使用硬件校验和计算，但在 VLAN 设备上进行 TCP 通信时，内核不会进行校验和计算。这是从实际的客户咨询中发现的问题。

本 hack 介绍了发现和修正方法。使用的是 Red Hat 家族的发行版，内核为 2.6.18。

故障复现的准备

首先建立 VLAN 设备，此处为 eth3.510。用 vconfig 命令也可以建立，但这里修改了配置文件，以便能在系统启动时建立。

```
# vi /etc/sysconfig/network-scripts/ifcfg-eth3.510
DEVICE=eth3.510
...
VLAN=yes
...
```

务必与 eth3 放在不同的网段中。

```
# ifconfig
...
eth3    Link encap:Ethernet  HWaddr 00:15:17:3A:61:09
        inet addr:192.168.1.200  Bcast:192.168.1.255  Mask:255.255.255.0
...
eth3.510  Link encap:Ethernet  HWaddr 00:15:17:3A:61:09
        inet addr:192.168.0.200  Bcast:192.168.0.255  Mask:255.255.255.0
...
#
```

为了评测网络性能，我们准备了两台机器，一台用于发送 TCP 包的 sender，一台用于接收的 receiver。环境配置如图 5-15 所示。

254

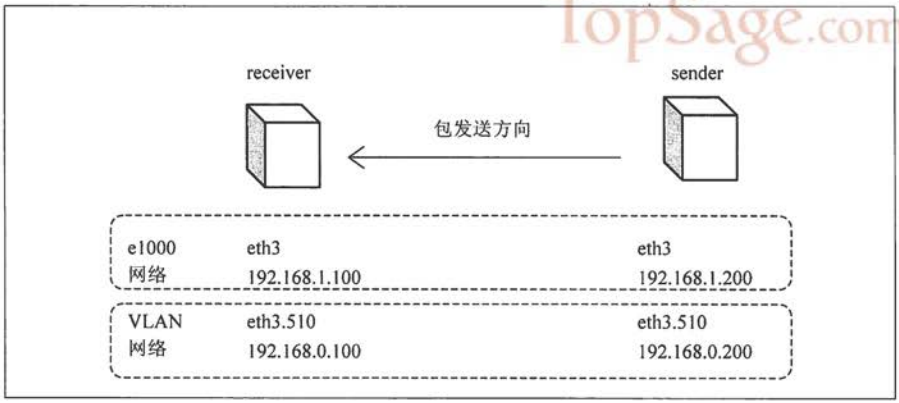


图 5-15 网络性能评测的环境

设置好 eth3 和 eth3.510 之后，通过 ping 命令等确认 receiver 和 sender 之间能够正常通信。

使用 nuttcp 测量吞吐量

下面测量普通的网络设备和 VLAN 设备的吞吐量，这里使用 nuttcp。nuttcp 是个 TCP/UDP 网络测试工具，可以测量吞吐量。

```
# wget -t0 -c http://www.lcp.nrl.navy.mil/nuttcp/nuttcp-5.5.5.tar.bz2
# tar jxvf nuttcp-5.5.5.tar.bz2
# cd nuttcp-5.5.5
# gcc -O2 -o nuttcp nuttcp-5.5.5.c
```

在 receiver 上通过以下选项启动 nuttcp 服务器。

```
[receiver]# ./nuttcp -S
```

同样在 sender 上执行 nuttcp 发包，用-n 选项设置发送的总数据量为 1GB。为使数据包分段，可以通过-l 选项设置数据写入长度为 1500 字节。

```
[sender]# ./nuttcp -n1G -l1500 192.168.1.100        /* 一般网络 */
 1023.9987 MB /  9.12 sec = 941.3975 Mbps 12 %TX 18 %RX
[sender]# ./nuttcp -n1G -l1500 192.168.0.100        /* VLAN 网络 */
 1023.9987 MB /  9.15 sec = 938.5309 Mbps 22 %TX 19 %RX
```

255

开头的数据为发送的数据大小，通过选项设置成发送 1GB。下一个值是发送数据所需的时间，接下来是吞吐量（Mbps）。VLAN 网络的吞吐量稍稍低一些。此外，%TX

和%RX为发送进程（sender）和接收进程（receiver）的CPU使用率。ps等命令中的CPU使用率是进程生存期间使用的CPU时间百分比，但nuttcp是根据发送接收信息之前到发送接收信息完成之后的时间段中，所用的CPU时间（用户时间+内核时间）计算的百分比。可见，一般网络中发送进程的CPU使用率（%TX）为12%，而VLAN网络中该值为22%。尽管吞吐量没有太大差异，但CPU使用率却增加了10%。

接下来使用oprofile，比较一下究竟哪里产生了额外开销。

使用oprofile确认额外开销

使用oprofile时，需要用到vmlinux来解析（resolve）符号（symbol）。安装kernel-debuginfo这个RPM包之后，vmlinux就位于/usr/lib/debug/lib/modules/2.6.18/vmlinux，将它用到opcontrol命令上。

在nuttcp发送信息的状态下，执行opcontrol命令（实际上要用脚本执行）。

```
[sender]# opcontrol --init
[sender]# opcontrol --start --vmlinux=/boot/vmlinux-2.6.18
[sender]# ./nuttcp -T20s -l1500 192.168.0.100 &     /* VLAN网络 */
[sender]# sleep 10
[sender]# opcontrol --stop
```

首先对VLAN网络进行测量，用opreport命令可以看到简单的结果。从下面可以看出，内核运行得最多，其次是e1000e。

```
[sender]# opreport
...
 samples|      %|
------------------
   59056 81.6808 vmlinux-2.6.18
   10712 14.8158 e1000e
    1053  1.4564 nuttcp
     869  1.2019 libc-2.5.so
     243  0.3361 oprofiled
     119  0.1646 bash
      87  0.1203 8021q
      50  0.0692 oprofile
...
```

用 opreport 命令也可以看到详细点的信息。

```
[sender]# opreport -l
...
samples %      app name        symbol name
11049   15.2819 vmlinux-2.6.18 csum_partial_copy_generic
10712   14.8158 e1000e         (no symbols) ——————————————————①
3093     4.2779 vmlinux-2.6.18 tcp_sendmsg
2800     3.8727 vmlinux-2.6.18 kfree
2627     3.6334 vmlinux-2.6.18 tcp_init_tso_segs
2140     2.9598 vmlinux-2.6.18 skb_clone
2133     2.9502 vmlinux-2.6.18 kmem_cache_free
1866     2.5809 vmlinux-2.6.18 tcp_ack
1805     2.4965 vmlinux-2.6.18 cache_grow
1554     2.1493 vmlinux-2.6.18 mwait_idle
1539     2.1286 vmlinux-2.6.18 tcp_v4_rcv
1473     2.0373 vmlinux-2.6.18 tcp_transmit_skb
1383     1.9128 vmlinux-2.6.18 __kfree_skb
1382     1.9115 vmlinux-2.6.18 eth_header
1166     1.6127 vmlinux-2.6.18 __alloc_skb
1139     1.5754 vmlinux-2.6.18 dev_queue_xmit
1072     1.4827 vmlinux-2.6.18 cache_alloc_refill
1047     1.4481 vmlinux-2.6.18 ip_queue_xmit
1004     1.3886 vmlinux-2.6.18 tcp_v4_send_check
949      1.3126 vmlinux-2.6.18 system_call
...
```

①的符号名为(no symbols)，因此要指定-p 选项以解析符号。下面的命令执行后会显示警告信息，但本例中没什么问题。

```
[sender]# opreport -l -p /lib/modules/2.6.18/kernel/
...
samples %       image name      app name        symbol name
11049   15.2819 vmlinux-2.6.18  vmlinux-2.6.18  csum_partial_copy_generic
3093     4.2779 vmlinux-2.6.18  vmlinux-2.6.18  tcp_sendmsg
2800     3.8727 vmlinux-2.6.18  vmlinux-2.6.18  kfree
2627     3.6334 vmlinux-2.6.18  vmlinux-2.6.18  tcp_init_tso_segs
2170     3.0013 e1000e.ko       e1000e          e1000_clean_tx_irq ——————②
2155     2.9806 e1000e.ko       e1000e          e1000_xmit_frame ———————③
```

257

```
2140    2.9598 vmlinux-2.6.18 vmlinux-2.6.18 skb_clone
2133    2.9502 vmlinux-2.6.18 vmlinux-2.6.18 kmem_cache_free
1866    2.5809 vmlinux-2.6.18 vmlinux-2.6.18 tcp_ack
1830    2.5311 e1000e.ko      e1000e         e1000_irq_enable ————————④
1805    2.4965 vmlinux-2.6.18 vmlinux-2.6.18 cache_grow
1694    2.3430 e1000e.ko      e1000e         e1000_intr_msi ————————⑤
1642    2.2711 e1000e.ko      e1000e         e1000_clean_rx_irq ————————⑥
1554    2.1493 vmlinux-2.6.18 vmlinux-2.6.18 mwait_idle
1539    2.1286 vmlinux-2.6.18 vmlinux-2.6.18 tcp_v4_rcv
1473    2.0373 vmlinux-2.6.18 vmlinux-2.6.18 tcp_transmit_skb
1383    1.9128 vmlinux-2.6.18 vmlinux-2.6.18 __kfree_skb
1382    1.9115 vmlinux-2.6.18 vmlinux-2.6.18 eth_header
1166    1.6127 vmlinux-2.6.18 vmlinux-2.6.18 __alloc_skb
1139    1.5754 vmlinux-2.6.18 vmlinux-2.6.18 dev_queue_xmit
1072    1.4827 vmlinux-2.6.18 vmlinux-2.6.18 cache_alloc_refill
1047    1.4481 vmlinux-2.6.18 vmlinux-2.6.18 ip_queue_xmit
1004    1.3886 vmlinux-2.6.18 vmlinux-2.6.18 tcp_v4_send_check
949     1.3126 vmlinux-2.6.18 vmlinux-2.6.18 system_call
...
```

解析符号之后，就能看出 e1000e 的各个函数的采样比例。①的 14.8%为②、③、④……⑥……的总和。接下来用同样的方法测量普通网络的情况。下面是普通网络的结果。

```
[sender]# ./nuttcp -T20s -l1500 192.168.1.100 &
...
[sender]# opreport -l -p /lib/modules/2.6.18/kernel/
...
samples  %        image name        app name        symbol name
8658    16.7839 vmlinux-2.6.18  vmlinux-2.6.18  copy_user_generic ————————⑦
2509     4.8638 vmlinux-2.6.18  vmlinux-2.6.18  tcp_sendmsg
2048     3.9701 e1000e.ko        e1000e          e1000_irq_enable
1941     3.7627 e1000e.ko        e1000e          e1000_xmit_frame
1803     3.4952 vmlinux-2.6.18   vmlinux-2.6.18   mwait_idle
1738     3.3692 e1000e.ko        e1000e          e1000_intr_msi
1566     3.0358 e1000e.ko        e1000e          e1000_clean_rx_irq
1424     2.7605 vmlinux-2.6.18   vmlinux-2.6.18   tcp_v4_rcv
1342     2.6015 vmlinux-2.6.18   vmlinux-2.6.18   ip_output
```

258

```
1295    2.5104      vmlinux-2.6.18    vmlinux-2.6.18    kfree
1098    2.1285      vmlinux-2.6.18    vmlinux-2.6.18    __tcp_push_pending_frames
1072    2.0781      vmlinux-2.6.18    vmlinux-2.6.18    kmem_cache_free
983     1.9056      vmlinux-2.6.18    vmlinux-2.6.18    system_call
974     1.8881      vmlinux-2.6.18    vmlinux-2.6.18    tcp_ack
800     1.5508      vmlinux-2.6.18    vmlinux-2.6.18    tcp_transmit_skb
772     1.4966      vmlinux-2.6.18    vmlinux-2.6.18    skb_split
768     1.4888      e1000e.ko         e1000e            e1000_clean_tx_irq
724     1.4035      vmlinux-2.6.18    vmlinux-2.6.18    skb_clone
721     1.3977      vmlinux-2.6.18    vmlinux-2.6.18    ip_queue_xmit
694     1.3454      nuttcp            nuttcp            Nwrite
691     1.3395      libc-2.5.so       libc-2.5.so       __write_nocancel
690     1.3376      vmlinux-2.6.18    vmlinux-2.6.18    dnotify_parent
678     1.3143      vmlinux-2.6.18    vmlinux-2.6.18    put_page
...
31      0.0601 vmlinux-2.6.18 vmlinux-2.6.18 copy_from_user ————————⑨
```

与 VLAN 网络相比，第 1 个⑦的位置不同，VLAN 网络的情况下为 csum_partial_copy_generic()函数。该函数的调用路径如图 5-16 所示。

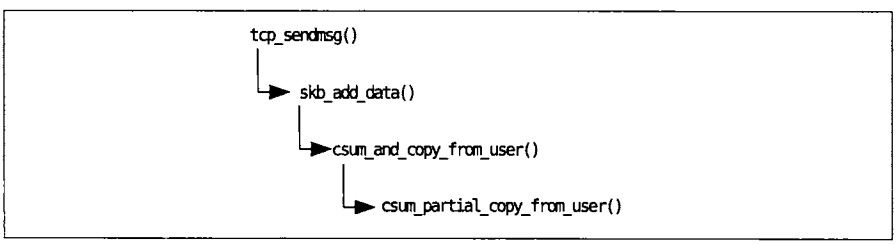


图 5-16 VLAN 网络中 csum_partial_copy_generic()函数的调用顺序

```
[include/linux/skbuff.h]
static inline int skb_add_data(struct sk_buff *skb,
                    char __user *from, int copy)
{
      const int off = skb->len;
      if (skb->ip_summed == CHECKSUM_NONE) { /* 发送信息时使用的NIC
            int err = 0;                        不支持校验和计算功能 */
            unsigned int csum = csum_and_copy_from_user(from,
                                                  skb_put(skb, copy),
                                                  copy, 0, &err);
            }
```

259

```
        } else if (!copy_from_user(skb_put(skb, copy), from, copy))————————⑧
              return 0;
    ...
```

skb_add_data()函数的作用是将用户需要发送的数据转给内核（数据包缓冲区）。在 CHECKSUM_NONE 的情况下，即 NIC 不支持校验和计算功能的情况下，执行 csum_partial_copy_generic()。该函数从用户空间传递数据，同时计算校验和。该函数被调用了，说明校验和是由软件（内核）计算的。

而在普通网络中，由于使用了硬件的校验和计算功能，从 opreport 命令结果的⑨中可以看出，源代码中的⑧几乎没有任何负载。

设置硬件校验和计算功能的原理

接下来要探寻 skb->ip_summed 值为 CHECKSUM_NONE 的原因。e1000e 驱动程序在初始化时获取设备信息，因此应当检查一下内核加载 e1000e 驱动程序时，识别硬件校验和计算功能的原理（图 5-17）。

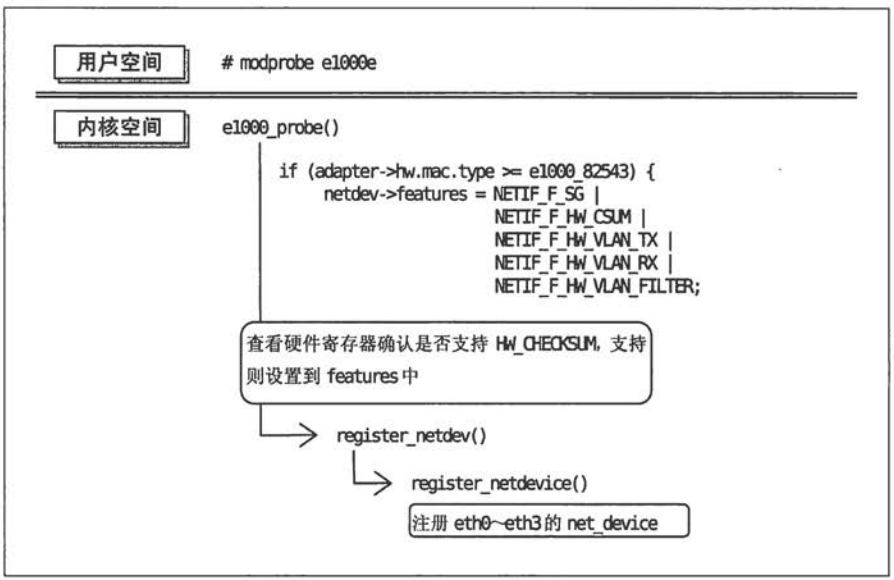


图 5-17 e1000e 设备的建立顺序

260

features 表明了网络设备所支持的特性，这里所说的特性即校验和计算、VLAN 等。features 可以通过 sysfs 确认。

```
[sender]# cat /sys/class/net/eth3/features
0x1113a9
[sender]# cat /sys/class/net/eth3.510/features
0x0
```

features 为标志位的组合，其值由内核的 include/linux/netdevice.h 定义。

```
[include/linux/netdevice.h]
...
     unsigned long          features;
#define NETIF_F_SG          1     /* Scatter/gather IO. */
#define NETIF_F_IP_CSUM     2     /* Can checksum only TCP/UDP over IPv4. */
#define NETIF_F_NO_CSUM     4     /* Does not require checksum. F.e. loopack. */
#define NETIF_F_HW_CSUM     8     /* Can checksum all the packets. */
#define NETIF_F_HIGHDMA     32    /* Can DMA to high memory. */
#define NETIF_F_FRAGLIST    64    /* Scatter/gather IO. */
...
```

eth3（普通网络）上 NETIF_F_HW_CSUM 有效，而在 VLAN 网络的 eth3.510 为 0x0，所有标志均被设置为无效。因此，下一步要确认 VLAN 设备的初始化（图 5-18）。

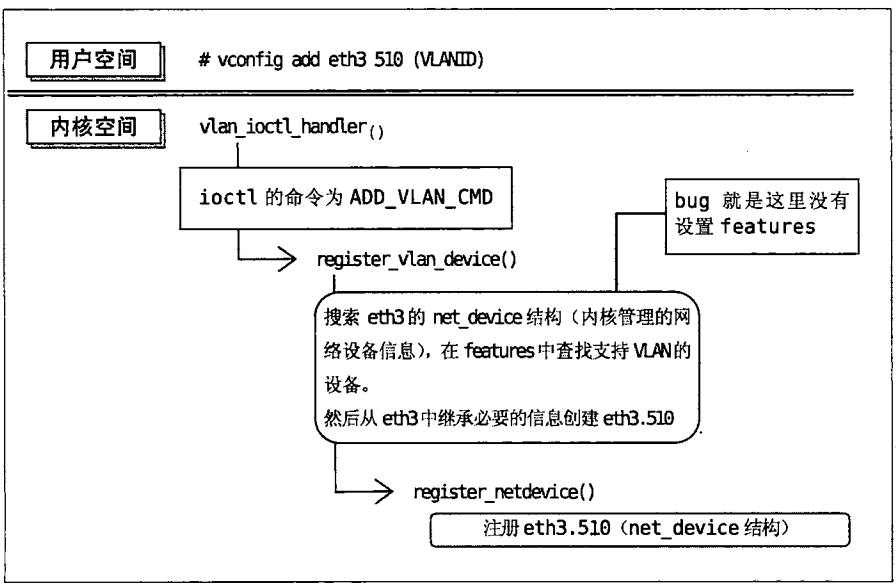


图 5-18 VLAN 设备的建立顺序

VLAN 设备没有继承物理设备 eth3 的 features，因此 eth3.510 的 features 成了 0x0。

261

上面说的是 features。接下来通过系统调用的处理，检查由于 skb->ip_summed== CHECKSUM_NONE 而执行 csum_partial_copy_generic()的原理。图 5-19、图 5-20 总结了所有系统调用。

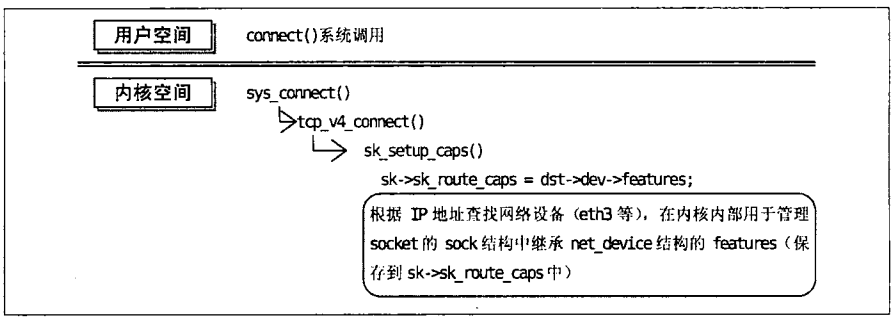


图 5-19 connect()系统调用处理

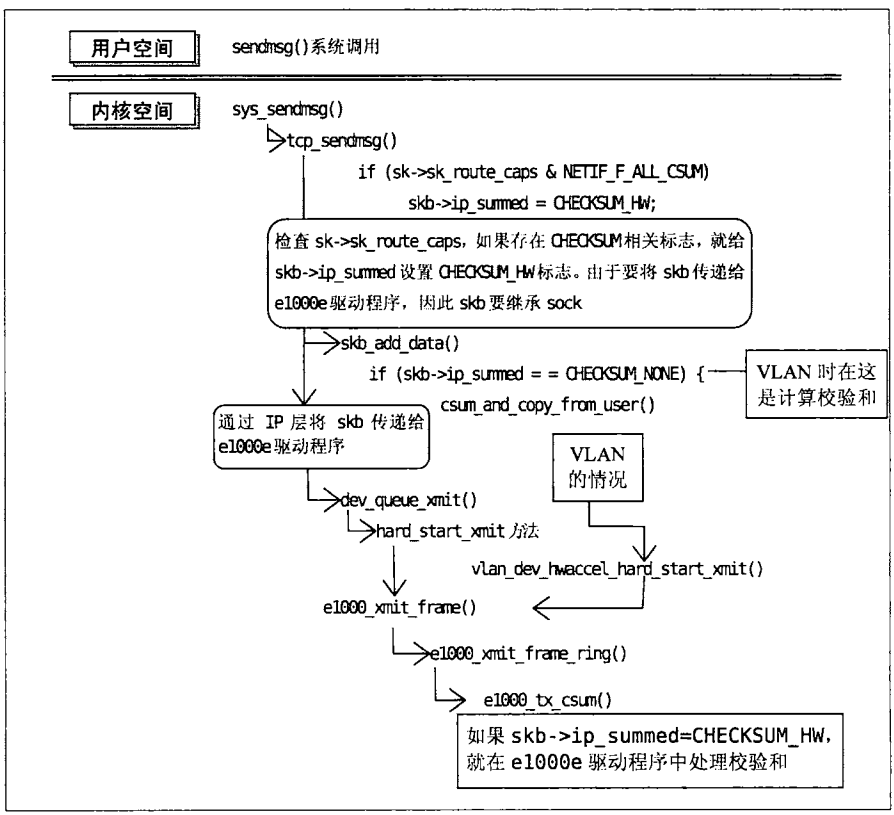


图 5-20 sendmsg()系统调用处理

262

总结到目前为止的调查结果可见，e1000e 驱动程序被加载时会检查硬件并给 net_device 结构的 features 设置 NETIF_F_HW_CSUM 标志位。connect()函数将 net_device 结构的 features 设置到 sk->sk_route_caps 上。sendmsg()检查 sk->sk_route_caps，设置 skb->ip_summed 为 CHECKSUM_HW。

从 VLAN 设备发送 TCP 数据包时内核执行校验和计算的原因是，eth3 的 features 没能继承到 eth3.510 的 socket 缓冲区中。

社区的修改

探明原因之后，我们给负责网络的社区（netdev）发了封邮件，该 bug 立即就被修正了。

下面是修正补丁，改成了在 register_vlan_device()函数中设置 features。

```
commit 5fb13570543f4ae022996c9d7c0c099c8abf22dd
Author: Patrick McHardy <kaber@trash.net>
Date:  Tue May 20 14:54:50 2008 -0700

    [VLAN]: Propagate selected feature bits to VLAN devices

commit 289c79a4bd350e8a25065102563ad1a183d1b402
Author: Patrick McHardy <kaber@trash.net>
Date:  Fri May 23 00:22:04 2008 -0700

    vlan: Use bitmask of feature flags instead of seperate feature bits
```

也有人提议在 e1000e 驱动程序中设置 VLAN 设备的 features（下面的补丁），但这样就只有 e1000e 驱动程序才能支持该功能，考虑到 ethtool 命令，最后还是把上面的补丁合并到主线上（Intel 网站的 e1000 驱动程序中包含了该补丁）。

```
[PATCH 4/4] e1000e: Allow TSO to trickle down to VLAN device
http://www.spinics.net/lists/netdev/msg63716.html
```

263

修改后的测试

首先检查打了补丁的内核中的 features。

```
# cat /sys/class/net/eth3.510/features
0x10009
```

可见NETIF_F_HW_CSUM标志是有效的。

接下来使用nuttcp发送数据，用oprofile查看结果。不出所料，csum_partial_copy_generic()函数不再出现了。

```
# opreport -l -p /lib/modules/2.6.18vlan/kernel/
...
samples  %        image name               app name                 symbol name
26955    23.3575  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       copy_user_generic
6554     5.6793   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       mwait_idle
5375     4.6576   e1000e.ko                e1000e                   e1000_irq_enable
4681     4.0563   e1000e.ko                e1000e                   e1000_intr_msi
4148     3.5944   e1000e.ko                e1000e                   e1000_clean_rx_irq
3770     3.2668   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       tcp_v4_rcv
3494     3.0277   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       tcp_sendmsg
3436     2.9774   e1000e.ko                e1000e                   e1000_xmit_frame
2523     2.1863   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       tcp_ack
2478     2.1473   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       kfree
2079     1.8015   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       kmem_cache_free
1715     1.4861   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       put_page
1554     1.3466   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       __tcp_push_pending_frames
1494     1.2946   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       IRQ0x42_interrupt
1462     1.2669   e1000e.ko                e1000e                   e1000_clean_tx_irq
1322     1.1456   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       __kfree_skb
1206     1.0450   vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       ip_rcv
...
```

与预期的相同，采样结果中没有csum_partial_copy_generic()。

接下来使用同样的选项再次执行nuttcp，将打补丁前后的内核分别执行10次，平均值总结如表5-3所示。Mbps为nuttcp的吞吐量，sec为nuttcp的时间，%TX为发送进程（sender）的CPU使用率。

264

表5-3 2.6.18内核和打补丁后的内核的结果

	2.6.18			打补丁之后		
	Mbps	sec	%TX	Mbps	sec	%TX
e1000e（eth3）	941.38	9.12	14.6	941.41	9.12	14.1
VLAN（eth3.510）	938.57	9.15	22.0	938.98	9.15	14.0

吞吐量没什么变化，但 CPU 使用率从 22%降低到 14%，与普通网络基本相同。

没打补丁的 2.6.18 内核也能产生相当的吞吐量的原因是，测试时只运行了 nuttcp，CPU 还有富余。

这样，网络设备的校验和计算功能也能在 VLAN 设备中使用了，降低了 CPU 的负载。

总结

本 hack 是降低 CPU 使用率的例子。即使仅通过吞吐量看不出问题，也能像这样通过 oprofile 找出问题所在。

参考文献

- Phil Dykstra's nuttcp quick start guide
  http://www.wcisd.hpc.mil/nuttcp/Nuttcp-HOWTO.html
- Linux Man Page TCP(7)
- ギガビット PCI ベースのネットワーク・コネクション（Linux*）用ネットワーク・アダプター・ドライバー（千兆位 PCI 网卡驱动程序 Linux 版）
  http://downloadcenter.intel.com/Detail_Desc.aspx?strState=LIVE&ProductID=2776&DwnldID=16563&lang=jpn

——大岩尚宏

第6章

高手们的调试技术

hack #43~#66

265

本章集合了调试时用到的各种各样的工具，以及一些经验技巧。介绍的工具和技术包括 strace、objdump、Valgrind、kprobes、jprobes、KAHO、systemtap、proc 文件系统、oprofile、VMware vprobe、错误注入（fault injection）、Xen 等。此外，还介绍了 OOM Killer 的行为和原理、通过 GOT/PLT 进行函数调用的原理和理解方法、initramfs 的调试方法、使用 RT Watchdog 检测实时进程停止响应的方法，以及调查手头的 x86 机器是否支持 64bit 的方法等。

使用 strace 寻找故障原因的线索

本 Hack 讲解发生故障时，使用 strace 跟踪系统调用，以寻找故障原因的线索的方法。

strace

strace 能够跟踪进程使用的系统调用，并显示其内容。因此，调试原因不明的故障时，首先使用 strace 找出系统调用中出错的地方，通常能得到故障发生的线索。特别是与文件有关的错误、参数错误等，通过此方法可以较简单地发现问题所在。

该方法能有效地发现系统调用失败有关的故障，但无法发现用户写出的程序或共享库中发生的错误。

strace 的使用范例

找不到要访问的文件或无权限访问文件等情况下，系统调用通常会原样返回错误内容。下面用程序 st1.c 确认一下实际情况。

266

```
[st1.c]
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fp;
   fp = fopen("/etc/shadow", "r");
   if (fp == NULL) {
      printf("Error!\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

执行该程序后显示如下。

```
$ ./st1
Error!
```

原因是程序试图打开一般用户没有读取权限的/etc/shadow文件，但是通过错误消息却无法得知这一点。真实的程序也会有错误消息内容不明确、所有地方都显示同样的错误信息的情况，甚至可能什么都不显示。上面就是这种程序的一个例子。这种情况下，很难确定错误发生在源代码的什么地方，因此也无法用 GDB 设置断点。这时可以像下面这样执行 strace。

```
$ strace st1
execve("./st1", ["st1"], [/* 44 vars */]) = 0
brk(0)                                  = 0x486b000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaaab000
uname({sys="Linux", node="cglsv1.cgl", ...}) = 0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/usr/local/X11R7/lib/tls/x86_64/libc.so.6", O_RDONLY) = -1 ENOENT (No such file
or directory)
...
```

267

```
(省略)
...
open("/lib64/libc.so.6", O_RDONLY)      = 3 ---------- ①
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\331"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1687632, ...}) = 0
```

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaad7000
mmap(0x359de00000, 3461272, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x359de00000
mprotect(0x359df44000, 2097152, PROT_NONE) = 0
mmap(0x359e144000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x144000) = 0x359e144000
mmap(0x359e149000, 16536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x359e149000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaad8000
arch_prctl(ARCH_SET_FS, 0x2aaaaaad8210)    = 0
mprotect(0x359e144000, 16384, PROT_READ)   = 0
mprotect(0x359dc19000, 4096, PROT_READ)    = 0
munmap(0x2aaaaaaac000, 173557)             = 0
brk(0)                                  = 0x486b000
brk(0x488c000)                          = 0x488c000
open("/etc/shadow", O_RDONLY)           = -1 EACCES (Permission denied) ————②
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x2aaaaaaac000
write(1, "Error!\n", 7)                 = 7 ————————————————————③
exit_group(1)                           = ?
Process 18399 detached
```

出错并结束的话，从后往前看 strace 的输出结果是解决问题的捷径。上述例子中，最后的③即为在界面上显示错误消息的系统调用。再往前看，可知②处系统调用 open()失败，而且立即可以得知程序在试图打开/etc/shadow 时发生了 Permission denied 错误。

显示的信息有很多，但开头的信息都是关于启动进程时的处理。尽管这一部分有很多错误，但这些错误是进程在试图从各种路径中加载共享库而导致的。从①开始的十几行可以看出，stl 成功地将所用的库连接到了进程。附近都是运行时加载器（runtime loader）的处理，可以忽略。

使用 GDB 详细调查

268

刚才的例子仅通过 strace 的输出结果就能完全理解问题的情况，但有些程序引发错误的真正原因可能在其他位置。这种情况下，通常会使用 GDB 进行进一步的调查，而实现这一步就需要在某些地址上设置断点。给 strace 添加-i 选项即可显示程序在哪个地址进行了系统调用，可以将该地址作为断点使用。

```
$ strace -i st1
...
[    359debf310] open("/etc/shadow", O_RDONLY) = -1 EACCES (Permission denied)
...
```

各行开头[]中的数字就是执行系统调用的代码的地址。在 GDB 中试试看指定该地址并显示 backtrace。

```
$ gdb st1
...
(gdb) start
Breakpoint 1 at 0x4004c0: file st1.c, line 7.
Starting program: /home/kyamato/DebugHacks/kyamato/chapter5/strace.work/st1g
main () at st1.c:7
7       fp = fopen("/etc/shadow", "r");
(gdb) b *0x359debf310 ——————————————————————————————strace 显示的地址
Breakpoint 2 at 0x359debf310
(gdb) c
Continuing.

Breakpoint 2, 0x000000359debf310 in __open_nocancel () from /lib64/libc.so.6
(gdb) bt
#0  0x000000359debf310 in __open_nocancel () from /lib64/libc.so.6
#1  0x000000359de68f23 in __GI__IO_file_open () from /lib64/libc.so.6
#2  0x000000359de6906c in _IO_new_file_fopen () from /lib64/libc.so.6
#3  0x000000359de5eba4 in __fopen_internal () from /lib64/libc.so.6
#4  0x00000000004004cf in main () at st1.c:7
gdb st1
```

attach 到进程上

刚才用 strace 启动进程并检查了它的行为，接下来以下面的程序为例讲解一下如何用 strace 查看运行中的进程（如守护进程）的行为。

269

```
[st2.c]
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
```

```
    while(1) {
        FILE *fp;
        fp = fopen("/etc/shadow", "r");
        if (fp == NULL)
            printf("Error!\n");
        else
            close(fp);
        sleep(3);
    }
    return EXIT_SUCCESS;
}
```

首先执行上面这个 st2 程序。该程序不会结束，会一直运行下去。跟踪该程序的系统调用要用到-p <PID>选项，其结果如下。这样就能跟踪 st2 的系统调用，而且能显示④、⑤两处出现的错误。按 Ctrl-C 键即可结束程序。

```
$ strace -p `pidof st2`
Process 23030 attached - interrupt to quit
restart_syscall(<... resuming interrupted call ...>) = 0
open("/etc/shadow", O_RDONLY)      = -1 EACCES (Permission denied) ————————④
write(1, "Error!\n", 7)            = 7
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({3, 0}, {3, 0})          = 0
open("/etc/shadow", O_RDONLY)      = -1 EACCES (Permission denied) ————————⑤
write(1, "Error!\n", 7)            = 7
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({3, 0}, <unfinished ...>
```

270

其他方便的用法

如下加上-o 选项可以将显示内容输出到文件中。

```
$ strace -o output.log command
```

strace 的输出为标准错误输出，因此可以像下面这样将显示内容输出到标准输出上，再传给 grep、less 等（以 bash 为例）。

```
$ strace command 2>&1 | grep map
$ strace command 2>&1 | less
```

进程执行 fork()之后，要跟踪 fork()之后的进程，可以使用-f 选项。

```
$ strace -f command
```

系统调用的执行时刻可以用-t 或-tt 选项显示。两者的不同点是，-t 以秒为单位，-tt 以毫秒为单位。

```
$ strace -t command
$ strace -tt command
```

总结

本 hack 讲述了利用跟踪系统调用的 strace 命令在故障发生时寻找线索的方法。

——大和一洋

objdump 的方便选项

本 hack 介绍在利用 objdump 处理带有调试信息的二进制文件时，能让工作事半功倍的选项。

objdump 的选项-S、-l 十分方便。如果二进制文件中带有调试信息，可以像下面这样将源代码、文件名及行数与汇编代码对应显示。下例中用到了**“HACK#13 怎样学习汇编语言”**中用到的测试程序。

```
$ gcc -Wall -O0 -g assemble.c -o assemble
$ objdump -Sl --no-show-raw-insn assemble
...
080483cc <main>:
...
/home/user/assemble.c:18
     unsigned int i = 0;
 80483dd:    movl   $0x0,-0x10(%ebp)
...
/home/user/assemble.c:24

     i = 0xabcd;
 804840a:    movl   $0xabcd,-0x10(%ebp)
```

271

```
/home/user/assemble.c:26

     if (i != 0x1234)
 8048411:     cmpl   $0x1234,-0x10(%ebp)
 8048418:     je     8048427 <main+0x5b>
/home/user/assemble.c:27
          i = 0;
 804841a:     movl   $0x0,-0x10(%ebp)
/home/user/assemble.c:29

     while (i == 0)
 8048421:     jmp    8048427 <main+0x5b>
/home/user/assemble.c:30
          i++;
 8048423:     addl   $0x1,-0x10(%ebp)
/home/user/assemble.c:29
     i = 0xabcd;

     if (i != 0x1234)
          i = 0;

     while (i == 0)
 8048427:     cmpl   $0x0,-0x10(%ebp)
 804842b:     je     8048423 <main+0x57>
/home/user/assemble.c:32
          i++;

     func();
 804842d:     call   804839e <func>
/home/user/assemble.c:33
     i = func_pointer();
 8048432:     mov    -0x8(%ebp),%eax
 8048435:     call   *%eax
 8048437:     mov    %eax,-0x10(%ebp)
/home/user/assemble.c:35

     for (i=0; i<MAX_WORD-1 ; i++)
 804843a:     movl   $0x0,-0x10(%ebp)
 8048441:     jmp    8048452 <main+0x86>
```

272

```
/home/user/assemble.c:36
            word = words[i];
 8048443:     mov   -0x10(%ebp),%eax
 8048446:     movzbl -0x20(%ebp,%eax,1),%eax
 804844b:     mov   %al,-0x9(%ebp)
/home/user/assemble.c:35
            i++;

      func();
      i = func_pointer();

      for (i=0; i<MAX_WORD-1 ; i++)
 804844e:     addl  $0x1,-0x10(%ebp)
 8048452:     cmpl  $0xe,-0x10(%ebp)
 8048456:     jbe   8048443 <main+0x77>
/home/user/assemble.c:38
            word = words[i];

      return 0;
 8048458:     mov   $0x0,%eax
/home/user/assemble.c:39
}
 804845d:     add   $0x24,%esp
 8048460:     pop   %ecx
 8048461:     pop   %ebp
 8048462:     lea   -0x4(%ecx),%esp
 8048465:     ret
...
```

这种显示方式需要使用调试信息，因此用 GCC 编译时必须加上-g 选项。而且，汇编语言和源代码总会发生一些偏差，如上文中的 while 语句出现的两个地方。虽然不是完全正确，但十分方便。

273

总结

本 hack 介绍了使用 objdump 时的一些让工作事半功倍的选项，但并不是靠这些选项就可以洞察一切，特别是对优化过的二进制文件，结果更是不着边际，只能作为参考。还有个 addr2line 命令，能输出与 objdump 的-l 选项相同的信息。

Valgrind 的使用方法（基本篇）

本 hack 介绍程序动态分析工具 Valgrind 的基本使用方法。

Valgrind 是什么

Valgrind 能检测出内存的非法使用，对缓存、堆进行评测（profile），检测 POSIX 线程的冲突等。Valgrind 的特征之一就是检测对象程序在编译时无须指定特别的选项，也不需要连接特别的函数库。本 hack 介绍 Valgrind 最为典型的用途——内存非法使用的检测方法。基本的使用方法如下所示。这里 program 为要检查的程序的文件名。

```
$ valgrind --tool=memcheck --leak-check=yes program
```

此外，检测内存非法使用（memcheck）为默认启用的选项，因此也可以只输入下面的命令。

```
$ valgrind --leak-check=yes program
```

检测内存泄漏

下面利用程序 test1 说明一下内存泄漏的检测方法，源代码为 test1.c。test1 在 malloc()之后没有执行 free()，是典型的内存泄漏。另外，为了在 Valgrind 中显示源代码行号等，编译时添加了-g 选项。

[test1.c]

```
int main(void)
{
   char *p = malloc(10);     /* 内存分配 */
   return EXIT_SUCCESS;      /* 不释放内存就结束 */
}
```

274

运行结果如下所示。

```
$ valgrind test1
...
==3125== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==3125== malloc/free: in use at exit: 10 bytes in 1 blocks.
==3125== malloc/free: 1 allocs, 0 frees, 10 bytes allocated. ———————— (A)
```

```
==3125== For counts of detected errors, rerun with: -v
==3125== searching for pointers to 1 not-freed blocks.
==3125== checked 65,416 bytes.
==3125==
==3125== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1 ——— (B)
==3125==    at 0x4A05809: malloc (vg_replace_malloc.c:149)
==3125==    by 0x400489: main (test1.c:6)
==3125==
==3125== LEAK SUMMARY: ———————————————————————————————— (C)
==3125==    definitely lost: 10 bytes in 1 blocks.
==3125==      possibly lost: 0 bytes in 0 blocks.
==3125==    still reachable: 0 bytes in 0 blocks.
==3125==         suppressed: 0 bytes in 0 blocks.
==3125== Reachable blocks (those to which a pointer was found) are not shown.
==3125== To see them, rerun with: --show-reachable=yes
```

上面输出的==3125==中的数字部分 3125，是运行的 Valgrind 的进程 ID，因此每次运行都会显示不同的值。所有行都有这个数字，因此十分醒目。这个信息并不是很重要。

Valgrind 的输出中，首先要注意的就是 malloc/free 的次数(A)。malloc 执行了 1 次，而 free 运行了 0 次。LEAK SUMMARY(C)中也显示了 definitely lost。更详细的信息显示在(B)中，据此可知地址 0x400489（test1.c 的第 6 行）执行的 malloc() 分配的内存泄漏了。对比源代码，可知它正确地检测出了内存泄漏。

275

检测对非法内存地址的访问

接下来介绍访问未分配地址的 bug 的检测方法，比如下面的 test2.c。

[test2.c]

```
int main(void)
{
    char *p = malloc(10);       /* 分配10字节 */
    p[10] = 1;                  /* 向已分配的地址(10字节)外写入 */
    free(p);
    return EXIT_SUCCESS;
}
```

用 Valgrind 检查该程序，输出结果如下，可知程序对分配的内存区域之外的地址执行了写入操作。

```
...
=3438= Invalid write of size 1
=3438=    at 0x4004D6: main (test2.c:7)
=3438= Address 0x4C3603A is 0 bytes after a block of size 10 alloc'd
=3438=    at 0x4A05809: malloc (vg_replace_malloc.c:149)
=3438=    by 0x4004C9: main (test2.c:6)
...
```

读取未初始化区域

还可以检测出对未初始化区域的读取操作，这里用 test3.c 为例测试一下。

[test3.c]

```
int main()
{
   int *x = malloc(sizeof(int));     /* 分配内存大小为 int 的大小 */
   int a = *x + 1;                   /* 不初始化就使用分配的内存 */
   free(x);
   return a;
}
```

编译上述代码后用 Valgrind 检查，能得到下述结果。

276

```
...
=3941= Syscall param exit_group(exit_code) contains uninitialised byte(s)
=3941=    at 0x359DE948CF: _Exit (in /lib64/libc-2.5.so)
=3941=    by 0x359DE32D04: exit (in /lib64/libc-2.5.so)
=3941=    by 0x359DE1D8AA: (below main) (in /lib64/libc-2.5.so)
...
```

访问已释放的区域

Valgrind 也能检测出对已释放区域的内存访问，以 test4.c 为例测试一下。

[test4.c]

```
int main(void)
{
   int *x = malloc(sizeof(int));     /* 分配内存 */
   free(x);                          /* 释放内存 */
   int a = *x + 1;                   /* 访问已释放的内存区域 */
```

```
    return a;
}
```

下面指出了访问所在的位置。

```
...
==4134== Invalid read of size 1
==4134==    at 0x4004DB: main (test4.c:8)
==4134== Address 0x4C36030 is 0 bytes inside a block of size 4 free'd
==4134==    at 0x4A0541E: free (vg_replace_malloc.c:233)
==4134==    by 0x4004D6: main (test4.c:7)
...
```

内存双重释放

如 test5.c，Valgrind 还能检测出内存的双重释放。

```
[test5.c]
int main(void)
{
    char *x = malloc(sizeof(int));    /* 分配内存 */
    free(x);                          /* 释放内存 */
    free(x);                          /* 双重释放同一地址 */
    return EXIT_SUCCESS;
}
```

277

编译上述代码并用 Valgrind 检查，就能指出问题。

```
...
==4236== Invalid free() / delete / delete[]
==4236==    at 0x4A0541E: free (vg_replace_malloc.c:233)
==4236==    by 0x4004DF: main (test5.c:8)
==4236== Address 0x4C36030 is 0 bytes inside a block of size 4 free'd
==4236==    at 0x4A0541E: free (vg_replace_malloc.c:233)
==4236==    by 0x4004D6: main (test5.c:7)
...
```

非法栈操作

下面的 test6.c 向比栈指针更低的地址写入数据。

[test6.c]

```
int main(void)
{
        int a;          /* 在栈上分配变量（内存） */
        int* p = &a;    /* 创建指向 a 的地址的指针 */
        p -= 0x20;      /* 错误的指针操作 */
        *p = 1;         /* 向错误的内存地址写入 */
        return EXIT_SUCCESS;
}
```

利用 GCC 编译后用 Valgrind 检查，输出结果如下。但是，具体行为依赖于编译器，因此不同环境下输出内容也可能不尽相同。

```
...
==5928==
==5928== Invalid write of size 4
==5928==    at 0x40043D: main (test6.c:9)
==5928== Address 0x7FF000024 is just below the stack ptr. To suppress, use:
--workaround-gcc296-bugs=yes
...
```

test6 中发生的问题如下。首先在栈上创建变量 a。main 函数中只有 a 和 p 两个变量，因此栈的下限（栈指针指向的值）就是 a 的地址本身，或是比它小几个字节到几十个字节的地址，因此，p -= 0x20 得到的地址（从 a 的地址偏移 0x80 个字节，即小 0x20 * sizeof(int)个字节的地址）绝大多数情况下会小于栈指针。向该地址写入 1，Valgrind 就会检测出该行为并输出信息。

278

无法检测的错误

Valgrind 在检测非法使用内存方面效果非凡，但并不是万能的。例如，下面这种对栈上生成的内存区域的非法访问就无法检测到。

```
test_a.c
int main(void)
{
   char p[10];
   p[100] = 1;
   return EXIT_SUCCESS;
}
```

编译上述代码并检查，如下并没有显示出任何错误。

```
...
==6284== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==6284== malloc/free: in use at exit: 0 bytes in 0 blocks.
==6284== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==6284== For counts of detected errors, rerun with: -v
==6284== All heap blocks were freed -- no leaks are possible.
```

总结

本 hack 介绍了利用 Valgrind 检测内存泄漏、访问非法内存地址、读取未初始化内存区域、访问已释放内存区域、内存双重释放、非法栈空间操作的方法。

——大和一洋

279

Valgrind 的使用方法（实践篇）

本 hack 介绍使用 Valgrind 检测出难以发现的内存泄漏的实例。

检测内存泄漏

内存泄漏的危险之一就是，即使发生，很多情况下也不会立即引发什么问题。即使如此，内存泄漏大到一定程度后，利用 top 等命令长时间监视内存使用量，也可以发现内存泄漏的征兆。例如，现在 foo 程序内存泄漏的话，进程的虚拟内存使用量（VIRT 和 RES）就会随着时间增长（如下所示）。VIRT 和 RES 的区别是，VIRT 中包含了被交换（swap）出的空间，而 RES 不包含。

```
[刚开始]
PID USER PR NI  VIRT  RES SHR S %CPU %MEM  TIME+  COMMAND
2512 root 25  0 12268 6880 284 R  100 1.8  0:18.66 foo

[10 分钟之后]
PID USER PR NI  VIRT  RES SHR S %CPU %MEM  TIME+  COMMAND
2512 root 25  0 31084  23m 284 R  100 6.2  1:08.65 foo

[40 分钟之后]
PID USER PR NI  VIRT  RES SHR S %CPU %MEM  TIME+  COMMAND
```

```
2512 root 25  0  121m 114m 284 R  100 30.9 5:38.63 foo
```

在 VIRT 或 RES 逐渐增加的情况下，有时并不是内存泄漏。比如库函数中分配内存时，为了提高效率而多分配内存，或是内存不再需要时也不释放而是继续当做缓存使用，保持内存占用。例如，用户程序中 malloc()几个字节时，接受请求的 glibc 在内部会分配更多的内存，之后执行 free()也不会立即将内存区域还给内核。但是，这种多余分配的内存的大小虽然各个函数库和版本不尽相同，但最多只有几 MB 而已。

很难发现的内存泄漏

相反，如果内存泄漏量很小，用前面说的方法就很难检测。但是，如果是几个月、几年长时间执行的程序，内存资源也会被慢慢消耗，某天就会发生 OOM Killer（参见 "HACK#56 OOM Killer 的行为和原理"），导致不可预料的后果。 280

像这种长时间运行的程序，建议用 Valgrind 检查一次。下面是个实际的例子，通过该程序发现了内存泄漏，而这段代码就是内存泄漏的相关部分。dlopen()是在运行时加载库的函数，而 dlclose()与 dlopen()相反，是将已加载的库关闭的函数。由于下面只是代码片段，因此打开共享库后就直接关闭了，而实际上，打开后还调用了共享库中的函数，函数不再需要时才关闭。

```
#include <stdio.h>
#include <dlfcn.h>

int main(){
   void *p = NULL;

   while(1){
      p = dlopen("./lib1.so",RTLD_LAZY);
      if (NULL == p){
         printf("Error: dlopen()\n");
         return 1;
      }
      dlclose(p);
      sleep(100);
   }
   return 0;
}
```

单从代码来看，一眼看上去似乎没有问题，实际上这段代码的确没有问题。但是，使用 Valgrind 后就显示了以下信息。

```
$ valgrind --leak-check=full prog
(执行后输入Ctrl-C)
...
==2370== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 5 from 1)
==2370== malloc/free: in use at exit: 8 bytes in 1 blocks.
==2370== malloc/free: 6 allocs, 5 frees, 1,454 bytes allocated.
==2370== For counts of detected errors, rerun with: -v
==2370== searching for pointers to 1 not-freed blocks.
==2370== checked 68,368 bytes.
==2370==
==2370== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2370==    at 0x4A05809: malloc (vg_replace_malloc.c:149)
==2370==    by 0x37D7E067EF: _dl_map_object_from_fd (dl-load.c:1473)
==2370==    by 0x37D7E07CAB: _dl_map_object (dl-load.c:2232)
==2370==    by 0x37D7E1088E: dl_open_worker (dl-open.c:252)
==2370==    by 0x37D7E0CC35: _dl_catch_error (dl-error.c:178)
==2370==    by 0x37D7E1036B: _dl_open (dl-open.c:551)
==2370==    by 0x37D8A00F79: dlopen_doit (dlopen.c:66)
==2370==    by 0x37D7E0CC35: _dl_catch_error (dl-error.c:178)
==2370==    by 0x37D8A014EC: _dlerror_run (dlerror.c:164)
==2370==    by 0x37D8A00EF0: dlopen@@GLIBC_2.2.5 (dlopen.c:87)
==2370==    by 0x4005C6: main (main.c:8)  ——————————————————①
==2370==
==2370== LEAK SUMMARY:
==2370==    definitely lost: 8 bytes in 1 blocks.
==2370==      possibly lost: 0 bytes in 0 blocks.
==2370==    still reachable: 0 bytes in 0 blocks.
==2370==         suppressed: 0 bytes in 0 blocks.
==2370== Reachable blocks (those to which a pointer was found) are not shown.
==2370== To see them, rerun with: --show-reachable=yes
```

281

此外，该程序像守护程序一样，是个不会结束的程序，因此我们在执行后不久按 Ctrl-C 键将其强制结束。输出内容的详细阅读方法请参见 **“HACK#45 Valgrind 的使用方法（基本篇）”**。这里①处报告 main.c 的第 8 行 dlopen()被调用，其中调用的 malloc()分配的内存发生了泄漏。实际上源代码 dl-load.c:1473（这行代码实

际上换了行，应该是 1474 行）中，malloc()是这样使用的。另外，没找到释放这里分配的内存的代码。

```
1471  /* Create an appropriate searchlist. It contains only this map.
1472     This is the definition of DT_SYMBOLIC in SysVr4. */
1473  l->l_symbolic_searchlist.r_list =
1474   (struct link_map **) malloc (sizeof (struct link_map *));
```

该问题的修正方法参见参考文献。

282

dlopen()是 glibc 提供的函数，因此本例并不是用户应用程序的 bug，而是操作系统函数库的 bug。像这种函数库出问题的情况，当然对用户的代码审查多少遍也不可能发现。此外，dlopen()每次调用都会泄漏几个字节的内存，因此即使用本 hack 前半部分介绍的 top 命令监视内存使用量，几小时之内也看不出有显著的增加。这种状况可以通过 Valgrind 发现问题。

参考文献

- glibc/efl/dl-load.c 的 CVS
  http://sourceware.org/cgi-bin/cvsweb.cgi/libc/elf/dl-load.c.diff?r1=1.249.2.31&r2=1.290&cvsroot=glibc&f=h

总结

本 hack 介绍了利用 Valgrind 检测到操作系统提供的函数库中的问题——难以发现的内存泄漏的方法。

——大和一洋

利用 kprobes 获取内核内部信息

本 hack 介绍利用内核调试功能之一——kprobes，动态插入侦测器（probe）获取内核内部信息的方法。

kprobes

用 printk 显示变量等方法，是有效的内核调试方法之一，但是这种方法必须重新

构建并用新内核启动，调试效率很低。以内核模块的方式使用 kprobes，就可以在任意地址插入侦测器（probe），执行包括 printk 在内的各种调试工作，而无须重新构建内核，也无须重启。

简单的例子

下面这个例子在 do_execve()函数开头进行侦测。要注册侦测器，首先要分配一个 kprobe 结构的变量供 kprobes 运行时使用。接下来，在 kprobe 结构的 addr 成员中设置要插入侦测器的地址，有很多方法可以获取该地址，其中一种就是使用
283
kallsyms_lookup_name()。但是，内核版本 2.6.19 以后就不再导出（EXPORT）该函数的符号了，模块也就无法再使用它。内核版本 2.6.19 以上版本可以在 struct kprobes 的 symbol_name 成员中设置要侦测的函数的符号名称(参见下面的源代码)。设置了 symbol_name 的话，addr 成员就无须再设置。

RHEL5 等某些发行版中，即使是 2.6.18 之前的内核也可能可以使用 symbol_name。

另外，如下使用/proc/kallsyms 也可以求出直接地址。

```
# cat /proc/kallsyms | grep "\ doexecve$"
ffffffff8003e1b4 T do_execve
```

接下来在 pre_handler 成员中设置侦测器函数，并调用 register_kprobe()函数，这样在执行指定的地址时（严格来说是在执行之前），会先调用侦测器函数。此外，用 unregister_kprobe()函数可以撤销侦测器。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

struct kprobe kp; /* 分配kprobe结构的变量 */

int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
   printk(KERN_INFO "pt_regs: %p, pid: %d, jiffies: %ld\n",
                regs, current->tgid, jiffies);
   return 0;
}
```

```
static __init int init_kprobe_sample(void)
{
    /* 设置do_execve()地址 */
    kp.addr = (kprobe_opcode_t *) 0xffffffff8003e1b4;

    /* 如果不想直接给addr成员设置地址，也可以使用符号名 */
    /* kernel version在2.6.18以下时 */
    /* kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("do_execve"); */
    /* kernel version在2.6.19以上时 */
    /* kp.symbol_name = "do_execve"; */

    /* 在设置地址的命令执行之前，设置侦测器 */
    kp.pre_handler = handler_pre;

    /* 注册侦测器 */
    register_kprobe(&kp);

    return 0;
}

module_init(init_kprobe_sample);

static __exit void cleanup_kprobe_sample(void)
{
    /* 撤销侦测器注册 */
    unregister_kprobe(&kp);
}
module_exit(cleanup_kprobe_sample);

MODULE_LICENSE("GPL");
```

284

编译上述示例并 insmod，那么在 do_execve()执行之前就会输出下面的内核信息。由于该函数用于生成进程，因此执行 ls 等命令时应该会显示一次。

```
pt_regs: ffffffff80414f58,     pid: 6899, jiffies:   4405656851
pt_regs: ffff810009189f58,     pid: 6902, jiffies:   4405656857
pt_regs: ffffffff80414f58,     pid: 6903, jiffies:   4405656864
...
```

调查参数

用上例的方法可以在指定函数执行时方便地显示全局变量 current、jiffies 等，但是实际调试中经常需要调查函数使用的变量的值。要在 kprobes 的侦测器内显示某个函数的局部变量的值，需要一些技巧，原因是在 printk 的参数中无法直接指定变量名，因此必须给侦测器函数提供一个 pt_regs 结构，其中保存了指定地址的命令执行时的寄存器信息。当然，不同架构下该结构的成员变量不尽相同，但用该结构可以显示变量等更为详细的信息。

285

```
struct pt_regs {
   unsigned long r15;
   unsigned long r14;
   unsigned long r13;
   unsigned long r12;
   unsigned long rbp;
   unsigned long rbx;
   unsigned long r11;
   unsigned long r10;
   unsigned long r9;
   unsigned long r8;
   unsigned long rax;
   unsigned long rcx;
   unsigned long rdx;
   unsigned long rsi;
   unsigned long rdi;
   unsigned long orig_rax;
   unsigned long rip;
   unsigned long cs;
   unsigned long eflags;
   unsigned long rsp;
   unsigned long ss;
};
```

首先显示一下参数信息。do_execve()接收如下参数。

```
int do_execve(char * filename, char __user *__user *argv,
            char __user *__user *envp, struct pt_regs * regs)
```

例如，显示 filename 和 argv 的侦测器的代码如下所示。

```
int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
   int cnt = 0;
   char __user *__user *argv;

   printk(KERN_INFO "filename: %s\n", (char*)regs->rdi);
   for (argv = (char __user *__user *)regs->rsi; *argv != NULL; argv++, cnt++)
      printk(KERN_INFO "argv[%d]: %s\n", cnt, *argv);
     return 0;
}
```

286

在 x86_64 架构中，函数的参数从左到右分别保存在 rdi、rsi、rdx、rcx、r8、r9 中，因此查看 rdi 和 rsi 就能得到第 1、第 2 个参数的值（参见 **“HACK#10 函数调用时的参数传递方法（x86_64 篇）”**、**“HACK#11 函数调用时的参数传递方法（i386 篇）”**。

如果只想查看参数的值，那么用 **“HACK#48 使用 jprobes 查看内核内部的信息”** 讲述的方法使用 jprobes 更简单。

显示栈跟踪

使用 kprobes 的另一个有效的调试方法，就是显示栈跟踪。

```
int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
   dump_stack();
   return 0;
}
```

插入上面的侦测器后，就会显示下面的内核信息。

```
Call Trace:
 <#DB> [<ffffffff8003e1b4>] do_execve+0x0/0x243
 [<ffffffff883e3009>] :kpro3:handler_pre+0x9/0x10
 [<ffffffff8006512c>] kprobe_handler+0x198/0x1c8
 [<ffffffff80065197>] kprobe_exceptions_notify+0x3b/0x75
 [<ffffffff80065def>] notifier_call_chain+0x20/0x32
 [<ffffffff800649b4>] do_int3+0x42/0x83
 [<ffffffff8006413b>] int3+0x93/0xa4
```

```
[<ffffffff8003e1b5>] do_execve+0x1/0x243
<<EOE>> [<ffffffff80052a64>] sys_execve+0x36/0x4c
[<ffffffff8005c4d3>] stub_execve+0x67/0xb0
```

参考文献

内核源代码树/Documentation/kprobes.txt

总结

本 hack 介绍了使用 kprobes 显示内核的全局变量、参数及栈跟踪的方法。

——大和一洋

使用 jprobes 查看内核内部的信息

本 hack 介绍了使用内核调试的功能之一——jprobes，将侦测器插入到内核函数的开头，获取内核内部信息的方法。

jprobes

“HACK#47 利用 kprobes 获取内核内部信息”介绍了 kprobes 的使用方法。kprobes 几乎可以在内核中的任意地点插入侦测器，相反，jprobes 则是特别为函数开头的侦测器准备的。因此，与 kprobes 相比，jprobes 能更容易地获取传给函数的参数。

简单的例子

这里给出的例子仍是插入到 do_execve()函数开头的侦测器，与[HACK#47]介绍 krpobes 时使用的例子相同。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

struct jprobe jp;

int jp_do_execve(char *filename, char __user *__user *argv,
```

```
            char __user *__user *envp, struct pt_regs *regs)
{
   int cnt = 0;

   printk(KERN_INFO "filename: %s\n", filename);
   for (; *argv != NULL; argv++, cnt++)
      printk(KERN_INFO "argv[%d]: %s\n", cnt, *argv);

   jprobe_return();
   return 0;
}

static __init int init_jprobe_sample(void)
{
   jp.kp.symbol_name = "do_execve";
   jp.entry = JPROBE_ENTRY(jp_do_execve);
   register_jprobe(&jp);

   return 0;
}
module_init(init_jprobe_sample);

static __exit void cleanup_jprobe_sample(void)
{
   unregister_jprobe(&jp);
}
module_exit(cleanup_jprobe_sample);

MODULE_LICENSE("GPL");
```

288

代码大体上与 kprobes 的情况完全相同，只有 3 点不一样。第 1 点就是给侦测器预备的数据结构是 jprobe 结构，并将指针传递给 register_jprobe()和 unregister_jprobe()。jprobe 结构的成员如下所示，只包括 kprobe 结构和 entry 两者。

```
struct jprobe {
   struct kprobe kp;
   kprobe_opcode_t *entry; /* probe handling code to jump to */
};
```

struct kprobe kp 成员设置的是要侦测的函数（上例中为 do_execve()）的符号

（symbol_name）或地址（addr）。entry 中为 JPROBE_ENTRY()宏处理过的侦测器处理程序（上例中为 jp_do_execve()）。

第 2 点不同就是侦测器处理程序的参数应当与要侦测的函数（上例中为 do_execve()）的参数相同。这样 printk()等函数在获取变量内容时，就能原封不动地使用变量名，编写侦测器处理程序就会更方便。使用 kprobes 时，必须通过寄存器或栈才能计算出参数的值。此外，计算方法还依赖于架构。如果使用 jprobes，那么无须了解架构的详细知识，也能简单地查看参数的值。

第 3 点不同就是，侦测器处理程序的末尾不仅要用 return 语句，还要写上 jprobe_return()。该函数会回到被侦测的函数，后面的那个 return 实际上不会被执行。这个 return 语句是为了避免编译器的警告和错误而写的。

总结

本 hack 介绍了使用 jprobes 在函数的开头放置侦测器的方法。用 jprobes 获取参数比 kprobes 更简单。

289

——大和一洋

使用 kprobes 获取内核内部任意位置的信息

本 hack 介绍利用 kprobes 在内核函数的任意位置插入侦测器并获取信息的方法。

kprobes 的强大功能

[HACK#47]介绍了利用 kprobes 在函数开头插入侦测器以显示内核内的全局变量、函数参数和栈跟踪的例子。但是，如果只在函数开头插入侦测器，那么大多数情况下使用[HACK#48]介绍的 jprobes 更方便。但是，kprobes 拥有 jprobes 没有的强大功能，那就是它能在内核的任意地址插入侦测器。此外，侦测器可以在任意地址的指令执行之前或之后执行，或者前后都执行。

向任意地址插入侦测器

内核大部分用 C 语言写成，但遗憾的是，kprobes 并不能向源代码内的任意行插入侦测器，可以插入的位置只能是任意地址。因此，应当观察汇编代码，找到源代码中想要调查的位置对应于编译后的二进制文件中的什么地址，并调查希望显示的变量保存在哪个寄存器、哪个内存地址。

下面的例子演示了在 do_execve()的①处插入侦测器，并显示 kzalloc()的返回值 bprm。

```
int do_execve(char * filename,
      char __user *__user *argv,
      char __user *__user *envp,
      struct pt_regs * regs)
{
      struct linux_binprm *bprm;
      struct file *file;
      int retval;
      int i;

      retval = -ENOMEM; ——————————————————— ⓪
      bprm = kzalloc(sizeof(*bprm), GFP_KERNEL); ——————— ①
      if (!bprm) ——————————————————————— ②
            goto out_ret;

      file = open_exec(filename);
      (以下省略)
```

290

首先调查一下①的地址。这里所谓的地址并不是在被调查的机器上的绝对逻辑地址，而是相对于 do_execve()开头的相对地址。下面是用 crash 命令调查地址的例子。

在无法使用 crash 命令的环境中，可以用 objdump 反汇编 vmlinux 文件或模块（.ko）来找出地址。*

```
crash> dis do_execve
0xffffffff8003e290 <do_execve>:       push  %r15
0xffffffff8003e292 <do_execve+2>:     mov   %rsi,%r15
0xffffffff8003e295 <do_execve+5>:     mov   $0xd0,%esi
```

```
0xffffffff8003e29a <do_execve+10>:    push  %r14
0xffffffff8003e29c <do_execve+12>:    mov   %rdx,%r14
0xffffffff8003e29f <do_execve+15>:    push  %r13
0xffffffff8003e2a1 <do_execve+17>:    mov   %rdi,%r13
0xffffffff8003e2a4 <do_execve+20>:    push  %r12
0xffffffff8003e2a6 <do_execve+22>:    mov   $0xfffffff4,%r12d
0xffffffff8003e2ac <do_execve+28>:    push  %rbp
0xffffffff8003e2ad <do_execve+29>:    push  %rbx
0xffffffff8003e2ae <do_execve+30>:    sub   $0x8,%rsp
0xffffffff8003e2b2 <do_execve+34>:    mov   2811247(%rip),%rdi      # 0xffff
ffff802ec828 <malloc_sizes+104>
0xffffffff8003e2b9 <do_execve+41>:    mov   %rcx,(%rsp)
0xffffffff8003e2bd <do_execve+45>:    callq 0xffffffff800d2528 <kmem_cache_zalloc>
0xffffffff8003e2c2 <do_execve+50>:    test  %rax,%rax
0xffffffff8003e2c5 <do_execve+53>:    mov   %rax,%rbp
0xffffffff8003e2c8 <do_execve+56>:    je    0xffffffff8003e4c3 <do_execve+563>
0xffffffff8003e2ce <do_execve+62>:    mov   %r13,%rdi
0xffffffff8003e2d1 <do_execve+65>:    callq 0xffffffff8003b769 <open_exec>
...
```

291 源代码中，kzalloc()为内联函数（inline），因此处理内容展开到了 do_execve()函数中。汇编代码和源代码并不是严格的一对一关系，因此正确地对应范围比较困难，但可以估计到，kzalloc 展开的位置大概在 do_execve+28 到 do_execve+45 左右，只要将①前后的源代码与汇编代码比较一下就可以知道。do_execve+2 到 do_execve+17 之间只是将参数保存到了其他寄存器中，似乎是编译器自动生成的函数初始化部分。接下来的 do_execve+20 和 do_execve+22 将-12（$0xfffffff4）赋给了某个寄存器 r12d。由于 ENOMEM 就是 12，似乎对应于源代码上的⓪，而 do_execve+50 的 test 命令和 do_execve+56 的 je 命令是 C 语言 if 语句对应的典型代码，因此可以认为这里对应于②。

因此，似乎可以认为 kzalloc()的返回值（即 bprm 值）就是 do_execve+45 处 kmem_cache_zalloc()的返回值，故侦测器的插入地址就是在调用 kmem_cache_zalloc()的 do_execve+45 之后，而且在保存返回值的 rax 被改变之前，只要符合这两个条件，放在哪里都无所谓。但是，必须放在指令的起始地址上。以上面的反汇编代码为例，必须放在 do_execve+50/+53/+56/+62/+65 等地址。

创建侦测器

侦测 do_execve+50 的代码如下所示。要点在于，在内核版本 2.6.19（或者 RHEL5）以上版本要设置 struct kp 的 symbol_name 成员和 offset 成员，而在低版本上要将 kallsyms_lookup_name()的返回值加上从函数开头算起的偏移量，设置到 addr 成员中。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

struct kprobe kp;

int handler(struct kprobe *p, struct pt_regs *regs) {
   printk(KERN_INFO "rax: %016lx, eflags: %08x, %rip: %016lx\n",
               regs->rax, regs->eflags, regs->rip);
   return 0;
}

static __init int init_kprobe_sample(void)
{
   kp.symbol_name = "do_execve";
   kp.offset = 50;
   /* 如果内核版本在2.6.18以下 */
   /* kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("do_execve") + 50; */

  kp.pre_handler = handler;
  register_kprobe(&kp);

   return 0;
}
module_init(init_kprobe_sample);

static __exit void cleanup_kprobe_sample(void)
{
   unregister_kprobe(&kp);
}
module_exit(cleanup_kprobe_sample);
```

292

```
MODULE_LICENSE("GPL");
```

加载该模块之后，执行某个命令以调用 do_execve()，在笔者的环境中得到了如下的内核信息，其中的 rax 的值就是源代码中 bprm 的值。

```
kernel: rax: ffff81000b9a3800, eflags: 00000246, %rip: ffffffff8003e2c3
```

使用 kprobes 时要注意的是，侦测器处理程序中的 rip 的值并不是侦测器被插入的地址。在笔者的环境中，do_execve()的绝对地址为 ffffffff8003e290。

```
# cat /proc/kallsyms | grep do_execve
ffffffff8003e290 T do_execve
```

插入侦测器的地址为从函数开头起的第 50 字节，因此应该是该地址加上 50（0x32），即 ffffffff8003e2c2。但是 kprobes 处理程序显示的 rip 值比这个地址大 1 字节，这是因为在 x86_64、i386 架构中，调用侦测器时要向被侦测的地址处插入一条 int 3 指令，其长度为 1 字节。侦测器函数的参数 regs 中保存的是这条 int 3 指令执行后的寄存器值，因此程序计数器 rip 的值比侦测器插入地址大一个字节。

293

在指令执行之后插入侦测程序

本 hack 开头说过，kprobes 的强大之处之一，就是能在指令执行之后的位置插入侦测程序，只需如下设置 struct kprobes 的 post_handler 即可。post_handler 成员可以设置与 pre_handler 相同的函数，也可以设置不同的函数。

```
kp.post_handler = handler;
```

将上述代码添加到刚才的示例源代码中并执行，其结果如下。第 1 行为 do_execve+50 的 test 指令执行之前的侦测结果，第 2 行为 test 指令执行之后的侦测结果。

```
kernel: rax: ffff81001e4b9a00, eflags: 00000246, %rip: ffffffff8003e2c3
kernel: rax: ffff81001e4b9a00, eflags: 00000186, %rip: ffffffff88000012
```

test 指令计算两个操作数的逻辑与，并将结果反映到标志寄存器中。该指令不会改变寄存器内容，因此 rax 的值在执行前后是相同的。注意 eflags，现在 rax 和 rax 的逻辑与不是 0，因此 test 指令执行后的零标志（第 6 比特）被清空，而且逻辑与的 MSB（MostSignificant Bit：最高位比特）为 1，可知进位标志（第 7 比特）也被设置了（参见 **“HACK#8 Intel 架构的基本知识”**）。

eflags 中其他发生变化的比特有中断标志（第 9 比特）和陷阱标志（第 8 比特）。这是因为 kprobes 为了在指令执行之后插入侦测器，在禁止中断的状态下使用了 CPU 的 TRAP 功能（每执行一条指令都引发异常）。

此外，指令执行后的 rip 值与侦测对象的地址大相径庭。这是因为，为了调用指令执行前的侦测器，do_execve+50 的 test 指令被 int 3 指令覆盖而破坏，实际上这条 test 指令被保存到内核中的其他位置并执行。rip 显示的地址就是临时保存指令的地址。

总结

本 hack 介绍了将 kprobes 插入任意地址，在内核函数源代码中获取变量值的方法。

——大和一洋

294

使用 kprobes 在内核内部任意位置通过变量名获取信息

本 hack 介绍利用 kprobes 将内核函数整体替换，在任意位置通过变量名获取信息的方法。

函数替换

[HACK#49]介绍了利用 kprobes 在函数的任意位置插入侦测器的方法，但是使用这种方法查看变量的值，就必须分析汇编代码，并查看寄存器或栈。可能的话，大多数情况下我们希望在任意位置通过指定源代码中的变量名来显示变量的值。这里就介绍这种方法。

但是，这种方法需要改变源代码，生成的二进制代码也可能发生大规模的变动。因此，调查的问题可能无法复现。

创建侦测器

下面的侦测器的例子与[HACK#49]相同，都是在 do_execve()的开头显示指针 bprm 的值。首先，把要查看的内核函数（本例中为 do_execve()）整个复制，并改变函数名④，以避免符号名冲突。然后添加 printk()语句⑤以查看变量值。在侦测器处理函数中，⑥处设置 regs->rip，将侦测器结束后的返回地址改成了

my_do_execve()。⑦到⑨为中断运行中的侦测程序的处理，①到③为编译④的函数必需的声明等。这些函数写在了 fs/exec.c 中，因此必须在该文件中声明，否则无法编译。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/binfmts.h>
#include <linux/security.h>
#include <linux/err.h>
#include <linux/kallsyms.h>
#include <linux/acct.h>
#include <linux/file.h>
#include <asm/mmu_context.h>
#include <asm/percpu.h>
```

295

```
#define free_arg_pages(bprm) do { } while (0) ——————————————— ①
int count(char __user * __user * argv, int max); ——————————— ②
int copy_strings(int argc, char __user * __user * argv,
            struct linux_binprm *bprm); ——————————————————— ③

struct kprobe kp;

int my_do_execve(char * filename, char __user *__user *argv, char __user *__user *envp,
      struct pt_regs * regs) ——————————————————————————— ④
{
      struct linux_binprm *bprm;
      struct file *file;
      int retval;
      int i;

      retval = -ENOMEM;
      bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
      if (!bprm)
            goto out_ret;
      printk("bprm: %p, filename: %s\n", bprm, filename); ——————— ⑤

      file = open_exec(filename);
      retval = PTR_ERR(file);
      /* 下面与原来的代码相同 */
```

```
}

int handler(struct kprobe *p, struct pt_regs *regs) {
   regs->rip = (unsigned long)my_do_execve; ——————————⑥
   reset_current_kprobe(); ——————————⑦
   preempt_enable_no_resched(); ——————————⑧
   return 1; ——————————⑨
}

static __init int init_kprobe_sample(void)
{
   kp.symbol_name = "do_execve";
   /* 如果内核版本在2.6.18以下 */
   /* kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("do_execve"); */

  kp.pre_handler = handler;
  register_kprobe(&kp);

   return 0;
}
module_init(init_kprobe_sample);

static __exit void cleanup_kprobe_sample(void)
{
   unregister_kprobe(&kp);
}
module_exit(cleanup_kprobe_sample);

MODULE_LICENSE("GPL");
```

296

安装侦测器的模块及回避问题

上面的代码用内核的标准方法可以编译成模块（这里模块的名字为 kpro3.ko），但是 insmod 时，在笔者的环境下发生了下面的错误。一般说，这个问题经常发生（当然，有些被复制的内核函数不会输出这种错误信息），这是因为模块没有得到使用这些数据的许可。上述源代码中（不是模块中）复制了内核中使用的 do_execve() 函数，因此发生了这种问题。

```
kernel: kpro3: Unknown symbol per_cpu__current_kprobe
kernel: kpro3: Unknown symbol init_new_context
kernel: kpro3: Unknown symbol mm_alloc
kernel: kpro3: Unknown symbol acct_update_integrals
kernel: kpro3: Unknown symbol sched_exec
kernel: kpro3: Unknown symbol __mmdrop
kernel: kpro3: Unknown symbol count
kernel: kpro3: Unknown symbol copy_strings
```

这个限制是Linux的策略，普通方法无法解决。但是，我们只是为了在特定环境下调试，因此用下面的方法给出模块中无法解析的符号的地址，就能回避该问题。

297

```
# cat addrs.dat
SECTIONS
{
 per_cpu__current_kprobe     = 0xffffffff804052a0;
 init_new_context            = 0xffffffff8006d7f7;
 acct_update_integrals       = 0xffffffff8004ecd8;
 sched_exec                  = 0xffffffff800457be;
 __mmdrop                    = 0xffffffff8008d5e2;
 count                       = 0xffffffff80039821;
 copy_strings                = 0xffffffff80017381;
 mm_alloc                    = 0xffffffff8004c098;
}

# ld -r -o kpro3a.ko kpro3.ko -R addrs.dat
```

addrs.dat文件中记载了符号名及其地址，各符号的值可以用下面的方法从/proc/kallsyms或vmlinux文件中获得。

```
# nm vmlinux | grep per_cpu__current_kprobe
ffffffff804052a0 D per_cpu__current_kprobe
# cat /proc/kallsyms | grep init_new_context
ffffffff8006d7f7 T init_new_context
```

如上所示，强行解析地址问题后对模块执行insmod，执行某些命令调用do_execve()后，就得到了下面的内核消息。

```
kernel: bprm: ffff81001ff1a800, filename: /bin/ls
```

总结

本 hack 介绍了使用 kprobes 整个替换内核函数，以通过变量名查看任意位置的变量信息的方法。

——大和一洋

298

使用 KAHO 获取被编译器优化掉的变量的值

本 hack 介绍利用 KAHO 替换进程中的函数，以获取被优化掉导致 GDB 无法检测到的变量的值的方法。

优化和变量显示

本书许多地方都说过，使用 GDB 可以在任意地点暂停进程，并显示当时的寄存器、内存的值。如果被调试进程的可执行文件中包含调试信息（用-g 选项构建的话），那么给出变量名就能显示出变量的值。但是，优化之后有可能无法获取变量的值。例如，用优化选项（-O2）试着编译一下下面这段源代码。

[calc.c]

```
#include <stdio.h>
#include <stdlib.h>

int func(int x)
{
   int a, a0, a1, a2;
   a0 = x * x * 2 + 1;
   a1 = x + 2 - a0;
   a2 = x / 2 + a1;
   a = a0 + a1 + a2;
   printf("a: %d\n", a);
   return a;
}

int main(void)
{
   int i = 1;
```

```
    while(1) {
        i = func(i);
        sleep(3);
    }
    return EXIT_SUCCESS;
}
```

299

来试试显示该程序中的 func()中的变量 a0、a1。在笔者的环境中显示出<value optimized out>，无法显示变量的值。

```
(gdb) b func
Breakpoint 1 at 0x4004c0: file opt.c, line 7.
(gdb) run
Starting program: /home/kyamato/DebugHacks/kyamato/chapter5/kaho.work/opt2

Breakpoint 1, func (x=1) at opt.c:7
7       a0 = x * x * 2 + 1;
(gdb) n
10      a = a0 + a1 + a2;
(gdb) n
8       a1 = x + 2 - a0;
(gdb) n
7       a0 = x * x * 2 + 1;
(gdb) n
10      a = a0 + a1 + a2;
(gdb) n
5   {
(gdb) n
10      a = a0 + a1 + a2;
(gdb) n
11      printf("a: %d\n", a);
(gdb) p a0
$1 = <value optimized out>
(gdb) p a1
$2 = <value optimized out>
```

大多数程序在编译时都启用了优化，因此这种情况十分常见。用 GDB 单步执行时，如果出现被 Optimized out（优化掉）的变量，确认程序行为时就会遇到阻碍，导致调试无法高效率地进行。一个简单的解决方法就是改变源代码，用 printf()

显示 a0、a1 等。但是这种方法对于启动或初始化需要花费很长时间的程序（如 X11，以及由众多进程构成的大型应用程序等），重新构建并执行，直到实际的变量值被显示出来，需要等待很长时间，导致调试无法顺利进行。

调试更换进程的函数

像这种情况下，可以尝试使用 live patch，也称 runtime binary patch 的程序。这种程序原本用于高可用性系统，这样无须重新启动进程就能对 bug 和安全漏洞进行修正。这里介绍一种方法，利用 KAHO 获取运行进程中的变量。把想要显示的变量所在的函数整个用包含 printf()等调试输出的函数替换，这样无须重新启动进程，也就是说，无须等待初始化，就能输出变量的值。

[HACK#50]替换了内核中的函数。本 hack 的方法可以说是它的用户进程版。因此，与[HACK#50]相同，插入 printf()语句后，生成的代码也有可能发生变化，可能导致问题不再复现。

安装 KAHO（Fedora10）

这里介绍在 Fedora10 中安装 KAHO 的方法，KAHO 由内核补丁和命令行工具构成。从下面的网站可以下载。

```
http://sourceforge.net/projects/kaho-01/
```

首先下载内核源代码，并用下面的命令打补丁。

```
# yumdownloader --source kernel
# rpm -ihv kernel-2.6.27.9-159.fc10.src.rpm
```

编辑 spec 文件以应用 KAHO 的内核补丁。spec 文件解压后位于~/rpmbuild/SPECS/kernel.spec。为标识加入了 KAHO 功能的内核，可以将 spec 文件的第 15 行的 buildid 的定义注释掉，并将.local 改成.kaho。

```
%define buildid .kaho
```

在 1360 行前后的# END OF PATCH APPLICATIONS 行之前插入下面这一行。

```
ApplyPatch kaho_kernel_fedora10.patch
```

接下来将 KAHO 的内核补丁放在 SOURCES/目录下，构建内核并安装。

301

```
# cp ~/kaho_kernel_fedora10.patch ~/rpmbuild/SOURCES
# rpmbuild -bb rpmbuild/SPECS/kernel.spec
# rpm -ihv rpmbuild/RPMS/kernel-2.6.27.9.x86_64.rpm
```

安装后重新启动新内核。

KAHO 用到了名为 UTRACE 的功能，该功能包含在 RHEL、Fedora、Cent OS、Asianux 中，但还没被合并到主线上。因此，要想在主线内核上使用 KAHO，就必须同时应用 UTRACE 补丁和 KAHO 补丁。UTRACE 补丁可以从下面的 Web 页面获得。

http://people.redhat.com/roland/utrace/

另外，Fedora10 的内核构建方法可以参考以下网站。

http://www.atmarkit.co.jp/flinux/rensai/linuxtips/all3rebuild.html

至于命令行工具，上面的网站中包含了用于 Fedora10 的二进制 RPM 文件，只需下载并安装即可。

使用 KAHO 进行调试

首先准备下面 3 个文件。

```
/home/kaho_dbg/calc
/home/kaho_dbg/1/debug.so
/home/kaho_dbg/1/debug.cmd
```

calc 就是开头介绍的 calc.c 用优化选项（-02）构建出的可执行文件，而 debug.so 是将下面的源代码 debug.c 用跟 calc 同样的编译选项构建出的共享库。debug.c 把要调试的函数 func()从 calc.c 中完整地复制过来，并改名为 func_debug()，再加上用于输出变量的 printf()语句①。下面就是源代码和构建方法。

```
# cat debug.c
#include <stdio.h>
```

302

```
int func_debug(int x)
{
   int a, a0, a1, a2;
   a0 = x * x * 2 + 1;
   a1 = x + 2 - a0;
   a2 = x / 2 + a1;
   a = a0 + a1 + a2;
```

```
    printf("[Debug]a0: %d, a1: %d, a2: %d\n", a0, a1, a2); ——————————①
    printf("a: %d\n", a);
    return a
}
```

```
# gcc -o debug.so debug.c -fPIC -shared -O2 -g
```

debug.cmd是记载KAHO命令的文件，其格式如下。

```
patch-file 替换函数所在的共享库名
] func 替换函数的名称（符号名）
```

KAHO不仅能替换函数，还能替换数据。第2行的] func关键字指示了要替换的不是数据而是函数。

以下的运行示例中所用的文件如下。

```
# cat debug.cmd
patch-file debug.so
] func func_debug
```

来看看实际的运行情况。为便于显示，我们准备了两个终端（分别称为终端A和终端B），首先在终端A上启动calc。

```
[在终端A上操作]
# ./calc
a: 3
a: -8
a: -145
...
```

接下来在终端B上输入两次KAHO命令，如下所示。第1条KAHO命令把要替换的函数func_debug()加载到calc进程的内存空间中，但是这个状态尚未替换。第2条KAHO命令执行实际的替换操作（该操作称为激活（activation））。此外，KAHO命令的参数末尾指定的“1”的意思是读入$KAHO_HOME/1目录中的命令文件（扩展名为.cmd的文件）。

303

```
[终端B上的操作]
# cd /home/kaho_dbg
# export KAHO_HOME=`pwd`
# kaho -l `pidof calc` 1
```

```
3693,1,"loaded"
# kaho -a `pidof calc` 1
3693,1,"activated"
```

执行上述操作之后，就可以在终端 A 上显示要确认的变量的值了。

```
[激活后在终端A上]
...
[Debug]a0: -795409887, a1: -1135769211, a2: -2101358761
a: 262429437
[Debug]a0: -1258439661, a1: 1520869100, a2: 1652083818
a: 1914513257
...
```

下面的命令可以将替换的函数恢复原状。

```
# kaho -d `pidof calc` 1
3693,1,"deactivated"
# kaho -u `pidof calc` 1
3693,1,"unloaded"
```

参考文献

- A Runtime Code Modification Method for Application Programs
  http://ols.fedoraproject.org/OLS/Reprints-2008/yamato-reprint.pdf
- 开发运行时二进制补丁（KAHO）
  http://blog.miraclelinux.com/yume/files/YLUG-2008-0225update.pdf

总结

304

使用 KAHO 这个 Runtime Binary Patch（运行时二进制补丁）实现，可以将函数整体替换。本 hack 说明了通过该方法查看被 Optimized out 的变量值的方法。

——大和一洋

使用 systemtap 调试运行中的内核（之一）

本 hack 以测量时间的程序为例，介绍 systemtap 的使用方法。

开篇

systemtap 是用 kprobes 创建的工具。它使用类似于 C 语言的特有脚本语言创建侦测器处理程序，用专门的解析程序将侦测器脚本变换为 C 语言，并自动创建内核模块。此时会检查代码的安全性，而且被称为 tapset 的脚本群中事先准备了许多方便的函数，比 kprobes 要好用得多。

准备

要使用 systemtap，需要在编译内核时加上调试信息，而且 stap 命令会自动构建内核模块，因此还要安装内核头文件。笔者使用的是 Fedora9，内核版本为 2.6.27.7-53.fc9，systemtap 版本为 0.8-1.fc9。Fedora9 中要用下述命令安装必要的软件包。

```
# yum intall kernel-devel kernel-headers
# debuginfo-install kernel
```

示例脚本

本 hack 使用的脚本如下所示。这段脚本以 systemtap 自带的 sleeptime.stp 示例脚本为基础，稍加改动而成。该脚本测量 nanosleep()系统调用实际睡眠了多长时间。

```
#!/usr/bin/stap -v

 /*
  * Format is:
  * 12799538 3389 (xchat) nanosleep: 9547
  * 12846944 2805 (NetworkManager) nanosleep: 100964
  * 12947924 2805 (NetworkManager) nanosleep: 100946
  * 13002925 4757 (sleep) nanosleep: 13000717
  */

global start
```

305

```
global entry_nanosleep
global entry_nanosleep_restart

function timestamp:long() {
  return gettimeofday_us() - start
}

function proc:string() {
  return sprintf("%d (%s)", pid(), execname())
}

probe begin {
 start = gettimeofday_us()
}

probe syscall.nanosleep {
  if (uid() != 500) next;
  t = gettimeofday_us(); p = pid()
  entry_nanosleep[p] = t
}

probe syscall.nanosleep.return {
  if (uid() != 500) next;
  t = gettimeofday_us(); p = pid()
  elapsed_time = t - entry_nanosleep[p]
  printf("%d %s nanosleep: %d\n", timestamp(), proc(), elapsed_time)
  delete entry_nanosleep[p]
}

probe kernel.statement("hrtimer_nanosleep@kernel/hrtimer.c:1551") {
  if (uid() != 500) next;
  printf("%d %s nanosleep is interrupted.\n", timestamp(), proc());
  p = pid();
  entry_nanosleep_restart[p] = entry_nanosleep[p];
}

probe kernel.function("hrtimer_nanosleep_restart").return {
  if (uid() != 500) next;
  t = gettimeofday_us(); p = pid()
```

306

```
  elapsed_time = t - entry_nanosleep_restart[p]
  printf("%d %s nanosleep_restart: %d\n", timestamp(), proc(), elapsed_time)
  delete entry_nanosleep_restart[p];
}
```

测量时间

使用 systemtap 进行实时调试（live debug）的一个方便用法就是测量某项处理的时间，但是它不适合几毫秒以下的测量。这是因为尽管 systemtap 自身的额外开销（overhead）与脚本编写方法和机器性能有关，但其影响已经不可忽略。

内核处理中，通过异步事件触发的处理非常多。本 hack 讨论的 nanosleep()就是在定时器中断事件发生时唤醒睡眠状态的进程，而且定时器中断事件是异步事件，系统负载状况会影响事件的发生时机。此外，除了定时器中断之外，其他异步时间也会受到 nanosleep()的影响，例如给睡眠中的任务发送信号等，这会导致 nanosleep()调用返回的时间通常要比要求的睡眠时间长一点。这里我们测量一下实际的睡眠时间。

方法很简单，就是保存执行 nanosleep()的时刻和 nanosleep()返回的时刻，返回时将时间差显示到用户控制台。保存时刻要使用 tapset 提供的 gettimeofday_us()函数。

侦测点定义

插入侦测器的位置称为侦测点。侦测点的定义方法有很多，这里介绍几种经常使用的定义方法。

```
probe begin
probe end
```

定义脚本启动、结束时执行的处理程序。用于脚本内的全局变量初始化，或是在结束时收集日志、按特定格式显示。

```
probe kernel.function("函数名")
probe kernel.function("函数名").return
```

307

分别用于定义“函数名”指定的内核函数在调用时、返回时执行的侦测器。“函

数名”部分可以使用通配符（*），如下所示。

```
probe kernel.function("*init*")
probe kernel.function("*init*@kernel/sched.c")
```

第 1 个例子向所有的内核初始化函数插入侦测器，第 2 个例子给 kernel/sched.c 中定义的所有初始化函数插入侦测器。像这样在函数名后面写上文件名，就能限制对象范围，如果不同文件中包含同名函数，那么用这种写法就能确定要侦测的对象。另外，文件名部分也可以使用通配符。

在内核模块中定义侦测器时写法如下。

```
probe module("模块名").function("函数名")
probe module("模块名").function("函数名").return
```

只需将前面的 kernel 改成 module("模块名")，剩下的部分完全一样。

```
probe syscall.系统调用名
probe syscall.系统调用名.return
```

这是侦测系统调用时的写法。与前面的 kernel.function()的效果相同，在侦测系统调用时，用哪个都没问题。

```
probe kernel.statement("函数名@文件名:行号")
probe kernel.statement(地址)
```

这两行用于在函数中插入侦测器。如果只想调试函数中某个 if 语句的条件匹配部分，这种方法就很方便。但是，如果指定行号，那么某些内核代码的写法会导致侦测器插入的实际位置稍有偏差。这是因为 stap 命令要根据内核的调试信息找出对应的地址，以确定侦测器的位置。而直接指定地址的话，只要地址是正确的指令边界，就能按照期望插入。

侦测内核模块时同样可以像下面这样写。

308

```
probe module("模块名").statement("函数名@文件名:行号")
probe module("模块名").statement(地址)
```

本 hack 中使用下面这种写法在 nanosleep()的信号中断处理处插入侦测器。

```
probe kernel.statement("hrtimer_nanosleep@kernel/hrtimer.c:1551")
```

被侦测部分的内核源代码如下所示。用这种方法，就能创建脚本，获知函数内的

if语句的判断结果是真还是假。

[kernel/hrtimer.c]

```
1527 long hrtimer_nanosleep(struct timespec *rqtp, struct timespec __user *rmtp,
1528                 const enum hrtimer_mode mode, const clockid_t clockid)
1529 {
...
        /* 信号中断时该if语句为假 */
1536      if (do_nanosleep(&t, mode))
1537           goto out;
...
1551      restart = &current_thread_info()->restart_block;
1552      restart->fn = hrtimer_nanosleep_restart;
1553      restart->nanosleep.index = t.timer.base->index;
1554      restart->nanosleep.rmtp = rmtp;
1555      restart->nanosleep.expires = t.timer.expires.tv64;
1556
1557      ret = -ERESTART_RESTARTBLOCK;
1558 out:
1559      destroy_hrtimer_on_stack(&t.timer);
...
```

最后，只用这种写法，系统上执行的所有 nanosleep()都会被侦测，有点难用。因此，可以在各个侦测器处理程序开头加上下述处理，只对测试用户（UID=500）执行的 nanosleep()进行测量。

```
if (uid() != 500) next;
```

309

尝试执行

执行脚本的命令为 stap。默认情况下几乎不会显示任何信息，很难看出侦测器是否生效了，因此笔者在执行时一定会加上 verbose 选项（-v）。

```
# stap -v sleeptime.stp
Pass 1: parsed user script and 45 library script(s) in 230usr/10sys/243real ms.
Pass 2: analyzed script: 6 probe(s), 8 function(s), 15 embed(s), 3 global(s) in 450usr/
320sys/778real ms.
Pass 3: translated to C into "/tmp/stap7JYrBT/stap_2aa0c877c1a26e0304cc924aa0dcfbb3_
13404.c" in 370usr/620sys/988real ms.
Pass 4: compiled C into "stap_2aa0c877c1a26e0304cc924aa0dcfbb3_13404.ko" in 5290usr/
```

```
900sys/6210real ms.
Pass 5: starting run.
```

显示上述信息，说明侦测器生效了。接下来在其他终端上用测试账号执行 nanosleep()，睡眠 10 秒试试看。

```
$ usleep 10000000
```

执行后，执行 stap 命令的终端上就会显示下面的信息。

```
13304586 19724 (usleep) nanosleep: 10000008
```

消息的格式如下所示。显示的经过时间都以微秒为单位。

```
侦测器生效后经过的时间 PID (命令名) nanosleep: 实际经过的时间
```

可见，延迟只有 8 微秒，误差几乎可以忽略，是个优秀的系统。

如果故意用信号中断执行会怎样呢？和刚才一样用 usleep 命令睡眠 10 秒钟，但是，途中发送 SIGSTOP 中断 nanosleep()的执行，等待大约 10 秒之后发送 SIGCONT。

```
11920140 26702 (usleep) nanosleep is interrupted.
11920148 26702 (usleep) nanosleep: 1233996
23249021 26702 (usleep) nanosleep_restart: 12562868
```

310

输出的信息表明 nanosleep()被中断了。从第 2 条消息可以看出，nanosleep()执行后约 1.2 秒后收到了 SIGSTOP。第 3 条信息表明从 nanosleep()返回大约花费了 12.5 秒。也就是说，笔者数秒比实际的时钟慢 2.5 秒左右。

确认完成后就可以结束 systemtap。按 Ctrl-C 键，就可以卸载所有已加载的侦测器并结束 systemtap。

总结

本 hack 以测试 nanosleep()系统调用的实际时间为例，介绍了 systemtap 的使用方法。systemtap 用到了 kprobes，因此 kprobes 能实现的功能大都能用 systemtap 实现。

参考

除了 man page 之外，systemtap 附带的示例脚本和 tapset 都有参考价值。此外，项目页面中的入门教程等文档也十分丰富。

- systemtap 附带的示例脚本
  /usr/share/doc/systemtap-<version>/examples/
- systemtap 附带的 tapset
  /usr/share/systemtap/tapset/
- SystemTap 项目页面
  http://sourceware.org/systemtap/documentation.html

——安部东洋

使用 systemtap 调试运行中的内核（之二）

本 hack 介绍使用 systemtap 查看函数 backtrace、结构内容等的方法。

阅读内核代码时，经常希望知道某个函数从哪里被调用，某个参数的值是什么。本 hack 扩展了“HACK#52 使用 systemtap 调试运行中的内核（之一）”中介绍的 systemtap 脚本（sleeptime.stp），介绍一下利用 systemtap 查看调用跟踪和内核中数据的方法。运行环境为 Fedora9，内核版本为 2.6.27.7-53.fc9，systemtap 版本为 0.8-1.fc9。

311

扩展 sleeptime.stp

我们向 sleeptime.stp 中添加了以下代码，其余部分与[HACK#52]完全相同。

```
#include <linux/thread_info.h>
function res_expires:long(res:long) %{
 struct restart_block *restart = (struct restart_block *)(THIS->res);
 THIS->__retvalue = restart->nanosleep.expires;
%}

probe kernel.function("hrtimer_nanosleep_restart") {
 if (uid() != 500) next;
 printf("%d %s Call trace:\n", timestamp(), proc());
 print_backtrace();
 printf("restart->nanosleep.expires = %u\n", res_expires($restart));
}
```

查看调用跟踪

阅读内核代码时，经常会遇到这种情况：关心的函数太复杂，单从源代码上很难看出调用源，或是需要花费太多时间才能调查出来。此时会想到在内核代码中加入 WARN_ON(1)等并重新编译内核，但是重新编译内核也是件相当麻烦的事儿。不过，使用 systemtap 的话，只需编写侦测器处理程序，而不用这么麻烦了。只需在处理程序中写这样一行即可。

```
print_backtrace();
```

本 hack 的例子在 nanosleep()被信号中断后，接收 SIGCONT 信号恢复运行的位置（hrtimer_nanosleep_restart()）写下了这一行。

查看内核内部数据

函数内使用的变量可以用“$变量”的写法来查看。但是要注意，某些内核代码的写法或侦测点的位置会导致变量无法查看。如果实在不能用“$变量”的方式查看，就只能直接指定地址或直接查看寄存器了。在这种情况下就得对内核的二进制代码进行反汇编，自行查找要查看的地址或寄存器。

312

本 hack 用$restart 查看 hrtimer_nanosleep_restart()的参数 restart 的值。

```
[kernel/hrtimer.c]
1500 long __sched hrtimer_nanosleep_restart(struct restart_block *restart)
1501 {
1502      struct hrtimer_sleeper t;
1503      struct timespec __user *rmtp;
1504      int ret = 0;
1505
1506      hrtimer_init_on_stack(&t.timer, restart->nanosleep.index,
1507                            HRTIMER_MODE_ABS);
1508      t.timer.expires.tv64 = restart->nanosleep.expires;
1509
1510      if (do_nanosleep(&t, HRTIMER_MODE_ABS))
1511           goto out;
```

恢复被中断的 nanosleep()时，如 1508 行所述，会将 restart->nanosleep.expires 保存的值作为唤醒时刻调用 do_nanosleep()，我们用侦测器处理程序显示该值。在查看内核内部结构的成员时，可以在脚本内使用 C 语言。

在脚本内使用 C 语言

在 systemtap 脚本中使用 C 语言有两个要点。

定义 C 语言函数

用以下格式定义函数。

```
function 函数名:返回值类型(参数:参数类型, ...) %{ C语言编写的处理 %}
```

本hack的示例中包含了 restart_block 结构定义的内核头文件<linux/thread_info.h>，并定义了函数 res_expires()，根据参数中给出的指针返回 restart->nanosleep.expires。

在 guru 模式下执行 systemtap

脚本中包含 C 语言时，必须在 guru 模式下执行，执行 stap 命令时加上-g 选项即可进入 guru 模式。但是，guru 模式下 systemtap 执行的安全检查功能就会无效。访问被锁保护的数据时，用户必须自己处理锁，而且在脚本内编写互斥处理也要注意。设计侦测器处理程序时，必须注意不要陷入等待自旋锁、睡眠等状态。因此，一般在编写处理程序时，加锁时要使用 trylock，获取失败就返回错误并结束。

313

尝试执行

如前所述，我们加上-g 选项执行 stap 命令。同[HACK#52]，在其他终端上用测试账号执行 usleep 命令，睡眠 10 秒。再发送 SIGSTOP 和 SIGCONT，激活本 hack 添加的侦测器处理程序。

```
# stap -vg sleeptime.stp
Pass 1: parsed user script and 45 library script(s) in 230usr/10sys/244real ms.
Pass 2: analyzed script: 6 probe(s), 10 function(s), 15 embed(s), 3 global(s) in
490usr/320sys/818real ms.
Pass 3: translated to C into "/tmp/stapv0IXZ1/stap_5879bfa558535efa4dced96a1adff5e3_
13896.c" in 370usr/640sys/1018real ms.
Pass 4: compiled C into "stap_5879bfa558535efa4dced96a1adff5e3_13896.ko" in
5320usr/870sys/6219real ms.
Pass 5: starting run.
9144183 27784 (usleep) nanosleep is interrupted.
9144192 27784 (usleep) nanosleep: 1162593
11000541 27784 (usleep) Call trace:
```

```
0xffffffff812bff5c : hrtimer_nanosleep_restart+0x1/0x62 [kernel]
0xffffffff812c2870 : kretprobe_trampoline_holder+0x4/0x50 [kernel] (inexact)
0xffffffff8101024a : sys_rt_sigreturn+0x558/0x189e [kernel] (inexact)
0xffffffffff000000 : packet_exit+0x7d9e988a/0x7dfe988a [kernel] (inexact)
0xffffffffffff0000 : vgetcpu+0x9ef800/0x0 [kernel] (inexact)
0xffffffffffffff00 : vgetcpu+0x9ff700/0x0 [kernel] (inexact)
0xffffffffffffffff : vgetcpu+0x9ff7ff/0x0 [kernel] (inexact)
restart->nanosleep.expires = 990839225309246
17981629 27784 (usleep) nanosleep_restart: 10000029
```

hrtimer_nanosleep_restart()中插入的侦测器处理程序显示了调用跟踪。调用跟踪中显示了 kretprobe_trampoline_holder()，这一行可以忽视，它是由于插入的侦测器在函数返回时启动而显示的。从上面的结果可知，调用流程如图 6-1 所示。

314

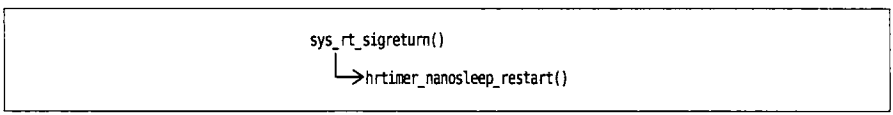


图 6-1 SIGCONT 恢复 nanosleep()运行时的流程

而且，还能查看 restart->nanosleep.expires 的值。

总结

本 hack 介绍了利用 systemtap 查看调用跟踪和内核内部数据的方法，该方法可以代替在内核代码中添加 printk()或 WARN_ON()的调试方法。这样无须重新编译内核，就能在内核中插入调试代码，熟练以后应该能提高工作效率。

参考

- systemtap 附带的示例脚本
  /usr/share/doc/systemtap-<version>/examples/
- systemtap 附带的 tapset
  /usr/share/systemtap/tapset/
- SystemTap 项目页面
  http://sourceware.org/systemtap/documentation.html

——安部东洋

/proc/meminfo 中的宝藏

本 hack 介绍/proc/meminfo 中包含的系统内存相关信息，以及内存泄漏时发生变化的项目。

/proc/meminfo

从/proc/meminfo 中可以获得系统整体（包括内核和进程）的内存使用状况。下面介绍下主要项目，如表 6-1 所示。

315

表 6-1 /proc/meminfo 的显示项目

项目	说明
MemFree	空闲内存总量
Buffers	缓存（块设备数据的缓存）总量。对设备文件（/dev/sda1 等）进行读写时，该值的增加量与读写量相等。读写普通文件时，文件系统的驱动程序会访问设备的超级块（super block）或 inode 块，该值也会少量增加
Cached	页面缓存（普通文件的缓存）总量，读写文件时增加。通常，只要有空闲内存，页面缓存就会一直维持。此外，该容量不包含在 Buffers 和 SwapCached 中
SwapCached	被换出（page out）的数据在换入（page in）之后仍残留在交换设备中的页面的总大小。它表示空闲内存不足时，无须执行 I/O 就能直接释放的内存总量
Active	Active 的 LRU 链表连接的页面的总大小
Inactive	Inactive 的 LRU 链表连接的页面的总大小
Mapped	映射到文件上的页面总大小。与运行中的进程的种类和代码量成比例增加。此外，指定 MAP_SHARED 标志执行 mmap()时也会增加
Slab	Slab allocator（Slab 分配器）的内存使用量。Slab allocator 接受内核或驱动程序发出的、数量较少的内存分配要求（几十字节到几 MB），进行内存分配和释放
PageTables	页表使用的内存总大小。内存使用的地址空间越大，该值也就越大
Committed_AS	进程提交的内存总大小。其中包含尚未指定实际页面的区域
AnonPages	属于匿名区域（anonymous region）的页面总大小。主要作为进程的堆空间使用

内存泄漏的大致数值

上面大致介绍了各个项目数值增值的时机，但是实际上，增加的内存减少的时机非常复杂，很难一概而论。因此，并没有哪个特定项目能够单独发现内存泄漏。即便如此，Committed_AS可以作为进程的内存泄漏估计。Committed_AS的值比预想得多时，就可以怀疑内存泄漏。

我们实际执行了下面的程序，并测定其Committed_AS、MemFree、SwapFree、AnonPages和Cached的值。该程序的5个线程会分配随机大小的内存，最大为4MB，分配时会泄漏一定量的内存（10%）。

316

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <string.h>

const float Pleak = 0.1;
const float Paccess = 0.5;
const float MinAlloc = 1;
const float MaxAlloc = 4*1024*1024;

double frand(void)
{
   return ((double)rand())/RAND_MAX;
}

size_t calc_size(void)
{
   double r = frand();
   double a = pow(MaxAlloc/MinAlloc, r) * MinAlloc;
   return (size_t)a;
}

int leak(void)
{
   if (frand() < Pleak)
      return 1;
```

```
    return 0;
}

int access(void)
{
    if (frand() < Paccess)
        return 1;
    return 0;
}

void *thr_func(void *arg)
{
    while (1) {
        size_t s = calc_size();
        void *p = malloc(s);
        if (p == NULL) {
            printf("Failed to malloc: %d\n", s);
            return NULL;
        }

        if (!access())
            memset(p, 0xaa, s);

        if (!leak())
            free(p);

        sleep(1);
    }
}

int main(void)
{
    pthread_t t1, t2, t3, t4;
    pthread_create(&t1, NULL, thr_func, NULL);
    pthread_create(&t2, NULL, thr_func, NULL);
    pthread_create(&t3, NULL, thr_func, NULL);
    pthread_create(&t4, NULL, thr_func, NULL);
    thr_func(NULL);
```

317

```
    return EXIT_SUCCESS;
}
```

在物理内存 1024MB、交换区 1024MB 的 x86_64 架构的机器上运行该程序，使用的内核为 2.6.18，程序运行结果如图 6-2 所示。随着时间增长，Committed_AS 的值基本上是单调增加的，其原因就是发生了内存泄漏。刚开始执行程序时，MemFree 会随着 Commited_AS 的增加而减少，随着 MemFree 几乎变成 0，Cached 就会开始减少。等 Cached 几乎变成 0 之后，就开始使用交换区，SwapFree 开始减少。值得注意的是，即使开始使用交换区，Committed_AS 也会随着内存泄漏而不断增加，但由于可分配的实际内存有上限，因此 AnonPages 从某处开始基本上是固定值。

318

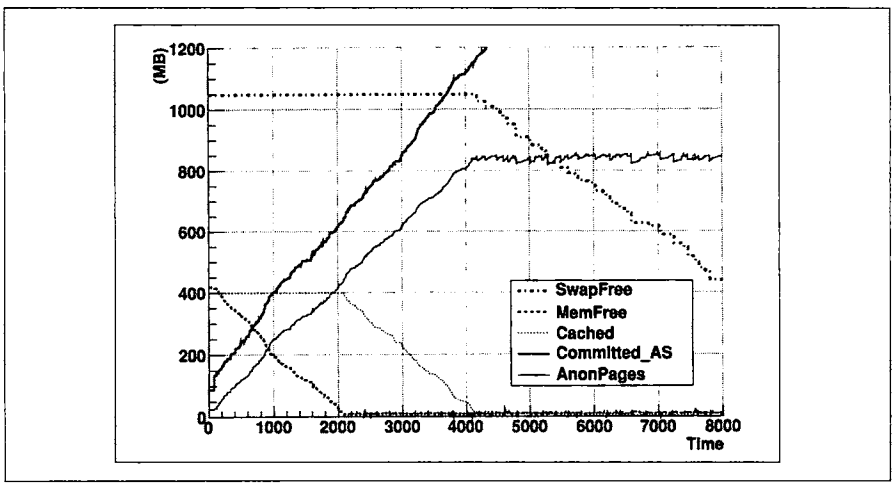


图 6-2 程序运行结果

此外，将刚才程序中的变量 Pleak 改成 0（也就是说设置为没有内存泄漏），执行结果如图 6-3 所示。由于没有内存泄漏，Committed_AS 的值与图 6-2 中的不同，整体上看基本上是固定的。另外，Committed_AS 的增减只是偶尔发生（图中有毛刺的地方），这是由于分配少量内存（大概 1MB 以下）时，glibc 会返回内部已分配的区域。因此从进程整体来看，对操作系统提出的内存请求频率较低，因此进程的内存使用量的变化就很少。

319

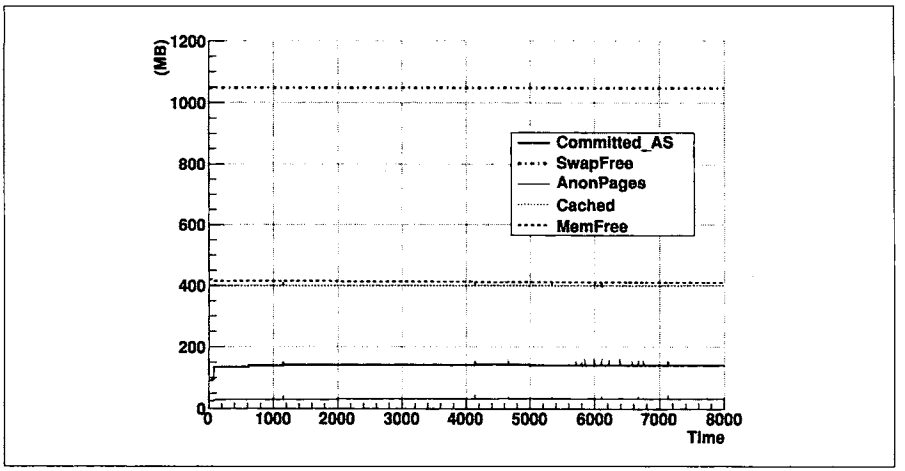


图 6-3 没有内存泄漏时的结果

总结

本 hack 介绍了/proc/meminfo 包含的与内存相关的信息和内存泄漏时的变化项目。

——大和一洋

用/proc/<PID>/mem 快速读取进程的内存内容

本 hack 介绍利用/proc 接口实现比 ptrace 系统调用更快的访问内存空间的方法。

/proc/<PID>/mem 接口

通过虚拟文件/proc/<PID>/mem 可以读取任意进程的内存内容。<PID>就是要读取的进程 ID。

同样的处理用 ptrace 系统调用中的 PTRACE_PEEKDATA 也可以实现。但是，PTRACE_PEEKDATA 一次读取的内存大小在 386 架构下为 4 字节，x86_64 架构下也只有 8 字节。因此，读取量较大时，就要反复调用 ptrace 系统调用，处理时间较长。而通过/proc/<PID>/mem 接口用 read 系统调用，只需一次就能读取任意大小的内存，只需很短时间就能完成，在检查大块数据区域时十分有用。

示例程序

下面的示例程序用参数指定进程，并通过/proc/<PID>/mem 读取进程的内存。第 1 个参数为要读取的进程的 PID，第 2 个参数为要读取的内存的起始地址，第 3 个参数为读取大小。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

int main( int argc, char *argv[] )
{
   const int npath = 32;
   char path[npath];      /* /proc/<PID>/mem */
   pid_t pid;             /* 要读取的进程的 PID */
   int target_addr;       /* 读取的起始地址 */
   int read_sz;           /* 读取大小 */
   int fd;
   off_t ofs;
   ssize_t sz;
   unsigned char *buf;

   /* 读取参数 */
   if (argc < 4) {
      printf("Usage:\n");
      printf(" # %s pid target_addr(hex) read_sz(hex)\n\n", argv[0]);
      exit(1);
   }
   pid = atoi(argv[1]);
   sscanf(argv[2], "%x", &target_addr);
   sscanf(argv[3], "%x", &read_sz);
```

320

```
/* 分配读取用的缓冲区 */
buf = malloc(read_sz);
if (buf == NULL) {
   fprintf(stderr, "Failed to malloc (size: %d)\n", read_sz);
   exit(1);
}

/* 把要读取的对象变成跟踪状态，并打开/proc/<PID>/mem */
if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) != 0) {
   fprintf(stderr, "Failed to attach (pid: %d)\n", pid);
   exit(1);
}
if (waitpid(pid, NULL, 0) < 0) { /* 等待ATTACH完成 */
   fprintf(stderr, "Failed to waitpid (pid: %d)\n", pid);
   exit(1);
}

snprintf(path, npath, "/proc/%d/mem", pid);
fd = open(path, O_RDONLY);
if (fd < 0 ) {
   fprintf(stderr, "Failed to open: %s\n", path);
   ptrace(PTRACE_DETACH, pid, NULL, NULL);
   exit(1);
}

/* 找到目的地址并读取 */
ofs = lseek(fd, target_addr, SEEK_SET);
if (ofs == (off_t)-1) {
   fprintf(stderr, "Failed to lseek, errno: %d\n", errno);
   ptrace(PTRACE_DETACH, pid, NULL, NULL);
   exit(1);
}

sz = read(fd, buf, read_sz);
if (sz != read_sz) {
   fprintf(stderr, "Failed to read, errno: %d\n", errno);
   ptrace(PTRACE_DETACH, pid, NULL, NULL);
   exit(1);
```

321

```
    }

    /* 显示内存内容 */
    for (sz = 0; sz < read_sz; sz++) {
      if (sz%16 == 0) printf("\n");
      printf("%02x ", buf[sz]);
    }
    printf("\n");

    close(fd);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);
    free(buf);
    return EXIT_SUCCESS;
}
```

322

上面的示例程序的基本流程为：首先把要读取的对象进程改成跟踪状态，然后打开/proc/<PID>/mem，找到目的地址后用 read()读出。如果不用 ptrace(PTREACE_ATTACH, ...)将读取对象进程改成跟踪状态，之后的处理就会失败。

评测

我们来测试一下使用/proc/<PID>/mem 和 ptrace(PTRACE_PEEKDATA, ...)究竟有多大差异。使用的机器为 Core 2 6400 2.13GHz，内核为 Linux 2.6.22.1(i386)，要读取的进程拥有 128MB 数据区域，该区域中写入了随机值。用两种方法读取数据区域的时间如表 6-2 所示。4KB 以下时两者基本上没有差距，当读取数据大小超过 64KB 时，用 read()读取更快。特别是读取 16MB 数据的时间差了将近 100 倍。

表 6-2 read()和 ptrace()的读取时间

读取大小	read()	ptrace()
4KB	0.009s	0.010s
64KB	0.009s	0.023s
1MB	0.011s	0.242s
16MB	0.048s	3.675s

总结

本节介绍了利用/proc/<PID>/mem 接口快速读取任意进程的内存的方法。根据测试，

读取的数据超过几十 KB 时，这种方法要比 ptrace(PTRACE_PEEKDATA, ...)快得多。

——大和一洋

OOM Killer 的行为和原理

本 hack 介绍 OOM Killer 的行为和原理。

Linux 中有个 Out Of Memory(OOM) Killer 功能，当系统的内存和交换区用尽时，会采取保证内存的最终手段，给进程发送信号使之强制结束。

由于该功能的存在，内存无法释放的情况下持续分配内存导致系统停止的现象就不会出现，而且能检测出消耗内存过大的进程。本 hack 介绍内核 2.6 中的 OOM Killer。

323

确认行为和日志

在验证系统或做负载测试时，经常会出现本应正常运行的进程终止、ssh 连接突然中断，导致无法重新连接的情况。

此时就要确认日志。有时会输出下面的信息。

```
Pid: 4629, comm: stress Not tainted 2.6.26 #3

Call Trace:
 [<ffffffff80265a2c>] oom_kill_process+0x57/0x1dc
 [<ffffffff80238855>] __capable+0x9/0x1c
 [<ffffffff80265d39>] badness+0x16a/0x1a9
 [<ffffffff80265f59>] out_of_memory+0x1e1/0x24b
 [<ffffffff80268967>] __alloc_pages_internal+0x320/0x3c2
 [<ffffffff802726cb>] handle_mm_fault+0x225/0x708
 [<ffffffff8047514b>] do_page_fault+0x3b4/0x76f
 [<ffffffff80473259>] error_exit+0x0/0x51

Node 0 DMA per-cpu:
CPU    0: hi:    0, btch:  1 usd:  0
CPU    1: hi:    0, btch:  1 usd:  0
...
```

```
Active:250206 inactive:251609 dirty:0 writeback:0 unstable:0
 free:3397 slab:2889 mapped:1 pagetables:2544 bounce:0
Node 0 DMA free:8024kB min:20kB low:24kB high:28kB active:8kB inactive:180kB present:
7448kB pa ges_scanned:308 all_unreclaimable? yes
lowmem_reserve[]: 0 2003 2003 2003
...
Node 0 DMA: 6*4kB 4*8kB 2*16kB 2*32kB 5*64kB 1*128kB 3*256kB 1*512kB 2*1024kB 2*2048kB
0*4096k B = 8024kB
Node 0 DMA32: 1*4kB 13*8kB 1*16kB 6*32kB 2*64kB 2*128kB 1*256kB 1*512kB 0*1024kB 0*2048kB
1*40 96kB = 5564kB
29 total pagecache pages
Swap cache: add 1630129, delete 1630129, find 2279/2761
Free swap  = 0kB
Total swap = 2048248kB
Out of memory: kill process 2875 (sshd) score 94830592 or a child
Killed process 3082 (sshd)
```

324

最后显示 Out of memory（内存不足），这一行显示了 OOM Killer 运行的原因。无法重新连接的原因就是 sshd 被 OOM Killer 结束了，必须重新启动 sshd，才能登录。

OOM Killer 通过结束进程的方式保证空闲内存，下面介绍一下 OOM Killer 选择进程的方法。

选择进程的方法

内存耗尽时，OOM Killer 会遍历所有进程，并给每个进程评分，然后向得分最高的进程发送信号。

评分方法

OOM Killer 在评分时考虑了很多因素，对每个进程考虑下面 1～9 点。

1. 首先以进程的虚拟内存大小作为计算分数的基准。虚拟内存大小可以从 ps 命令的 VSZ 或/proc/<PID>/status 的 VmSize [注1]查看。耗费内存越多的进程，起始分数就越高，1KB 算做 1 分。如果进程耗费了 1GB 内存，分数就大概是 1000000。

注 1： /proc/<PID>/status 的 VmSize 与分数稍有不同。

2. 执行了 swapoff 系统调用的进程的分数被设置为最大值（unsigned long 的最大值）。禁用交换区这种行为与内存不足的状况格格不入，因此迅速成了 OOM Killer 的目标。

3. 如果进程是父进程，那么所有子进程的内存大小的一半作为分数加到父进程上。

4. 根据进程的 CPU 使用时间和启动时间调整分数。启动时间越长，或者是正在运行的进程，就认为其重要性越高，分数就越低。

首先用分数除以 CPU 使用时间（10 秒为单位），然后开平方。如果 CPU 使用时间为 90 秒，那么以 10 秒为单位就是 9，平方根为 3，用分数除以 3。

325

此外，进程启动后经过的时间也用于调整分数。将分数除以启动时间（1000 秒为单位）的平方根的平方根。对于已运行了 16000 秒的进程，16 的平方根是 4，再开平方就是 2，用分数除以 2。这两种方法都是将运行时间长的进程看做重要的进程。

源代码的注释中写的是以 10 秒、1000 秒为单位，但实际上用的是位运算，以 8 秒、1024 秒为单位进行计算。

5. 被 nice 等命令设置成优先级较低的进程的分数加倍。例如，nice -n 命令设置成 1～19 的进程，其分数就是 2 倍。

6. 超级用户的进程一般都比较重要，分数设置成原分数的 1/4。

7. 用 capset(3)等设置了 CAP_SYS_RAWIO[注2]功能的进程的分数为原分数的 1/4。能直接访问硬件的进程判断为重要进程。

8. 对于 cgroup 的情况，要看导致 OOM Killer 启动的进程被允许的内存结点之外的其他内存结点，如果某个进程只允许了“其他内存结点”，就将分数设置为原分数的 1/8。

9. 最后根据 proc 文件系统的 oom_adj 调整分数。

根据这些规则为所有进程评分，然后给分值最高的进程发送 SIGKILL 信号（2.6.10 之前（含 2.6.10），如果设置了 CAP_SYS_RAWIO 则发送 SIGTERM，否则发送 SIGKILL）。

注 2：　该选项默认有效。

通过/proc/<PID>/oom_score可以查看各进程的分数。

但是，init（PID为1）进程不是OOM Killer的处理对象。如果被选中的进程包含子进程，就先向子进程发送信号。

此外，给选中的进程发送信号之后，再访问系统中的所有线程，即使不属于同一线程组（TGID不同），只要存在与选中进程共享了同一片内存区域的进程，就也给它发送信号。

proc中与OOM Killer有关的内容

下面介绍proc中与OOM Killer有关的内容。

326

/proc/<PID>/oom_adj

设置/proc/<PID>/oom_adj可以调整分数。调整范围为-16～15。正值更容易被OOM Killer选中，而负值降低选中可能性。设置为3，分数就成了2^3倍，设置为-5则为$(1/2)^5$。

“-17”是个特别的数值，可以禁止OOM Killer向该进程发送信号（从内核2.6.12开始支持-17）。

如果想在OOM Killer启动后仍可以远程登录，可以从候选对象中去除sshd，设置方法如下。

```
# cat /proc/`cat /var/run/sshd.pid` /oom_score
15
# echo -17 > /proc/`cat /var/run/sshd.pid`/oom_adj
# tail /proc/`cat /var/run/sshd.pid`/oom_*
==> /proc/2278/oom_adj <==
-17
==> /proc/2278/oom_score <==
0
```

/proc/<PID>/oom_adj的文档出现在内核2.6.18以后的Documentation/filesystems/proc.txt中，实际上从内核2.6.11开始就可以使用了。

/proc/sys/vm/panic_on_oom

将/proc/sys/vm/panic_on_oom设置为1，那么OOM Killer启动时不会发送信号，而是引发panic。

```
# echo 1 > /proc/sys/vm/panic_on_oom
```

/proc/sys/vm/oom_kill_allocating_task

2.6.24 版以后的内核中的 proc 文件系统中有个 oom_kill_allocating_task，将其设置为非 0 值，引发 OOM Killer 启动的进程自身就会接收到信号，而不再对所有进程进行评分。

```
# echo 1 > /proc/sys/vm/oom_kill_allocating_task
```

这样就无须遍历进程，但这种方法不会考虑进程优先级、root 权限等，而只是单纯地发送信号而已。

327

/proc/sys/vm/oom_dump_tasks

2.6.25 版以后的内核中，将 oom_dump_tasks 设置为非 0 值，OOM Killer 启动时就会输出进程列表。

设置示例如下。

```
# echo 1 > /proc/sys/vm/oom_dump_tasks
```

信息如下所示。可以通过 dmesg 或 syslog 确认。

```
[ pid ] uid    tgid    total_vm   rss cpu oom_adj name
[   1]    0    1       2580       1  0     0     init
[ 500]    0    500     3231       0  1    -17    udevd
[ 2736]   0    2736    1470       1  0     0     syslogd
[ 2741]   0    2741    944        0  0     0     klogd
[ 2765]   81   2765    5307       0  0     0     dbus-daemon
[ 2861]   0    2861    944        0  0     0     acpid
...
[ 3320]   0    3320    525842     241215  1 0 stress
```

内核配置

2.4 版内核的内核配置中可以启用/禁用 OOM Killer.

```
General setup
  [ ] Select task to kill on out of memory condition
```

2.6 版内核中没有该配置，就无法设置了。

RHEL 的特点

RHEL5 中，OOM Killer 的行为比标准内核中还要谨慎。RHEL5 会计算 OOM Killer 被调用的次数，只有在一定时间内调用一定次数后才会启动。

1. OOM Killer 上次调用到本次调用之间超过 5 秒，就将调用次数清零。这是为了避免内存负载突发性膨胀导致进程被终止。
2. 计数器清零后 1 秒内被调用，则不算调用次数。

328

3. OOM Killer 调用次数不到 10 次，则不实际启动。只有 OOM Killer 被调用 10 次才认为发生了内存不足。
4. 最后一次 OOM Killer 启动之后 5 秒之内不会再次启动 OOM Killer，因此运行频率最大只有 5 秒一次。这样可以避免无意义地结束多个进程，也可让 OOM Killer 等着接受了信号的进程结束（释放内存）。
5. OOM Killer 启动时计数器清零。

也就是说，只有在 5 秒内 OOM Killer 被连续调用 10 次以上才会启动。

这些限制本来是标准内核 2.6.10 才有的功能，而基于 2.6.9 的 RHEL4 也加上了这些限制。

在 RHEL4 上的行为确认

我们在 RHEL4（内核 2.6.9）上确认了 OOM Killer 的行为。下例中的内存和交换区均为 2GB，使用负载测试工具 stress 故意消耗内存。

stress 是个内存、CPU 和磁盘 I/O 的负载测试工具，它可以单独增加某项的负载，也可以同时给几项增加负载。stress 运行期间接收到信号就会输出信息并结束。

```
# wget -t0 -c http://weather.ou.edu/~apw/projects/stress/stress-1.0.0.tar.gz
# tar zxvf stress-1.0.0.tar.gz
# cd stress-1.0.0
# ./configure ; make ; make install
# stress --vm 2 --vm-bytes 2G --vm-keep   /* 两个进程，分别消耗 2G 内存 */
```

```
stress: info: [17327] dispatching hogs: 0 cpu, 0 io, 2 vm, 0 hdd
stress: FAIL: [17327] (416) <-- worker 17328 got signal 15 /* 接收到SIGTERM信号 */
stress: WARN: [17327] (418) now reaping child worker processes
stress: FAIL: [17327] (452) failed run completed in 70s
```

此时控制台如下。

```
oom-killer: gfp_mask=0xd0
Mem-info:
...
Node 0 Normal per-cpu:
cpu 0 hot: low 32, high 96, batch 16
cpu 0 cold: low 0, high 32, batch 16
cpu 1 hot: low 32, high 96, batch 16
cpu 1 cold: low 0, high 32, batch 16
...
Free pages:     13144kB (0kB HighMem)   /* 由于存在reserve pages，该值不为0 */
Active:251180 inactive:249985 dirty:0 writeback:0 unstable:0 free:3286 slab:2731
mapped:500625 pagetables:2245
...
Node 0 Normal free:1424kB min:1428kB low:2856kB high:4284kB active:1004592kB inactive:
999940kB present:2080512kB
pages_scanned:2384217 all_unreclaimable? yes
protections[]: 0 0 0
...
Node 0 DMA: 4*4kB 5*8kB 1*16kB 4*32kB 2*64kB 3*128kB 1*256kB 1*512kB 0*1024kB 1*2048kB
2*4096kB = 11720kB
Node 0 Normal: 0*4kB 0*8kB 1*16kB 2*32kB 1*64kB 0*128kB 1*256kB 0*512kB 1*1024kB 0*2048kB
0*4096kB = 1424kB
...
Swap cache: add 524452, delete 524200, find 60/102, race 0+0
Free swap:        0kB                   /* 剩余的交换区为0 */
524224 pages of RAM                     /* 每页为4KB，因此内存的大小为2GB */
10227 reserved pages                    /* 内核内部保留的内存 */
19212 pages shared
253 pages swap cached
Out of Memory: Killed process 17328 (stress).   /* 被信号结束的进程 */
```

329

尽管标准内核中无法禁用 OOM Killer，但 RHEL4 有个/proc/sys/vm/oom-kill，可以将其禁用。

```
# echo 0 > /proc/sys/vm/oom-kill
```

或

```
# sbin/sysctl -w vm.oom-kill=0
```

禁用后 OOM Killer 就不再发送信号，但上述内存信息还是会输出。

在 RHEL5 上的行为确认

330

下面在 RHEL5（内核 2.6.18）上确认 OOM Killer 的行为。确认方法与 RHEL4 中相同。

```
# stress --vm 2 --vm-bytes 2G --vm-keep
stress: info: [11779] dispatching hogs: 0 cpu, 0 io, 2 vm, 0 hdd
stress: FAIL: [11779] (416) <-- worker 11780 got signal 9      /* SIGKILL */
stress: WARN: [11779] (418) now reaping child worker processes
stress: FAIL: [11779] (452) failed run completed in 46s
```

此时的控制台画面如下。其中增加了 backtrace，有助于调试。

```
Call Trace:
 [<ffffffff800bf551>] out_of_memory+0x8e/0x321
 [<ffffffff8000f08c>] __alloc_pages+0x22b/0x2b4
...
 [<ffffffff800087fd>] __handle_mm_fault+0x208/0xe04
 [<ffffffff80065a6a>] do_page_fault+0x4b8/0x81d
 [<ffffffff800894ad>] default_wake_function+0x0/0xe
 [<ffffffff80039dda>] tty_ldisc_deref+0x68/0x7b
 [<ffffffff8005cde9>] error_exit+0x0/0x84

Mem-info:
...
Swap cache: add 512503, delete 512504, find 90/129, race 0+0
Free swap  = 0kB
Total swap = 2048276kB
Free swap:        0kB
524224 pages of RAM
42102 reserved pages
78 pages shared
0 pages swap cached
Out of memory: Killed process 11780 (stress)
```

RHEL5 中不再有/proc/sys/vm/oom-kill。

总结

本 hack 介绍了 OOM Killer 的原理和设置行为。系统动作异常时检查一下 syslog 等，如果有 OOM Killer 的输出，就说明发生了内存不足。

331

参考文献

- stress
  http://weather.ou.edu/~apw/projects/stress/

——大岩尚宏

错误注入

本 hack 介绍 Linux 内核选项中的错误注入（fault injection）。

所谓错误注入，就是在测试软件时故意使之出错，以执行通常不会执行到的错误处理代码。这样可以提高软件的可靠性，而且，只要正确进行了错误处理，就不会被那些一目了然的问题烦恼，因此这个技术很有价值，可以尝试一下。

错误注入技术使用了一个名为 failmalloc 的库（http://www.nongnu.org/failmalloc/）。链接该库，就会在 malloc 等内存分配函数中插入钩子，就可以故意造成内存分配失败。

一般程序中，执行内存分配函数后都会有内存分配失败时的错误处理代码，或是结束程序，或是返回错误代码以通知调用者。而调用者在调用该函数之后，也有同样的错误处理代码，以进一步通知上层函数。

也就是说，该方法不仅能测试内存分配函数之后的错误处理代码，还可能测试到程序内许多位置上的错误处理代码。

Linux 内核的错误注入

受到 failmalloc 的影响，Linux 内核中也实现了同样的机制，这就是本 hack 要介绍的 Linux 内核错误注入。它实现了一个框架，通过该框架可以方便地实现各种

错误注入；还实现了表 6-3 列出的各种实用的错误注入。

332

表 6-3 错误注入的种类

种类	说明
fail_page_alloc	页面分配器内存分配失败
failslab	slab 分配器内存分配失败
fail_make_request	磁盘 I/O 请求失败
fail_io_timeout	磁盘 I/O 超时

该功能可以在 2.6.21 版本以后使用。但是，fail_io_timeout 在 2.6.28 版本以后才被支持。

failslab

下面介绍一下 failslab，它实现了 slab 分配器的内存分配的错误注入。

slab 分配器

根据用途，Linux 内核内部进行的内存分配有很多种，而最常用的一种就是 slab 分配器（请参见《深入理解 Linux 内核 第 3 版》[注3] 等书）。

此处的目的是尽量测试多个位置的错误处理，找出更多的 bug，因此让 slab 分配器失败是最合适的。

下面以 slab 分配器中的典型内存分配函数 kmalloc()为例介绍一下。

```
void *kmalloc(size_t size, gfp_t flags)
```

内存分配成功时，返回已分配的内存地址，失败则返回 NULL 指针。

size 参数以字节为单位指定要分配的内存大小，gfp_mask 参数为多个 GFP 标志的逻辑与，GFP 标志指示分配的内存属性和查找空闲内存时的行为。

启用 failslab 后，即使实际上 slab 分配器可以分配内存，也可以按照指定的条件使之失败（kmalloc()函数返回 NULL 指针）。

注3：《深入理解 Linux 内核 第 3 版》，中国电力出版社，2007 年，ISBN：9787508353944。——译者注

333

启用 failslab

要启用 failslab，应当在内核中启用以下 4 个选项。

- CONFIG_SLAB 或 CONFIG_SLUB
- CONFIG_FAULT_INJECTION
- CONFIG_FAILSLAB
- CONFIG_FAULT_INJECTION_DEBUG_FS

用 make menuconfig 配置的话，启用上述 4 个选项的步骤如下。

1. 从顶层菜单的 General setup 选择 Choose SLAB allocator，再选择 SLAB 或 SLUB。

   这里可以选择 3 种 slab 分配器的实现（SLAB、SLUB、SLOB），但 failslab 只支持 SLAB 和 SLUB（2.6.29 版本以前的内核仅支持 SLAB）。

2. 从顶层菜单的 Kernel hacking 中选择 Fault-injection framework，就会显示下面两个选项，将它们都选中。

- Fault-injection capability for kmalloc()
- Debugfs entries for fault-injection capabilities

设置参数

产生错误注入的条件，要通过 debugfs 挂载目录下的 failslab 目录中的文件设置。

failslab 目录中的文件如下所示（后文的运行范例中，debugfs 的挂载目录为 /debugfs）。

```
$ ls /debugfs/fail_page_alloc
ignore-gfp-wait interval probability space task-filter times verbose
```

向这些文件写入值即可改变设置，读取这些文件即可知道当前设置。

334

probability:

用百分比指定错误注入发生的比例（初始值为 0）。

例如，将 probability 设置为 1，那么调用 slab 分配器的内存分配函数时，就会

interval:

设置发生一次错误注入之后，不再发生错误的次数（初始值为1）。

例如，将interval设置为100，那么slab分配器的内存分配失败之后的100次内存分配就不会失败。但是，那些不是由于错误注入导致，而是真正的内存不足导致的内存分配失败，当然不受影响。

要想让失败概率小于1%，可以将probability设置为1以上，再将interval设置为较大的数值。

times:

设置错误注入发生次数的上限。

例如，设置times为10，内存分配就仅失败10次。

设置为-1，就一直发生（初始值为1）。

space:

设置产生错误注入之前的内存分配总量，单位为字节。

例如，将space设置为41943040（=40MB），那么slab分配器分配的内存数量达到40MB之前，错误注入不会引发内存分配失败（初始值为0）。

verbose:

设置错误注入发生时的内核信息的详细程度（初始值为2）。

设置为1时，错误注入引发失败时，内核日志中输出以下信息。

335

```
FAULT_INJECTION: forcing a failure
```

设置为2时，除了上述信息之外，还会显示错误发生时的调用跟踪。

```
$ dmesg
...
FAULT_INJECTION: forcing a failure
Pid: 2237, comm: rsyslogd Not tainted 2.6.28-rc9 #9
Call Trace:
```

```
[<ffffffff811557eb>] should_fail+0xc5/0x101
 [<ffffffff810ae093>] should_failslub+0x2b/0x34
 [<ffffffff810aebd6>] kmem_cache_alloc+0x20/0xb0
 [<ffffffff81084683>] mempool_alloc_slab+0x11/0x13
 [<ffffffff8108478f>] mempool_alloc+0x4a/0x106
 [<ffffffff81084683>] ? mempool_alloc_slab+0x11/0x13
 [<ffffffff8108478f>] ? mempool_alloc+0x4a/0x106
 [<ffffffff810d5966>] bvec_alloc_bs+0x90/0xd7
 [<ffffffff810d5a21>] bio_alloc_bioset+0x74/0xca
 [<ffffffff810d5ae1>] bio_alloc+0x10/0x1f
 [<ffffffff810d1805>] submit_bh+0x68/0x109
 [<ffffffff810d35f6>] __block_write_full_page+0x1d8/0x2da
 [<ffffffffa004ad91>] ? ext3_get_block+0x0/0xfc [ext3]
 [<ffffffff810d37ca>] block_write_full_page+0xd2/0xd7
 [<ffffffffa004c581>] ext3_ordered_writepage+0xd1/0x17b [ext3]
```

设置为 0 则不会显示任何信息。

task-filter:

设置是否仅在特定进程上进行错误注入。Y 为启用，N 为禁用（初始值为 N）。启用后，假设要进行错误注入的进程 ID 为<PID>，就要在/proc/<PID>/make-it-fail 中写入 1。

这样，只有在该进程的上下文环境中的 slab 分配器的内存分配，才会失败。

在其他进程的上下文环境及中断上下文环境中，不会失败。fork 生成的子进程也会继承该属性，因此用下面的脚本执行命令，就能仅让命令中的 slab 分配器失败。

336

failcmd 脚本

```
#!/bin/sh

echo 1 > /proc/self/make-it-fail
exec $@
```

[运行范例]

```
$ sh failcmd <command> <args...>
```

ignore-gfp-wait:

设置在分配 slab 时指定的 GFP 掩码中如果包含__GFP_WAIT 标志时，是否引发错误注入。设置为 Y，不发生错误注入，N 则发生（初始值为 Y）。禁用该设置时，通常要结合使用 task-filter。详情参见 **"HACK#58 利用错误注入发现 Linux 内核的潜在 bug"**。

总结

本 hack 介绍了 Linux 内核的附加功能——错误注入（fault injection）。

参考文献

- Wikipedia: Fault injection
  http://en.wikipedia.org/wiki/Fault_injection
- Failmalloc
  http://www.nongnu.org/failmalloc/
- 内核自带文档 fault-injection.txt
  Documentation/fault-injection/fault-injection.txt

——美田晃伸

337

利用错误注入发现 Linux 内核的潜在 bug

本 hack 以 failslab 为例，介绍利用错误注入技术发现 Linux 内核的潜在 bug 的步骤。

引发错误注入

下面来试试引发错误注入。尝试可能导致 kernel panic 或文件系统被破坏，因此要在测试环境中进行。

将错误注入的发生次数限制为 10 次。这是为了防止设置错误等导致错误注入无限发生，从而陷入无法操作的状态。

```
# echo 10 > /debugfs/failslab/times
```

错误注入的发生概率设置为 1%。设置之后，错误注入就可以发生了。

```
# echo 1 > /debugfs/failslab/probability
```

错误注入多与压力测试组合进行，这里仅用简单的命令为系统增加负载。

```
# dd if=/dev/zero of=/tmp/junk
^C
```

实际有无发生错误注入，只要将 verbose 设置为 1 以上，再用 dmesg 查看内核的日志信息就可以知道。这里将 times 设置为 10，因此只需显示 times 就能知道正确的发生次数。

```
# cat /debugfs/failslab/times
0
```

times 由 10 变成了 0，因此错误注入已发生了 10 次。以后只要不重新设置 times，错误注入就不会发生。

让几乎不会失败的 slab 分配过程失败

刚才的例子中启用了 ignore-gfp-wait（默认），下面的例子中禁用了 ignore-gfp-wait。这样，即使在 slab 分配时指定了__GFP_WAIT 标志也会失败。 338

指定了__GFP_WAIT 标志的 slab 分配会在找不到空闲内存时睡眠，因此实际上几乎不可能失败。也就是说，本例中许多根本执行不到的错误处理也能被执行，引发通常无法复现的内核 bug 的可能性就提高了。

相应地，如果对所有进程一视同仁，那么用户应用程序就可能由于系统调用失败而结束，因此禁用 ignore-gfp-wait 时，要结合使用 task-filter，避免给其他进程造成影响。

先把发生概率设置为 0%，使错误注入不再发生。

```
# echo 0 > /debugfs/failslab/probability
```

然后禁用 ignore-gfp-wait 并启用 task-filter。

```
# echo N > /debugfs/failslab/ignore-gfp-wait
# echo Y > /debugfs/failslab/task-filter
```

跟刚才的例子一样，将错误注入发生次数设置为 10，发生概率设置为 1%。

```
# echo 10 > /debugfs/failslab/times
# echo 1 > /debugfs/failslab/probability
```

本次启用了 task-filter，因此只有明确指定的命令上才会发生错误注入。

要像刚才的例子那样使用 dd 命令进行测试，只需将命令作为 failcmd 脚本的参数即可（请将[HACK#57]介绍的 failcmd 脚本放在 PATH 包含的目录中，并设置好执行权限）。

```
# failcmd dd if=/dev/zero of=/tmp/junk
dd: writing to `/tmp/junk`: Cannot allocate memory
105+0 records in
104+0 records out
53248 bytes (53 kB) copied, 0.00176169 s, 30.2 MBps
```

339

dd 命令中，write 系统调用产生了错误注入，导致内存分配失败，dd 命令结束。

内核发生 Oops

像上面的例子那样，错误注入可以跟各种压力测试结合运行。下面的内核日志就是将错误注入与 LTP（http://ltp.sourceforge.net/）结合运行时发生的 Oops。

```
$ dmesg
...
FAULT_INJECTION: forcing a failure
Pid: 8187, comm: mincore01 Not tainted 2.6.28-rc9 #6
Call Trace:
 [<ffffffff811537cb>] should_fail+0xc5/0x101
 [<ffffffff810ac4ff>] should_failslab+0x36/0x3f
 [<ffffffff810ad0ef>] kmem_cache_alloc+0x18/0xfe
 [<ffffffffa063137c>] ext4_mb_free_metadata+0x6b/0x336 [ext4]
 [<ffffffffa06319b4>] ext4_mb_free_blocks+0x36d/0x5dd [ext4]
 [<ffffffffa0615475>] ext4_free_blocks+0x7b/0xcf [ext4]
 [<ffffffffa061b4e7>] ext4_clear_blocks+0xe8/0xf4 [ext4]
 [<ffffffffa061b5a3>] ext4_free_data+0xb0/0x103 [ext4]
 [<ffffffffa061b91f>] ext4_truncate+0x175/0x4d4 [ext4]
 [<ffffffffa062c9db>] ? __ext4_journal_dirty_metadata+0x1f/0x48 [ext4]
 [<ffffffffa0618cc7>] ? ext4_mark_iloc_dirty+0x454/0x4da [ext4]
```

```
[<ffffffffa06193f7>] ? ext4_mark_inode_dirty+0x181/0x196 [ext4]
[<ffffffffa061dd21>] ext4_delete_inode+0x109/0x1cc [ext4]
[<ffffffffa061dc18>] ? ext4_delete_inode+0x0/0x1cc [ext4]
[<ffffffff810c3d8d>] generic_delete_inode+0xc7/0x147
[<ffffffff810c3e22>] generic_drop_inode+0x15/0x171
[<ffffffff810c34fd>] iput+0x61/0x65
[<ffffffff810bccc5>] do_unlinkat+0xfc/0x173
[<ffffffff81075831>] ? audit_syscall_entry+0x141/0x17c
[<ffffffff810bcd4d>] sys_unlink+0x11/0x13
[<ffffffff8100bfaa>] system_call_fastpath+0x16/0x1b
BUG: unable to handle kernel NULL pointer dereference at 0000000000000030
IP: [<ffffffffa063137c>] ext4_mb_free_metadata+0x6b/0x336 [ext4]
PGD 3e8a5067 PUD 33083067 PMD 0
Oops: 0002 [#1] SMP                                                    340
last sysfs file: /sys/devices/pci0000:00/0000:00:1e.0/0000:0a:0c.0/local_cpus
CPU 1
Modules linked in: ext4 jbd2 crc16 bridge stp bnep rfcomm l2cap
...
ata_piix ata_generic libata sd_mod scsi_mod ext3 jbd mbcache uhci_hcd
ohci_hcd ehci
_hcd [last unloaded: freq_table]
Pid: 8187, comm: mincore01 Not tainted 2.6.28-rc9 #6
RIP: 0010:[<ffffffffa063137c>]  [<ffffffffa063137c>]
ext4_mb_free_metadata+0x6b/0x336 [ext4]
RSP: 0018:ffff8800099a5b08  EFLAGS: 00010202
RAX: 0000000000000000 RBX: ffff8800099a5bb8 RCX: 0000000000018280
RDX: 0000000000018280 RSI: ffffffff8143f7f0 RDI: 00007fffa27f13f8
RBP: ffff8800099a5b58 R08: 00000000001827f R09: 0000000000000000
R10: 0000000000000010 R11: 00000000ffffffff R12: ffff88003ecb5cc8
R13: ffff88002a8b8000 R14: 0000000000000000 R15: 000000000000106e
FS:  00007fcc9a7c66f0(0000) GS:ffff88003f9cd240(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 0000000000000030 CR3: 000000003d847000 CR4: 00000000000026e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
Process mincore01 (pid: 8187, threadinfo ffff8800099a4000, task
ffff88003d00c240)
Stack:
 0000000232482540 ffff880032482540 ffff88003ecb5cc0 ffff88002a8b8000
```

```
ffff880009990000 0000000000000002 ffff88000a89e000 ffff88002a8b8000
0000000000000002 000000000000106f ffff8800099a5c48 ffffffffa06319b4
Call Trace:
 [<ffffffffa06319b4>] ext4_mb_free_blocks+0x36d/0x5dd [ext4]
 [<ffffffffa0615475>] ext4_free_blocks+0x7b/0xcf [ext4]
 [<ffffffffa061b4e7>] ext4_clear_blocks+0xe8/0xf4 [ext4]
 [<ffffffffa061b5a3>] ext4_free_data+0xb0/0x103 [ext4]
 [<ffffffffa061b91f>] ext4_truncate+0x175/0x4d4 [ext4]
 [<ffffffffa062c9db>] ? __ext4_journal_dirty_metadata+0x1f/0x48 [ext4]
 [<ffffffffa0618cc7>] ? ext4_mark_iloc_dirty+0x454/0x4da [ext4]
 [<ffffffffa06193f7>] ? ext4_mark_inode_dirty+0x181/0x196 [ext4]
 [<ffffffffa061dd21>] ext4_delete_inode+0x109/0x1cc [ext4]
 [<ffffffffa061dc18>] ? ext4_delete_inode+0x0/0x1cc [ext4]
 [<ffffffff810c3d8d>] generic_delete_inode+0xc7/0x147
 [<ffffffff810c3e22>] generic_drop_inode+0x15/0x171
 [<ffffffff810c34fd>] iput+0x61/0x65
 [<ffffffff810bccc5>] do_unlinkat+0xfc/0x173
 [<ffffffff81075831>] ? audit_syscall_entry+0x141/0x17c
 [<ffffffff810bcd4d>] sys_unlink+0x11/0x13
 [<ffffffff8100bfaa>] system_call_fastpath+0x16/0x1b
Code: 08 48 89 45 d0 48 83 7e 10 00 75 04 0f 0b eb fe 48 83 3e 00 75
04 0f 0b eb fe 48 8b 3d 36 8c 01 00 be 50 00 00 00 e8 5b bd a7 e0 <44>
89 78 30 4c 89 70 28 49 89 c5 8b 55 b4 89 50 34 48 8b 55 b8
RIP [<ffffffffa063137c>] ext4_mb_free_metadata+0x6b/0x336 [ext4]
 RSP <ffff8800099a5b08>
CR2: 0000000000000030
---[ end trace e8fc382609867b05 ]---
```

341

从内核日志中可以看出两点。

1. 从“FAULT_INJECTION: forcing a failure”之后的调用跟踪可以看出ext4_mb_free_metadata()函数内的slab分配kmem_cache_alloc()函数由于错误注入而失败了。

2. 从“BUG: unable to handle kernel NULL pointer derefrence at 0000000000000030”之后的调用跟踪可以看出，ext4_mb_free_metadata()函数内发生了NULL指针访问，导致Oops发生。

因此可以推测，ext4_mb_free_metadata()函数的slab分配的错误处理中有bug，导

致了 Oops 发生。

看看 ext4_mb_free_metadata()函数的源代码，原因就一目了然了。

```
[fs/ext4/mballoc.c]
static noinline_for_stack int
ext4_mb_free_metadata(handle_t *handle, struct ext4_buddy *e4b,
                      ext4_group_t group, ext4_grpblk_t block, int count)
{
        struct ext4_group_info *db = e4b->bd_info;
        struct super_block *sb = e4b->bd_sb;
        struct ext4_sb_info *sbi = EXT4_SB(sb);
        struct ext4_free_data *entry, *new_entry;
        struct rb_node **n = &db->bb_free_root.rb_node, *node;
        struct rb_node *parent = NULL, *new_node;

        BUG_ON(e4b->bd_bitmap_page == NULL);
        BUG_ON(e4b->bd_buddy_page == NULL);

        new_entry = kmem_cache_alloc(ext4_free_ext_cachep, GFP_NOFS);
        new_entry->start_blk = block;
        new_entry->group  = group;
        new_entry->count = count;
        new_entry->t_tid = handle->h_transaction->t_tid;
        new_node = &new_entry->node;
...
```

342

原因是 kmem_cache_alloc()函数进行 slab 分配时忘了写错误检查。

我向 linux-ext4 邮件列表中发送了名为"[PATCH] ext4: fix unhandled ext4_free_data allocation failure"的补丁，但本书执笔之时，修改方法仍未确定。

总结

本 hack 以 failslab 为例，介绍了利用错误注入技术发现 Linux 内核的潜在 bug 的步骤。

参考文献

- Linux Test Project
  http://ltp.sourceforge.net

——美田晃伸

Linux 内核的 init 节

本 hack 介绍问题分析时与内核中的 section（特别是 init section）有关的内容。

问题概要

LKML（Linux Kernel Mailing List）中出现了 panic 报告。报告内容为，升级成开发中的内核之后，启动时发生了 panic。下面就是 LKML 中报告的 panic 发生时的内核信息。

```
calling tcp_congestion_default+0x0/0x12 @ 1
initcall tcp_congestion_default+0x0/0x12 returned 0 after 2 usecs
Freeing unused kernel memory: 448k freed
Write protecting the kernel read-only data: 4816k
int3: 0000 [#1] SMP
last sysfs file:
CPU 2
Modules linked in:
Pid: 0, comm: events/0 Not tainted 2.6.27-next-20081023 #1
RIP: 0010:[<ffffffff8078ba2b>] [<ffffffff8078ba2b>] nmi_cpu_busy+0x1/0x15
RSP: 0018:ffff88017faa7f80 EFLAGS: 00000086
RAX: 00000000ffffffff RBX: ffff88027f60e000 RCX: ffff88017fa98000
RDX: ffffffff807eb480 RSI: 0000000000000000 RDI: ffffffff807b9e5c
RBP: ffff88017faa7f98 R08: 0000000000000000 R09: ffff88002802c768
R10: 0000000000000000 R11: ffff88027e023e90 R12: 0000000000000002
R13: 0000000000000000 R14: 0000000000000000 R15: 0000000000000000
FS: 0000000000000000(0000) GS:ffff88017fa32280(0000) knlGS:0000000000000000
CS: 0010 DS: 0018 ES: 0018 CR0: 000000008005003b
CR2: 0000000000000000 CR3: 0000000000201000 CR4: 00000000000006e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
```

343

```
Process events/0 (pid: 0, threadinfo ffff88017fa8c000, task ffff88017fa98000)
Stack:
 ffffffff80257afe ffffffff8076d938 0000000000000000 ffff88017faa7fa8
 ffffffff8021f1b0 ffff88017fa8de50 ffffffff8020cabb ffff88017fa8de50 <EOI>
 ffff88017fa8ded8 ffff88027e023e90 0000000000000000 ffff88002802c768
Call Trace:
 <IRQ> <0> [<ffffffff80257afe>] ?
generic_smp_call_function_interrupt+0x35/0xd7
 [<ffffffff8021f1b0>] smp_call_function_interrupt+0x1f/0x2f
 [<ffffffff8020cabb>] call_function_interrupt+0x6b/0x70
 <EOI> <0> [<ffffffff80212659>] ? default_idle+0x2b/0x40
 [<ffffffff8021287d>] ? c1e_idle+0xe5/0xec
 [<ffffffff8057072f>] ? atomic_notifier_call_chain+0xf/0x11
 [<ffffffff8020ad1d>] ? cpu_idle+0x48/0x66
 [<ffffffff80568784>] ? start_secondary+0x177/0x17c
Code: cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc <cc> cc cc cc cc cc
cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
RIP [<ffffffff8078ba2b>] nmi_cpu_busy+0x1/0x15
 RSP <ffff88017faa7f80>
Kernel panic - not syncing: Fatal exception in interrupt
```

344

回归

一般来说，升级到特定版本的内核，特别是开发中的内核时，本来正常工作的部分无法正常工作，这种现象称为回归（regression）。如果知道正常工作的内核版本，并且回归 bug 也容易复现，那直接 git-bisect 可能是最快的解决方法。但是，git-bisect 尽管简单，却十分麻烦。

这里试着采用其他方法解决。

查看日志详细内容

从下面的信息中可知执行了 int3 指令。

```
int3: 0000 [#1] SMP
```

int3 是产生调试中断 INT3 的指令，正常的处理中不可能写这条指令。

接下来查看一下指令指针 RIP。

```
RIP: 0010:[<ffffffff8078ba2b>] [<ffffffff8078ba2b>] nmi_cpu_busy+0x1/0x15
```

可知 int3 指令为函数 nmi_cpu_busy()执行的。

为何 nmi_cpu_busy()函数会执行 int3 指令呢？查看函数 nmi_cpu_busy()后发现，其声明如下。

```
static __int void nmi_cpu_busy(void *data)
```

这里要注意__init 关键字。这个关键字表示该函数要定位到 init section 中。

init section

Linux 内核利用 ELF 的 section 进行定位，下面介绍一下 init section。在 Linux 内核中，初始化代码和数据都放在 init section 中，内核首先启动这些初始化代码和数据，初始化结束之后，就不再需要这些代码和数据了。因此，初始化之后 init section 就会被释放，作为空闲内存继续使用。某些内核配置中，init section 可能达到几百 KB。

345

根据问题原因缩小调查范围

再次查看一下问题日志。日志开头的信息表明 init section 已被释放。

```
Freeing unused kernel memory: 448k freed
```

也就是说，有问题的 nmi_cpu_busy()在初始化完成后被调用了。但是，由于 nmi_cpu_busy()代码中有__init，因此被放到了 init section 中。

因此，问题原因就是已释放的 init section 中的代码被执行了。

在 x86 等部分架构中，为了便于发现访问已释放内存的 bug，会在已释放的内存中填入特殊的字节序列。这些字节序列称为 POISON_FREE_INITMEM，内容为 0xcc。

继续调查发现，SMP 中的跨处理器函数调用，调用了 nmi_cpu_busy()函数。

```
smp_call_function(nmi_cpu_busy, (void *)&endflag, 0);
```

接下来查看有关 SMP 跨处理器函数调用进行相关修改时的发现，本问题的原因

出在 kernel/smp.c 的补丁中。有问题的补丁在 SMP 跨处理器函数调用的判断语句中检查了无用的值，使得目标函数 nmi_cpu_busy()的调用被推迟了。将该问题报告给补丁作者之后，问题就被改正了。

总结

本 hack 介绍了与 init section 有关的问题。

通常，回归问题可以用 git-bisect 解决，但由于本问题错误地调用了 init section 上的函数 nmi_cpu_busy()，调查函数调用的位置，要比单纯使用 git-bisect 能更快地发现引发问题的改动。

参考

- Intel® 64 and IA-32 Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm
- Linux Kernel Mailing List（LKML）
  http://lkml.org/lkmn/2008/10/23/322

346

——岛本裕志

解决性能问题

利用 oprofile 进行性能调查和调优。

如果应用程序的性能无法达到预期，就要进行性能调查和调优。

本 hack 介绍利用 Linux 环境中的标准工具 oprofile 进行性能调查的方法。

oprofile 的使用方法——从初始化到评测

使用 oprofile 进行性能调查和调优的过程大致如下所示。

① oprofile 初始化。
② 设置要评测的事件。
③ 启动 oprofile 守护进程。
④ 评测应用程序。

⑤ 分析结果，解决问题。

下面分别看看各个步骤。

① oprofile 初始化

```
$ sudo opcontrol --init
```

② 设置要评测的事件

下面设置要评测的事件。各硬件架构下的默认设置并不相同，Intel 架构的情况请参见 Intel 的手册。最新版手册只有英文，但认真阅读，就能吸取工程师所需的最基本的营养。

可以评测的事件如下所示。各事件的含义请参见手册。

347

```
$ sudo opcontrol --list-events
oprofile: available events for CPU type "P4 / Xeon with 2 hyper-threads"

See Intel Architecture Developer's Manual Volume 3, Appendix A and
Intel Architecture Optimization Reference Manual (730795-001)

GLOBAL_POWER_EVENTS: (counter: 0)
   time during which processor is not stopped (min count: 6000)
   Unit masks (default 0x1)
   ----------
   0x01: mandatory
BRANCH_RETIRED: (counter: 3)
   retired branches (min count: 6000)
   Unit masks (default 0xc)
   ----------
   0x01: branch not-taken predicted
   0x02: branch not-taken mispredicted
   0x04: branch taken predicted
   0x08: branch taken mispredicted
...
```

③ 启动 oprofile 守护进程

这一步只是启动了守护进程，而评测尚未开始。

```
$ sudo opcontrol --start-daemon
Using 2.6+ OProfile kernel interface.
```

```
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
```

如果存在上次评测的数据，用下面的命令删除之。

```
$ sudo opcontrol --reset
Signalling daemon... done
```

④ 评测应用程序

流程为 oprofile 评测开始（opcontrol --start）、应用程序执行、oprofile 评测停止（opcontrol --stop）。

```
$ sudo opcontrol --start --no-vmlinux
Profiler running.
$ time make test-all
$ sudo opcontrol -setop
Stopping profiling.
```

348

⑤ 分析结果，解决问题

下面显示 oprofile 的评测数据并调查原因。

```
$ sudo opreport -l|head -10
warning: /no-vmlinux could not be found.
CPU: P4 / Xeon, speed 2400 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit
mask of 0x100 (read 2nd level cache miss) count 3000
samples      %   image name         app name         symbol name
67373   34.8114  no-vmlinux         no-vmlinux       (no symbols)
22364   11.5554  ruby               ruby             gc_mark
15421    7.9680  ruby               ruby             garbage_collect
10015   5.1747   ruby               ruby             st_foreach
10004    5.1690  ruby               ruby             gc_mark_children
9953    5.1427   libc-2.8.90.so     libc-2.8.90.so   free
8457    4.3697   ruby               ruby             iseq_mark
```

本例执行了 ruby 附带的测试（make test-all）。评测事件为 L2 缓存未命中（事件名称为 BSQ_CACHE_REFERENCE）。

从上述报告来看（opreport -l），vmlinux 中大约发生了 34.81%的事件。vmlinux 是 Linux 内核，但由于没有调试信息，所以符号名（symbol name）都是(no symbols)。

后面 L2 缓存未命中发生较多的有 ruby 的 gc_mark()，约 11.56%；garbage_collect()，约 7.97%。

我们来看看详细信息。

```
$ sudo opreport -d
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit
mask of 0x100 (read 2nd level cache miss) count 3000
warning: some functions compiled without debug information may have incorrect source line
attributions
vma     samples %   linenr info      image name       app name   symbol name
00000000 67373 34.8114 (no location information)    no-vmlinux no-vmlinux
(no symbols)
 c0102020 1    0.0015 (no location information)
...
08064790 22364 11.5554 gc.c:1273    ruby             ruby
gc_mark
 08064790 45   0.2012 gc.c:1273
 08064791 40   0.1789 gc.c:1273
 08064796 51   0.2280 (no location information)
 08064799 7    0.0313 gc.c:1273
 0806479c 19   0.0850 gc.c:1273
 080647a2 44   0.1967 gc.c:1273
 080647a8 37   0.1654 gc.c:1273
 080647ac 58   0.2593 (no location information)
 080647b2 18   0.0805 (no location information)
 080647b4 10   0.0447 gc.c:1295
 080647b7 43   0.1923 gc.c:1295
 080647bf 1    0.0045 gc.c:1295
 080647c1 16   0.0715 gc.c:1295
 080647c2 3    0.0134 gc.c:1295
 080647c8 4    0.0179 gc.c:1278
 080647ca 21697    97.0175 gc.c:1278        ←①
 080647cc 29   0.1297 gc.c:1278
 080647ce 39   0.1744 gc.c:1279
 080647d0 6    0.0268 gc.c:1279
 080647d2 20   0.0894 gc.c:1280
 080647d5 169  0.7557 gc.c:1282
 080647db 1    0.0045 gc.c:1280
```

```
080647df 1    0.0045 gc.c:1282
0806480b 2    0.0089 gc.c:1282
08064840 2    0.0089 gc.c:1294
08064845 1    0.0045 gc.c:1295
0806484b 1    0.0045 gc.c:1295
08064b10 15421    7.9680 gc.c:1919          ruby       ruby       garbage_collect
08064c29 1      0.0065 gc.c:1957
08064c41 1      0.0065 gc.c:1961
08064c4a 1      0.0065 gc.c:1962
08064c4c 3      0.0195 gc.c:1962
08064c6b 1      0.0065 (no location information)
08064cb2 1      0.0065 gc.c:1975
08064de9 1      0.0065 (no location information)
08064df0 1      0.0065 (no location information)
08064e09 1      0.0065 (no location information)
08064ebf 8      0.0519 (no location information)
08064ed8 170     1.1024 (no location information)
08064eda 162     1.0505 (no location information)
08064ee0 134     0.8689 (no location information)
08064ee6 13      0.0843 (no location information)
08064eec 73      0.4734 (no location information)
08064eef 24      0.1556 (no location information)
08064ef2 3       0.0195 (no location information)
08064ef5 102     0.6614 (no location information)
08064ef8 9       0.0584 (no location information)
08064efe 105     0.6809 (no location information)
08064f00 41      0.2659 (no location information)
08064f02 13205   85.6300 (no location information)   ←②
08064f05 305     1.9778 (no location information)
08064f07 194     1.2580 (no location information)
08064f09 70      0.4539 (no location information)
08064f0c 11      0.0713 (no location information)
08064f0f 305     1.9778 (no location information)
08064f11 36      0.2334 (no location information)
08064f14 2       0.0130 (no location information)
08064f16 61      0.3956 (no location information)
08064f19 27      0.1751 (no location information)
08064f20 234     1.5174 (no location information)
08064f23 38      0.2464 (no location information)
```

350

```
08064f25 1      0.0065  (no location information)
08064f2b 8      0.0519  (no location information)
08064f59 1      0.0065  (no location information)
08064f87 1      0.0065  (no location information)
08064f8e 1      0.0065  (no location information)
08064faa 4      0.0259  (no location information)
08064fc2 1      0.0065  (no location information)
08064fde 2      0.0130  (no location information)
08065010 1      0.0065  (no location information)
08065019 2      0.0130  (no location information)
...
```

我们来分析一下。我们不管 vmlinux 的部分，先来看看 ruby 实现中的内容。

opreport -d 的显示格式如下。

351

格式：地址 事件发生次数 在函数内发生的比例（100%）发生地点

gc_mark()的地址（0x080647ca）上缓存未命中发生较多（21697 次），大约有 97.02%位于 gc_mark()内，还可以知道发生位置（gc.c 文件的 1278 行）①。

来看看源代码（1278 行）。可知是 obj->as.basic.flags 发生了缓存未命中。

```
1271 static void
1272 gc_mark(rb_objspace_t *objspace, VALUE ptr, int lev)
1273 {
1274    register RVALUE *obj;
1275
1276    obj = RANY(ptr);
1277    if (rb_special_const_p(ptr)) return; /* special const not marked */
1278    if (obj->as.basic.flags == 0) return;      /* free cell */
1279    if (obj->as.basic.flags & FL_MARK) return;  /* already marked */
1280    obj->as.basic.flags |= FL_MARK;
1281
```

接下来，garbage_collect()内的地址（0x08064f02）发生了 13205 次（约 85.63%）缓存未命中，但没有源代码信息（no location information）②。很有可能是内联展开到了调用函数中。

```
1917 static int
1918 garbage_collect(rb_objspace_t *objspace)
```

```
1919 {
...
1974    /* gc_mark objects whose marking are not completed*/
1975    while (!MARK_STACK_EMPTY) {
1976        if (mark_stack_overflow) {
1977            gc_mark_all(objspace);
1978        }
1979        else {
1980            gc_mark_rest(objspace);
1981        }
1982    }
```

来看看 gc_mark_all()。

352

```
1089 gc_mark_all(rb_objspace_t *objspace)
1090 {
1091    RVALUE *p, *pend;
1092    size_t i;
1093
1094    init_mark_stack(objspace);
1095    for (i = 0; i < heaps_used; i++) {
1096        p = heaps[i].slot; pend = p + heaps[i].limit;
1097        while (p < pend) {
1098            if ((p->as.basic.flags & FL_MARK) &&
1099                (p->as.basic.flags != FL_MARK)) {
1100                gc_mark_children(objspace, (VALUE)p, 0);
1101            }
1102            p++;
1103        }
1104    }
1105 }
```

用 objdump 反汇编试试看[注4]。

```
$ objdump -CxS ruby
...
        p = heaps[i].slot; pend = p + heaps[i].limit;
        while (p < pend) {
 8064efb:       39 5d 84                cmp    %ebx,-0x7c(%ebp)
```

注 4：objdump 命令可以显示目标文件中的各种信息。

```
8064efe:     76 25              jbe   8064f25 <garbage_collect+0x415>
        if (!(p->as.basic.flags & FL_MARK)) {
8064f00:     8b 13              mov   (%ebx),%edx
8064f02:     f6 c2 20           test  $0x20,%dl
8064f05:     74 d1              je    8064ed8 <garbage_collect+0x3c8>
          else {
             add_freelist(objspace, p);
             free_num++;
          }
        }
```

这样就能看出何处发生了较多的缓存未命中[注5]。

353

减少缓存未命中的方法

由于主内存要比 CPU 的速度慢，因此有效利用缓存对于提高性能是不可或缺的。所谓缓存未命中，就是某次访问无法在缓存中满足，必须访问主内存。缓存未命中会降低性能，因此减少缓存未命中是非常重要的。

缓存未命中的发生原因有①第 1 次访问、②缓存容量不足、③缓存线冲突等，每种原因都有解决方法。

关于第 1 次访问，由于以前没有访问过，当然会发生缓存未命中。解决方法就是在正式访问之前预先访问一遍，使之加载到缓存中，也称为预读取（pre-fetch）技术。

第 2 种情况发生在程序运行范围大于缓存范围的情况下。例如，要复制 8MB 内存，假设缓存大小为 2KB，那么要复制的内容就无法放入缓存，因此需要在缓存中反复执行保存、释放处理，导致额外的开销。

第 3 种情况是，每块内存都会对应到某块缓存上，某些情况下两块内存会保存到同一缓存中，发生冲突（conflict）。

注 5：关于缓存和缓存未命中。缓存位于主内存和 CPU 寄存器之间，用于解决主内存的访问速度和 CPU 寄存器的访问速度之差，可以说是访问速度更快的小型内存。大多数程序的访问模式都有局部性，用快速的小型缓存可以提高系统性能。

例如，假设缓存有 8KB，那么内存空间除以 8KB 的余数相等时，就会保存到同一缓存线上。

```
int a[2048], b[2048];
for(i=0;i<n;i++)
 b[i] = a[i];
```

a[]的地址和 b[]的地址相隔 8KB，用 8KB 除的余数相等，因此上例中就会频繁发生缓存未命中。可以像下面这样按照缓存线的大小错开，使之余数不同，就能减少缓存未命中。

```
#define CACHE_LINE_SIZE 128;  /* 缓存线的大小 */
int a[2048];
char padding[CACHE_LINE_SIZE]; /* 按照缓存线的大小错开 */
int b[2048];
for(i=0;i<n;i++)
 b[i] = a[i];
```

至于本 hack 中的缓存未命中，查看源代码后发现，发生原因就是第 1 次访问。因此写了个预读补丁，补丁内容如下。

354

```
$ svn diff
Index: gc.c
===================================================================
--- gc.c (revision 21331)
+++ gc.c (working copy)
@@ -1095,6 +1095,11 @@
    for (i = 0; i < heaps_used; i++) {
      p = heaps[i].slot; pend = p + heaps[i].limit;
      while (p < pend) {
+         if ( (p+1) < pend) {
+           __asm__ __volatile__ (
+             " prefetch (%0)\n"
+             : : "r" ((p+1)->as.basic.flags) );
+         }
          if ((p->as.basic.flags & FL_MARK) &&
              (p->as.basic.flags != FL_MARK)) {
            gc_mark_children(objspace, (VALUE)p, 0);
@@ -1657,6 +1662,11 @@
```

```
    p = heaps[i].slot; pend = p + heaps[i].limit;
    while (p < pend) {
+       if ( (p+1) < pend) {
+         __asm__ __volatile__ (
+           " prefetch (%0)\n"
+           : : "r" ((p+1)->as.basic.flags) );
+       }
      if (!(p->as.basic.flags & FL_MARK)) {
        if (p->as.basic.flags &&
          ((deferred = obj_free(objspace, (VALUE)p)) ||
```

应用补丁后执行同样的测试。

```
CPU: P4 / Xeon, speed 2400 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit
mask of 0x100 (read 2nd level cache miss) count 3000
samples  %        image name       app name         symbol name
61186    33.2690  no-vmlinux       no-vmlinux       (no symbols)
22716    12.3515  ruby             ruby             gc_mark
11678    6.3497   ruby             ruby             garbage_collect
10014    5.4450   ruby             ruby             gc_mark_children
```

355

可见，garbage_collect()的缓存未命中数由15421次减少到了11678次。

总结

本hack以减少ruby实现中的缓存未命中为例，具体地介绍了利用oprofile进行性能调查和调优的方法。

——吉冈弘隆

利用VMware Vprobe获取信息

可以利用VMware Workstation 6.5以后的版本的VProbe功能查看虚拟机操作系统的状态。

VProbe是个分析、调试引擎，它运行于Hypervisor层，能获取有关虚拟机和物理硬件之间的交互信息。它能获取虚拟CPU寄存器、硬件虚拟状态、虚拟操作系统页面错误、中断情况等各种信息。

启用 Vprobe 的步骤

1. 在 VMware 的设置文件（config）中添加下面这一行。

```
vprobe.allow = TRUE

Linux下：/etc/vmware.config
Windows下：C:\Documents and Settings\All Users\Application Data\VMware\VMware Workstation\
config.ini
```

2. 在虚拟机的设置文件（.vmx 文件）中添加下面这行。

```
vprobe.enable = TRUE
```

使用 Vprobe 功能的注意事项

使用 Vprobe 需要启动虚拟机操作系统。

确认 Vprobe 功能的状态

通过安装 VMware Workstation 时附带的 vmrun 命令，可以查看 Vprobe 功能是否启用。

```
$ ./vmrun vprobeVersion 'vmx文件路径(下面假设为Linux.vmx)'
VProbes version: 0.2 (enabled)
```

如上所示，显示 enabled 表示可以使用。

356

测试 Vprobe 功能

执行下面的命令即可进行测试。

```
$ ./vmrun vprobeLoad 'Linux.vmx' '(vprobe VMM1Hz (printf "hello!\n"))'
```

这样在.vmx 的同一目录下会生成 vprobe.out，并且每秒钟输出一次 hello!。

上面的含义是用 VMM1Hz 这个 Vprobe 预定义的静态侦测器（Static probe）输出“hello!”字符串和换行。VMM1Hz 是个每秒发生一次的静态侦测器。虚拟机运行期间，vprobe.out 中就会输出字符串。

停止输出，停止 Vprobe 功能

要停止 Vprobe 功能并结束输出，可以使用以下命令。

```
$ ./vmrun vprobeReset 'Linux.vmx'
```

Vprobe 功能的使用例：显示 boot 设备

下面的例子演示了在特定地址的指令执行时执行动态侦测器（Dynamic probe）。

1. 创建下面的文件。；（分号）之后为注释，实际没有必要输入。

```
$ cat printboot.emt
; Print the boot device.
(defstring device)          ; 定义字符串变量
(definteger dl)             ; 定义数值变量
(vprobe GUEST:0x7c00        ; 在bootloader开始地址处执行侦测器
(setint dl (&RDX 0xff))     ; 获取RDX寄存器低8比特，即DL寄存器内容
(cond ((==dl 0x80)          ; 条件判断
(setstr device "hard drive"))
((== dl 0)
(setstr device "floppy drive"))
(1
(setstr device "unknown device CD etc.")))
(printf "Booting from %s (0x%x)\n" device dl))
```

357

2. 启动虚拟机

Vprobe 的代码必须在虚拟机执行时才能读取，因此要先启动虚拟机，并用 F2 等键使之停在 BIOS 设置画面上。

3. 读取 Vprobe

在宿主上执行下述命令。

```
$ ./vmrun vprobeLoad 'Linux.vmx' "`cat printboot.emt`"
```

脚本从命令行读入，因此必须用双引号和反引号进行转义。

4. 测试

结束虚拟机操作系统的 BIOS 设置界面，重新启动，vprobe.out 上就会显示启动设备的信息，如：

```
Booting from hard drive (0x80)
Booting from floppy drive (0x0)
Booting from unknown device CD etc. (0x9f)
```

用函数名指定地址

虚拟机的地址信息可以用下述方法导出成符号文件。

```
# cat /proc/kallsyms > kallsyms.txt
```

将上面获得的符号文件设置到 vmx 文件中，就可以用内核函数名称来指定地址。

```
vprobe.guestSyms = "kallsyms.txt"
```

符号文件的扩展名要设置为.txt，或没有扩展名。

使用符号文件的例子

1.

```
$ cat system_call.emt
(vprobe GUEST:system_call
(printf "Current RAX : 0x%016x RSP : 0x%016x \n" RAX RSP))
```

358

2. 读入 Vprobe

```
$ ./vmrun vprobeLoad 'Linux.vmx' "`cat system_call.emt`"
```

3. 在内核的 system_call 函数被调用的那一刻，vprobe.out 中就会输出信息。此时 RAX（EAX）寄存器中会包含系统调用号码，可以判断调用了哪个系统调用。

[例]

```
Current RAX : 0x0000000000000001 RSP : 0x00000000cee65fe4
Current RAX : 0x00000000000000af RSP : 0x00000000cb2f1fe4
```

编号和系统调用的对应关系可以参见内核源代码中的下述文件。

```
v2.6.23之前
include/asm-i386/unistd.h
include/asm-x86_64/unistd.h

v2.6.24
```

```
include/asm-x86/unistd_32.h
include/asm-x86/unistd_64.h
```

这样在调用函数时即可获取寄存器信息，十分方便。寄存器的详细情况参见**“HACK#8 Intel 架构的基本知识”**。函数调用请参见**“HACK#10 函数调用时的参数传递方法（x86_64 篇）”**和**“HACK#11 函数调用时的参数传递方法（i386 篇）”**。

如果宿主操作系统是 Windows，从标准的命令行提示符下执行多条 Vprobe 脚本就比较困难。此时可以使用 Cygwin 的控制台，并用与 Linux 相同的方法指定脚本文件，即可执行。

总结

使用 VMware 的功能可以动态地了解内部状态。此外，下面的参考资料中给出了 Vprobe 能获取的信息及静态侦测器的种类的参考手册，可以作为参考。

359

参考资料

- Vprobes Programming Reference
  http://www.vmware.com/pdf/ws65_vprobes_reference.pdf

——吉田俊辅

用 Xen 获取内存转储

本 hack 介绍获取在 Xen 虚拟机（Domain-U、HVM Domain）上运行的 Linux 的内存转储的方法。

如果在 Xen 虚拟机上运行 Linux，就可以利用 Xen 的控制台（Domain-0）获取虚拟机的内存转储，而且还能 livedump，无须停止虚拟机运行，就能获取内存转储。

不论是准虚拟化（Paravirtualization）的 Domain-U 虚拟机，还是完全虚拟化（FullVirtualization）的 HVM Domain 虚拟机，都能进行内存转储。

步骤

首先登录到 Xen 的管理操作系统的 Domain-0。

用 xm list 命令查看运行中的 domain（虚拟机）。

```
# xm list
Name                ID  Mem  VCPUs   State  Time(s)
Asianux3GA_HV        1  2048     1  r-----  580.1
Domain-0             0   668     8  r-----  173.0
```

用 xm dump-core 命令输出 dump 文件。指定--live 选项，无须停止目标 domain，就能获取它的内存转储。

[例]

```
# xm dump-core --live Asianux3GA_HV /home/user/axhvmstall.live
Dumping core of domain: Asianux3GA_HV ...
```

输出的 dump 文件大小等于运行中的虚拟机的内存大小，因此要注意保存 dump 文件分区的空闲容量。

用 crash 命令即可分析输出的 dump 文件。

360

```
# crash System.map-2.6.18-xxx vmlinux axhvmstall.live
[中略]
      SYSTEM MAP:  System.map-2.6.18-xxx
  DEBUG KERNEL:  vmlinux (2.6.18-xxx)
      DUMPFILE:  dump/axhvmstall.live
          CPUS:  1
          DATE:  Sun Sep 28 11:14:22 2008
        UPTIME:  00:53:02
  LOAD AVERAGE:  0.00, 0.00, 0.00
         TASKS:  50
      NODENAME:  axs3fullovm.miraclelinux.com
       RELEASE:  2.6.18-xxx
       VERSION:  #1 SMP Sun Mar 16 20:22:54 EDT 2008
       MACHINE:  i686  (2660 Mhz)
        MEMORY:  2 GB
         PANIC:  ""
           PID:  0
       COMMAND:  "swapper"
          TASK:  c0664bc0  [THREAD_INFO: c06d9000]
           CPU:  0
```

```
        STATE:    TASK_RUNNING
      WARNING:    panic task not found
```

指定--crash选项，可以在获取内存转储之后停止domain。

[例]

```
# xm dump-core --crash Asianux3GA_HV /home/user/axhvmstall.crash
Dumping core of domain: Asianux3GA_HV ...
```

Xen的其他信息

可以参考《Xen彻底入门》（翔泳社出版，ISBN：978-4-7981-1447-7）。

——吉田俊辅

361

理解用GOT/PLT调用函数的原理

本hack介绍通过GOT/PLT调用函数的原理，这是通过汇编代码调查程序执行流程时必不可少的知识。

程序和共享库

许多程序为了减少可执行文件的大小，使用了共享库。共享库内的代码部分和数据部分在使用该库的程序执行时映射到该程序的虚拟内存空间内。下面来具体看一看。

我们来编译下述源代码，它使用了库函数rand()。

[test1.c]

```
#include <stdio.h>
#include <stdlib.h>

int func1(void)
{
   return 1;
}

int main(void)
{
```

```
    int a, b;
    a = func1();
    b = rand();
    return a + b;
}
```

为了查看虚拟空间的内存映射，我们用 GDB 将该程序暂停。

```
$ gdb test1
...
(gdb) start
Breakpoint 1 at 0x400487
Starting program: /root/tmp/test1
0x0000000000400487 in main ()
```

362

该状态下，从其他终端输入以下内容，即可显示 test1 的虚拟内存。

```
$ cat /proc/`pidof test1`/maps
00400000-00401000 r-xp 00000000 fd:00 166905             /root/tmp/test1
00600000-00601000 rw-p 00000000 fd:00 166905             /root/tmp/test1
37d7e00000-37d7e1a000 r-xp 00000000 fd:00 360453         /lib64/ld-2.5.so
37d8019000-37d801a000 r--p 00019000 fd:00 360453         /lib64/ld-2.5.so
37d801a000-37d801b000 rw-p 0001a000 fd:00 360453         /lib64/ld-2.5.so
37d8200000-37d8344000 r-xp 00000000 fd:00 360460         /lib64/libc-2.5.so
37d8344000-37d8544000 ---p 00144000 fd:00 360460         /lib64/libc-2.5.so
37d8544000-37d8548000 r--p 00144000 fd:00 360460         /lib64/libc-2.5.so
37d8548000-37d8549000 rw-p 00148000 fd:00 360460         /lib64/libc-2.5.so
37d8549000-37d854e000 rw-p 37d8549000 00:00 0
2aaaaaaab000-2aaaaaaac000 rw-p 2aaaaaaab000 00:00 0
2aaaaaad7000-2aaaaaad9000 rw-p 2aaaaaad7000 00:00 0
7fff277ee000-7fff27803000 rw-p 7fff277ee000 00:00 0      [stack]
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0  [vdso]
```

可知 test1 使用了 ld-2.5.so 和 libc-2.5.so 两个共享库，ld-2.5.so 主要提供使用共享库所需的函数，因此程序很少明确地调用该共享库提供的函数；而 libc-2.5.so 提供 printf()、malloc()、rand()等 C 语言的主要函数。

现在，libc-2.5.so 被映射到 0x37d8200000 地址上，该地址是在程序运行时被确定的，每次运行都不一样，因此 rand()地址在编译时是不知道的，运行时要调查共享库中的函数地址再调用。GNU/Linux 系统中，该机制用到了 PLT（Procedure

Linkage Table）和 GOT（Global Offset Table）这两个程序中的区域。

简单介绍一下 PLT 和 GOT。GOT 是保存库函数地址的区域，程序运行时，用到的库函数地址会设置到该区域中。PLT 是调用库函数时的小型代码集合，程序可以像调用自己的用户函数一样调用这些小型代码。因此，PLT 中包含的小型代码基本上与用到的库函数数量相同。简单来说，这些代码只是跳转到 GOT 中设置的值而已。如果 GOT 中尚未设置调用函数的地址，就将地址设置到 GOT 中之后再跳转。但是，库函数地址在程序运行过程中不会改变，因此 GOT 的值一旦设置就不会再发生变化。因此，调用库函数时没有必要检查 GOT 是否被设置。glibc 用巧妙的方法在调用时避免了这种无用的检查，方法稍后说明。

363

函数调用

这里利用刚才的 test1 来看看实际上 PLT 和 GOT 是如何使用的。在刚才的 GDB 中输入了 start 命令的状态下，对 main()进行反汇编，即可看到编译结果如下所示。

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400483 <main+0>:   push  %rbp
0x0000000000400484 <main+1>:   mov   %rsp,%rbp
0x0000000000400487 <main+4>:   sub   $0x10,%rsp
0x000000000040048b <main+8>:   callq 0x400478 <func1>
0x0000000000400490 <main+13>:  mov   %eax,0xfffffffffffffff8(%rbp)
0x0000000000400493 <main+16>:  callq 0x4003a8 <rand@plt>
0x0000000000400498 <main+21>:  mov   %eax,0xfffffffffffffffc(%rbp)
0x000000000040049b <main+24>:  mov   0xfffffffffffffffc(%rbp),%eax
0x000000000040049e <main+27>:  add   0xfffffffffffffff8(%rbp),%eax
0x00000000004004a1 <main+30>:  leaveq
0x00000000004004a2 <main+31>:  retq
```

自己的函数 func1()和库函数 rand()的调用方法相同，都是给 call 指令指定相对地址。func1()的函数实体位于 0x400478，如下所示。程序中包含的用户函数与调用者的相对位置关系可以在编译时确定，因此通常使用相对地址的 call 指令直接调用。

```
(gdb) disas func1
0x0000000000400478 <func1+0>:  push  %rbp
0x0000000000400479 <func1+1>:  mov   %rsp,%rbp
0x000000000040047c <func1+4>:  mov   $0x1,%eax
```

```
0x0000000000400481 <func1+9>:  leaveq
0x0000000000400482 <func1+10>: retq
```

364

至于 rand()的调用方法，一眼看上去似乎与 func1()相同，都是采用了指定相对地址的 call 指令。我们来看看被调用的地址 0x4003a8 处的汇编代码。

```
(gdb) disas 0x4003a8
Dump of assembler code for function rand@plt:
0x00000000004003a8 <rand@plt+0>:    jmpq  *2098370(%rip)
                                    # 0x600870 <_GLOBAL_OFFSET_TABLE_+32>
0x00000000004003ae <rand@plt+6>:    pushq $0x1
0x00000000004003b3 <rand@plt+11>:   jmpq  0x400388
```

开头为跳转指令。跳转地址保存在 0x600870。如下，查看该值，为 0x4003ae，也就是说，这就是 jmpq 指令的地址。

```
(gdb) x 0x600870
0x600870 <_GLOBAL_OFFSET_TABLE_+32>:   0x004003ae
```

结果，尽管执行了 0x4003a8 处的 jmpq 指令，也只是跳转到了下一条指令而已。实际上，这是因为 GOT 尚未设置成库函数的地址。0x4003ae 之后的指令将 0x600870 处设置为 rand()函数的地址，设置完成后，0x4003a8 处的 jmpq 指令就可以直接调用 rand()。这就是刚才说过的无须检查 GOT 是否被设置的原理。我们接着看看 0x4003ae 地址后面的指令。push 指令之后跳转到 0x400388，该地址处的代码由于符号信息的关系，只把地址作为反汇编指令的参数是无法反汇编的，因此我们先用 x 命令显示 5 条指令看看。

```
(gdb) disas 0x400388
No function contains specified address.
(gdb) x/5i 0x400388
0x400388: pushq 2098378(%rip)   # 0x600858 <_GLOBAL_OFFSET_TABLE_+8>
0x40038e: jmpq *2098380(%rip)   # 0x600860 <_GLOBAL_OFFSET_TABLE_+16>
...
```

第 2 条指令为跳转指令，可以认为第 3 条以后的指令与此无关，这里就不再介绍了。0x40038e 处的 jmpq 指令调用_dl_runtime_resolve()函数（严格来说它不会返回，不能算做一个函数），如下所示。<_dl_runtime_resolve+61>处调用的_dl_fixup()将 rand()等库函数的地址设置到 GOT 中。后面还有许多处理，有兴趣的可以继续阅读 GLIBC 的代码。

365

此外，到这里一共执行了两次 push 指令，累积在栈上的这些值会作为后续处理的参数使用。_dl_fixup()函数将调用函数的地址返回到 rax 中，_dl_runtime_resolve()将它赋给 r11，最后作为 jmpq 指令的操作数跳转到要调用的函数。

```
(gdb) x 0x600860
0x600860 <_GLOBAL_OFFSET_TABLE_+16>:   0x00000037d7e122a0
(gdb) disas 0x00000037d7e122a0
Dump of assembler code for function _dl_runtime_resolve:
0x00000037d7e122a0 <_dl_runtime_resolve+0>:    sub   $0x38,%rsp
0x00000037d7e122a4 <_dl_runtime_resolve+4>:    mov   %rax,(%rsp)
0x00000037d7e122a8 <_dl_runtime_resolve+8>:    mov   %rcx,0x8(%rsp)
0x00000037d7e122ad <_dl_runtime_resolve+13>:   mov   %rdx,0x10(%rsp)
0x00000037d7e122b2 <_dl_runtime_resolve+18>:   mov   %rsi,0x18(%rsp)
0x00000037d7e122b7 <_dl_runtime_resolve+23>:   mov   %rdi,0x20(%rsp)
0x00000037d7e122bc <_dl_runtime_resolve+28>:   mov   %r8,0x28(%rsp)
0x00000037d7e122c1 <_dl_runtime_resolve+33>:   mov   %r9,0x30(%rsp)
0x00000037d7e122c6 <_dl_runtime_resolve+38>:   mov   0x40(%rsp),%rsi
0x00000037d7e122cb <_dl_runtime_resolve+43>:   mov   %rsi,%r11
0x00000037d7e122ce <_dl_runtime_resolve+46>:   add   %r11,%rsi
0x00000037d7e122d1 <_dl_runtime_resolve+49>:   add   %r11,%rsi
0x00000037d7e122d4 <_dl_runtime_resolve+52>:   shl   $0x3,%rsi
0x00000037d7e122d8 <_dl_runtime_resolve+56>:   mov   0x38(%rsp),%rdi
0x00000037d7e122dd <_dl_runtime_resolve+61>:   callq 0x37d7e0ca50 <_dl_fixup>
0x00000037d7e122e2 <_dl_runtime_resolve+66>:   mov   %rax,%r11
0x00000037d7e122e5 <_dl_runtime_resolve+69>:   mov   0x30(%rsp),%r9
0x00000037d7e122ea <_dl_runtime_resolve+74>:   mov   0x28(%rsp),%r8
0x00000037d7e122ef <_dl_runtime_resolve+79>:   mov   0x20(%rsp),%rdi
0x00000037d7e122f4 <_dl_runtime_resolve+84>:   mov   0x18(%rsp),%rsi
0x00000037d7e122f9 <_dl_runtime_resolve+89>:   mov   0x10(%rsp),%rdx
0x00000037d7e122fe <_dl_runtime_resolve+94>:   mov   0x8(%rsp),%rcx
0x00000037d7e12303 <_dl_runtime_resolve+99>:   mov   (%rsp),%rax
0x00000037d7e12307 <_dl_runtime_resolve+103>:  add   $0x48,%rsp
0x00000037d7e1230b <_dl_runtime_resolve+107>:  jmpq  *%r11
...
```

从 main()的 0x400493 处 rand@plt 被调用，到_dl_runtime_resolve()都是跳转，而也没有使用 call 指令一次。因此，在_dl_runtime_resolve()跳转后，在跳转目的地处执行 ret 指令时，返回到的地址就是 main()函数的下一个地址 0x400498。到此就能明白，尽管实际上经过了十分曲折的路径调用了目标库函数，但在 main 看来，跟调用用户函数没什么区别。

366

查看设置后的 GOT

接下来我们看看_dl_runtime_resolve()中对 GOT 的设置。

```
(gdb) b *0x00000037d7e1230b _dl_runtime_resolve的最后地址
Breakpoint 2 at 0x37d7e1230b
(gdb) c
Continuing.

Breakpoint 2, 0x00000037d7e1230b in _dl_runtime_resolve ()
   from /lib64/ld-linux-x86-64.so.2
(gdb) x 0x600870
0x600870 <_GLOBAL_OFFSET_TABLE_+32>:    0x00000037d8233a70
(gdb) x/i 0x00000037d8233a70
0x37d8233a70 <rand>:    sub    $0x8,%rsp
```

我们将断点设在了_dl_runtime_resolve()的最后那条 jmpq 指令上，运行到该处后，查看 rand@plt 跳转到的地址（0x600870 中保存的地址），发现是 0x37d8233a70，正是 rand()函数的起始地址。也就是说，下次调用 rand@plt 时，无须通过_dl_runtime_resolve()，就可以直接跳转到 rand()了。

总结

本 hack 介绍了跟踪程序时频繁遇到的通过 PLT/GOT 调用库函数的原理。

——大和一洋

367

调试 initramfs 镜像

许多发行版在启动时采用了 initramfs，本 hack 就介绍了它的调试方法。

什么是 initramfs

几乎所有发行版都用 initramfs 来挂载 root 文件系统。在这些 Linux 发行版中，像 SCSI 卡、文件系统等挂载 root 文件系统时必需的功能也被编译成内核模块。这是因为使用的磁盘接口（SATA 或 SCSI）、卡的种类、根文件系统的格式（ext3、ext4、xfs、reiserfs 等），各个用户也不尽相同。实现模块化之后，用户可以根据自己系统的需要，仅加载必要的模块，以节省内存消耗。但是，挂载根文件系统

所需的磁盘、文件系统等模块也都放在了根文件系统中。这样在内核启动的最终阶段尝试挂载 root 文件系统时，由于内核内部并没有处理根文件系统的功能，从而导致挂载失败。

解决该问题的方法就是 initramfs。initramfs 是个在 RAM 上创建的文件系统，挂载根文件系统所需的脚本和模块都被解压到这个文件系统中，将这些脚本和模块用 cpio 打包，再用 gzip 压缩。内核在启动的最终阶段执行该文档内的 init，这个 init 中记载了加载磁盘、文件系统等模块的处理，最后实现磁盘上的根文件系统的挂载。

initramfs 出现于内核版本 2.6，在这之前使用 initrd 实现同样功能。两者的区别是 initramfs 是 gzip 压缩的 cpio 镜像，而 initrd 是 ext2 等文件系统的镜像。后者在创建文件系统时大小固定，灵活性不佳，而且内核内部实现 initramfs 也要比 initrd 简单，因此现在 initramfs 成了主流。

调试 initramfs

系统启动时经常发生的问题就是根文件系统无法挂载导致的 kernel panic、运行过程中突然停止等。许多问题的原因就是 initramfs 中没有包含必要的模块和命令等，但大多数发行版的 initramfs 脚本并不会显示详细信息。因此，问题发生时，大多数情况下很难得知究竟发生了什么问题。本 hack 介绍的方法给 initramfs 添加了调试信息，以了解问题发生在什么地方。

368

如前所述，initramfs 是个 gzip 压缩过的 cpio 文件，通常保存在/boot 目录下。例如，Fedora10 为每个内核版本都准备了 initramfs。这些文件名以 initrd 开头（可能是由于历史原因），实际的内容是 initramfs。

```
# ls -l /boot/initrd*
-rw------- 1 root 3.8M Jan  3 00:15 /boot/initrd-2.6.27.5-117.fc10.x86_64.img
-rw------- 1 root 3.9M Jan  3 01:18 /boot/initrd-2.6.27.9-159.fc10.x86_64.img
-rw------- 1 root 3.9M Jan  4 18:44 /boot/initrd-2.6.27.9.img
```

将这些文件解压缩，看看其中的内容。

```
# mkdir work
# cd work
# gunzip -c /boot/initrd-2.6.27.9-159.fc10.x86_64.img | cpio -id
```

```
# ls -l
total 40
drwx------ 2 root root 4096 Jan 25 21:59 bin
drwx------ 3 root root 4096 Jan 25 21:59 dev
drwx------ 5 root root 4096 Jan 25 21:59 etc
-rwx------ 1 root root 1933 Jan 25 21:59 init
drwx------ 6 root root 4096 Jan 25 21:59 lib
drwx------ 2 root root 4096 Jan 25 21:59 lib64
drwx------ 2 root root 4096 Jan 25 21:59 proc
lrwxrwxrwx 1 root root   3 Jan 25 21:59 sbin -> bin
drwx------ 2 root root 4096 Jan 25 21:59 sys
drwx------ 2 root root 4096 Jan 25 21:59 sysroot
drwx------ 4 root root 4096 Jan 25 21:59 usr
# cat init
#!/bin/nash

mount -t proc /proc /proc
setquiet
echo Mounting proc filesystem
echo Mounting sysfs filesystem
mount -t sysfs /sys /sys
echo Creating /dev
mount -o mode=0755 -t tmpfs /dev /dev                    (a)
mkdir /dev/pts
mount -t devpts -o gid=5,mode=620 /dev/pts /dev/pts
mkdir /dev/shm
mkdir /dev/mapper
echo Creating initial device nodes
mknod /dev/null c 1 3
mknod /dev/zero c 1 5
mknod /dev/systty c 4 0
mknod /dev/tty c 5 0                                     (b)
mknod /dev/console c 5 1
<<省略>>
/lib/udev/console_init tty0
daemonize --ignore-missing /bin/plymouthd
plymouth --show-splash
```

369

```
echo Setting up hotplug.
hotplug
echo Creating block device nodes.
mkblkdevs
echo Creating character device nodes.
mkchardevs
echo "Loading scsi_transport_spi module"
modprobe -q scsi_transport_spi
echo "Loading mptbase module"
modprobe -q mptbase
echo "Loading mptscsih module"
modprobe -q mptscsih                                    (c)
echo "Loading mptspi module"
modprobe -q mptspi
echo Making device-mapper control node
mkdmnod
modprobe scsi_wait_scan
rmmod scsi_wait_scan
mkblkdevs
echo Scanning logical volumes
lvm vgscan --ignorelockingfailure
echo Activating logical volumes
370
lvm vgchange -ay --ignorelockingfailure VolGroup00
resume /dev/VolGroup00/LogVol01
echo Creating root device.
mkrootdev -t ext3 -o defaults,ro /dev/VolGroup00/LogVol00
echo Mounting root filesystem.
mount /sysroot                                          (d)
cond -ne 0 plymouth --hide-splash
echo Setting up other filesystems.
setuproot
loadpolicy
plymouth --newroot=/sysroot
echo Switching to new root and running init.
switchroot
echo Booting has failed.
sleep -1
```

可见 Fedora10 中 init 实际上是个脚本，其中包括/proc 的挂载等基本设置(a)、创

建最低限度的设备文件(b)、加载模块(c)、挂载磁盘上的 root 文件系统分区(d)等。其中多处用 echo 命令显示信息，但是为了确定问题所在，有时会希望给每行都加上 echo 命令。接下来在(d)的 loadpolicy 和 plymouth 执行之前加上个 echo message 试试看。

创建调试用的 initramfs

首先编辑解压后的 init。

```
echo Setting up other filesystems.
setuproot
echo Debug: executes loadpolicy ◄──────────── 添加
loadpolicy
echo Debug: executes plymouth   ◄──────────── 添加
plymouth --newroot=/sysroot
echo Switching to new root and running init.
```

接下来用下面的命令重新创建 initramfs。

```
# find | cpio -o -H newc | gzip - -c > /boot/initrd-2.6.27.9-159.fc10.debug.x86_64.img
```

然后将这个 initramfs 设置到 boot loader 的 initrd 项中，以便在系统启动时加载。如果用 grub 作为 boot loader，就将下面的内容添加到/boot/grub/menu.lst（或/etc/grub.conf）中。另外，出于调试的目的，最好不要指定内核选项中的 quiet（禁止向控制台输出信息的选项）、rhgb（Fedora 中显示图形化启动界面的选项）等。

371

```
title Fedora (2.6.27.9-159.fc10.x86_64) : Debug
      root (hd0,0)
      kernel /vmlinuz-2.6.27.9-159.fc10.x86_64 ro root=/dev/VolGroup00/LogVol00
      initrd /initrd-2.6.27.9-159.fc10.debug.x86_64.img——设置成创建的 initramfs
```

用上述内容启动系统，显示如下画面。可见添加的 echo 行已显示了。如果发生错误或停止响应，用这些信息就能确定问题所在。

```
...
Creeating root device.
Mounting root filesystem.
Setting up other filesystem.
Debug: executes loadpolicy
Debug: executes plymouth
```

```
Switching to new root and running init.
```

但是，用上述方法即使确定了问题所在，由于故障导致失去响应的情况下，也无法立即确定问题原因。这种情况下请参考**“HACK#18 使用 SysRq 键调试”**进行调试。

总结

许多发行版在启动时采用了 initramfs，本 hack 介绍了它的调试方法。

——大和一洋

使用 RT Watchdog 检测失去响应的实时进程

本 hack 介绍 Linux 2.6.25 以上版本支持的 RT Watchdog 及其使用方法。

“HACK#40 实时进程停止响应”中也介绍过，实时进程失控会造成重大影响，导致系统整体的响应恶化等。对此 Linux 采取了两种策略。第 1 种是进程调度器（正确来说应该是任务调度器）自身对分配给实时进程的 CPU 时间做出限制。Linux 2.6.23 引入的名为 CFS 的调度器实现了该功能，这个功能以所有实时进程为对象。另一种策略就是本 hack 介绍的 RT Watchdog，该功能扩展了 setrlimit(2)，只针对特定实时进程。从 Linux 2.6.25 开始可以使用 RT Watchdog。

372

什么是 RT Watchdog

RT Watchdog 会检测没有调用阻塞 API（blocking API）而持续使用 CPU 时间的实时进程，如果达到了用户预先设置的限制，就给进程发送信号。用 setrlimit(2) 给 RLIMIT_RTTIME 设置最大 CPU 时间（包括 soft limit 和 hard limit），就能启用 RT Watchdog，最大 CPU 时间以微秒为单位。达到 soft limit 后就给进程发送 SIGXCPU 信号，达到 hard limit 后发送 SIGKILL 信号。

确认 RT Watchdog 的行为

所谓阻塞 API，就是会导致进程进入睡眠或 I/O 处于等待状态的系统调用，比如 read()、write()、sleep()、select()、recv()等系统调用。要注意的是，sched_yield()

系统调用并不是阻塞 API。sched_yield()系统调用会主动释放 CPU，因此不能算做阻塞 API。

我们准备了下面的示例程序，用它来确认 RT Watchdog 的行为。该程序会 fork() 3 个子进程，其行为都稍有不同。各个子进程之间的区别请参考源代码中的注释。

```
$ cat rt-watchdog.c
#include <sched.h>
#include <sys/time.h>
#include <sys/resource.h>
/*
 * 笔者的环境(Fedora9)中RLIMIT_RTTIME不在<sys/resource.h>中，
 * 因此包含了<asm/resource.h>
 */
#include <asm/resource.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

#define USEC_PER_SEC  (1000000UL)
#define loop_10sec(start)  \
 for (start=time(NULL); time(NULL) < (start+10); )

int main(void)
{
 time_t start;
 struct rlimit rl;
 struct sched_param param;

 /* 设置soft limit为1秒，hard limit为4秒 */
 rl.rlim_cur = USEC_PER_SEC;
 rl.rlim_max = USEC_PER_SEC << 2;
 setrlimit(RLIMIT_RTTIME, &rl);

 /* 改变该进程的实时类 */
 param.sched_priority = sched_get_priority_min(SCHED_RR);
```

373

```
sched_setscheduler(0, SCHED_RR, &param);

if ( fork() == 0 ) {
 /*
  * 子进程1在10秒内连续调用sched_yield()系统调用
  */
 printf("%lu: Child-1: PID%d\n", time(NULL), getpid());
 loop_10sec(start);
  sched_yield();

} else if ( fork() == 0 ) {
 /*
  * 子进程2改变接受SIGXCPU信号时的行为，并在10秒内循环
  */
 struct sigaction act;
 memset(&act, 0, sizeof(act));
 act.sa_handler = SIG_IGN;
 sigaction(SIGXCPU, &act, NULL);

 printf("%lu: Child-2: PID%d\n", time(NULL), getpid());
 loop_10sec(start)
  ;

} else if ( fork() == 0 ) {
 /*
  * 子进程3在10秒内连续调用usleep()
  */
 printf("%lu: Child-3: PID%d\n", time(NULL), getpid());
 loop_10sec(start)
  usleep(1);

} else {
 /* 父进程检测子进程的结束状态 */
 int status, i;
 pid_t pid;

 for (i = 0; i < 3; i++) {
  pid = wait(&status);
  if (WIFSIGNALED(status)) {
```

374

```
      if (WTERMSIG(status) == SIGKILL)
        printf("%lu: PID%d is terminated by SIGKILL\n", time(NULL), pid);
      else if (WTERMSIG(status) == SIGXCPU)
        printf("%lu: PID%d is terminated by SIGXCPU\n", time(NULL), pid);
    } else if (WIFEXITED(status))
      printf("%lu: PID%d normally exits\n", time(NULL), pid);
  }
 }

 return 0;
}
```

该程序的执行结果如下所示[注6]。

375

```
# gcc rt-watchdog.c -o rt-watchdog
# ./rt-watchdog
1226852952: Child-1: PID5610
1226852952: Child-2: PID5611
1226852953: Child-3: PID5612
1226852953: PID5610 is terminated by SIGXCPU
1226852956: PID5611 is terminated by SIGKILL
1226852963: PID5612 normally exists
```

子进程 1 是连续调用 sched_yield()的进程。启动后大约 1 秒钟达到 soft limit，收到了 SIGXCPU 信号。子进程 2 是不调用阻塞 API 持续循环的进程。但是，由于设置成忽视 SIGXCPU，因此启动后约 4 秒后达到 hard limit，被 SIGKILL 强行结束。子进程 3 调用了阻塞 API，因此没有被当做失控的进程，没有受到 RT Watchdog 的控制而一直运行到最后。

总结

本 hack 介绍了检测实时进程失控的机制——RT Watchdog 的使用方法和行为。如果要在 Linux 上创建实时应用程序，那么设置好 RT Watchdog，万一应用程序失控时，就能进行恢复工作。

——安部东洋

注 6：该程序将调度策略改成了实时，因此运行时需要超级用户权限。

查看手头的 x86 机器是否支持 64 位模式

本 hack 介绍通过读取处理器信息来判断是否支持 64 位模式的方法。

最近的 x86 系列处理器支持 64 位模式也成了理所当然的事。自己的 PC 机是否支持 64 位模式，拥有者自然是很清楚，但是本 hack 正要介绍确认这一点的方法。

用 proc 文件系统查看

如果运行的操作系统是 Linux，查看/proc/cpuinfo 就能知道是否支持 64 位模式。

```
# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe sy
scall nx lm constant_tsc arch_perfmon pebs bts rep_good nopl pni monit
or ds_cpl est tm2 ssse3 cx16 xtpr lahf_lm
...
```

376

检查 flags 行中有没有 lm 标记。lm 是 Long Mode 的简写，表示支持 64 位模式。

用 CPUID 指令查看

CPUID 指令可以获取处理器信息，用它就能查看处理器支持什么功能。这里写了个程序，用它来查看 CPU 是否支持 64 位模式。该程序只要有 GCC 就能编译，因此 FreeBSD、Cygwin 等环境中也可以使用。

```
$ cat chklm.c
#include <stdio.h>

/* x86-64 Long Mode flag */
#define X86_FEATURE_LM   (1<<29)

void cpuid(int op, unsigned int *eax, unsigned int *ebx,
        unsigned int *ecx, unsigned int *edx)
```

```
{
      __asm__("cpuid"
            : "=a" (*eax),
              "=b" (*ebx),
              "=c" (*ecx),
              "=d" (*edx)
            : "0" (op));
}

int main(void)
{
   unsigned int eax,ebx,ecx,edx;

   /* 检查能否获取扩展功能CPUID信息 */
   cpuid(0x80000000, &eax, &ebx, &ecx, &edx);
   if (eax < 0x80000001)
        goto no_longmode;

   /* 检查Long Mode比特 */
   cpuid(0x80000001, &eax, &ebx, &ecx, &edx);
   if (!(X86_FEATURE_LM & edx))
        goto no_longmode;

   printf("x86_64 Long Mode is supported.\n");
   return 0;
no_longmode:
   printf("x86_64 Long Mode is not supported.\n");
   return 1;
}
```

377

在支持 64 位模式的系统上执行，显示结果如下。

```
$ gcc -o chklm chklm.c
$ ./chklm
x86_64 Long Mode is supported.
```

总结

本书执笔过程中，写作组成员讨论了一个问题，那就是“怎样才能判断自己的机

器的确支持 64 位模式？”，因此打算把这个问题也写入本书，于是出现了本 hack。本 hack 跟调试没什么关系，但希望大家能喜欢。

参考

- インテル®エクステンデッド・メモリ 64 テクノロジ・ソフトウェア・デベロッパーズ・ガイド[注7]

  http://download.intel.com/jp/developer/jpdoc/EM64T_VOL1_30083402_i.pdf

- インテル®プロセッサの識別と CPUID 命令[注8]

  http://download.intel.com/jp/developer/jpdoc/Processor_Identification_071405_i.pdf

378

- AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions

  http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf

——安部东洋

注 7：译者注：该文档的英文标题是《Intel® Extended Memory 64 Technology Software Developer's Guide》。

注 8：译者注：该文档的英文标题是《Intel® Processor Identification and the CPUID Instruction》（http://www.intel.com/Assets/PDF/appnote/241618.pdf）。

附录

Debug hacks 术语的基础知识

379

这里简单地介绍本书出现的有代表性的术语。

ABI

Application Binary Interface 的缩写，与规定源代码级别接口的 Application Programming Interface（API）不同，它规定了目标代码（即二进制代码）级别的接口，各种 CPU 架构各不相同。具体来说，它规定了函数调用时的参数传递方法、返回值的返回方法等。使用 GDB 调试不包含调试信息的可执行文件或分析崩溃转储（crash dump）时，是否理解 ABI，对调试者的水平有很大影响。

APIC、Local APIC、I/O APIC

Advanced Programmable Interrupt Controller 的缩写，是 SMP 系统中管理中断信号的控制器。APIC 包括各 CPU 内部实现的 Local APIC，以及接受周边设备的中断申请并将其通知某个 CPU 的 I/O APIC。

Asianux

由日本的 Miracle Linux（MIRACLE LINUX CORPORATION）、中国北京中科红旗软件技术有限公司（Red Flag Software Co., Ltd.）、韩国 Haansoft（HAANSOFT, Inc.）及 3 家公司联合成立的 Asianux Corporation 4 家公司共同开发的 Linux OS 的名称，也是该开发项目的名称。

Asianux Server 3

Asianux 项目开发的面向服务器的 Linux 发行版。它开发并包含了 kdump、Live Patch（KAHO）、kprobes、jprobes 等本书介绍的许多功能，将内核和用户空间（userland）的功能针对服务器进行了优化。它是 MIRACLE LINUX 4.0 的下一版本，也称为 MIRACLE LINUX V5。

380

CentOS

以完全兼容 RHEL（Red Hat Enterprise Linux）为目标的免费 Linux 发行版。它是根据 Red Hat 公司免费公开的源代码，去掉商标和商业软件包后重新编译的结果。

crash

分析崩溃转储（crash dump）的命令行工具。虽然也能分析执行中的内核，但由于无法停止内核的运行，因此称不上调试。Linux 上的内核调试器有 kgdb、KDB 等，但本书未介绍。

Debian GNU/Linux

由 Debian Project 开发的非商业 Linux 发行版。它拥有丰富的社区开发的软件包，软件包管理系统、设置方法等与 RedHat 家族的发行版有很大不同。极其重视遵守 Debian 社会契约（Debian Social Contract）和契约中包含的 Debian 自由软件指引（DFSG）。

diskdump

采集内核转储（kernel dump）的功能，被 RHEL4 等 RedHat 家族的一部分发行版采用。详细情况请参考 **“HACK#19 使用 diskdump 获取内核崩溃转储”**。

Fedora

由 Red Hat 支持的“Fedora Project”社区开发的免费 Linux 发行版。由社区进行开发，开发成果吸收进 RHEL，因此其目的为开发和验证。第 6 版之前称为 Fedora Core，从第 7 版开始仅称为 Fedora。

GDB、gdb

GNU/Linux 系统中的标准调试器。指代程序名称时常写为 GDB，指代命令名时常写为 gdb。使用 GDB 可以使程序在任意位置暂停，以进行变量值检查、调用顺序确认等调试的必要操作。本书详细说明了 GDB 的使用方法。

381

git

源代码管理工具的一种，Linux 内核代码管理用的就是 git。本书中不仅指代 git 工具，还指代由 git 创建的代码库（repository）。

i386

Intel 的 32 位架构的通称，由最初使用该架构的 CPU 而得名。Intel 的 32 位处

理器架构后来经过多次扩展，分别称为 i486、i586、i686，这些架构的总称也是 i386。

IPMI watchdog

检测系统停止响应（freeze，stall）的功能。能够在检测到系统停止响应时获取内核转储后重新启动。此外，即使发生致命错误，连该功能都无法执行，也能通过硬件强行重启或关闭电源。但是，实现该功能需要能检测系统停止响应的专用硬件，这种硬件通常只有中高端服务器才会配置。详细情况请参考**“HACK #23 用 NMI watchdog 在死机时获取崩溃转储”**。

kdump

收集内核转储的功能。在 RHEL5 等内核版本不低于 2.6.13 的发行版中可以使用。详细情况参考**“HACK#20 使用 kdump 获取内核崩溃转储”**。

kill [-9]

给进程发送信号的命令。指定选项“-9”则发送 KILL 信号，操作系统将强行关闭进程。此外，不指定选项时发送 TERM 信号。此时，一般情况下进程会自行结束，但如果进程停止响应，就无法执行结束处理。

NMI watchdog

检测系统停止响应的功能。检测到系统停止响应时，就采集内核转储并重新启动。该功能可以在大多数 PC 上使用，但与 IPMI watchdog 不同的是，它无法在致命错误情况下通过硬件强制重启。详细情况请参考**“HACK#22 死机时利用 IPMI watchdog timer 获取崩溃转储”**。382

objdump

获取 Linux 内核（vmlinux）或执行文件信息的命令。可以显示文件的头部（header）和各个节（section），也可以对代码进行反汇编，或是查看数据。详细请参照**“HACK#44 objdump 的方便选项”**。

OOM Killer（Out of Memory Killer）

是 Linux 内核的一种机制，可以在当操作系统无法分配必要的内存空间时，强行结束进程以获得空闲内存。该机制的目的是为了防止空闲内存不足而导致的操作系统停止响应这种最坏的情况。

ps

显示进程一览的命令。通过选项可以获得执行用户、进程组 ID、节 ID、PID、父进程 PID、TTY、内存使用量、运行状态、开始时间、执行时间、优先级、命令行选项等详细信息。

RHEL

Red Hat, Inc.开发并销售的面向企业的商业 Linux 发行版，为 Red Hat Enterprise Linux 的简称。打过更新补丁的版本简称为 RHEL4.7（旧称 Red Hat Enterprise Linux 4 Update 7）、RHEL5.2 等。软件包管理系统采用 RPM（RPM Package Manager，以前称为 Red Hat Package Manager）。

SIGSEGV、segmentation fault

内存非法访问的意思。代码跳转到不存在的地址或向不能写的内存空间试图写入时，操作系统就会通知该信号。如果不存在相应的信号处理程序，进程就直接结束。它是由 bug 引起的典型现象之一，参见[HACK#26]。

SMP

Symmetric Multi Processing 的缩写，系统中配置多个 CPU 核心，每个 CPU 核心都可以执行同样的处理。由于可以同时执行多个处理，与 UP 相比，软件层面上需要考虑的事情更多，由 SMP 引发的 bug 也屡屡发生。

383

softdog

检测系统停止响应的功能。可以在检测到系统停止响应时，采集内核转储并重新启动。与 IPMI watchdog 和 NMI watchdog 不同，该功能尽管能在所有 PC 上使用，但在某些原因引起的系统停止响应的情况下无法正常工作。

strace

跟踪进程的系统调用的命令。在调查由于系统调用引发错误时、区分错误发生地点时能发挥巨大威力。详细情况参见**“HACK#43 使用 strace 寻找故障原因的线索”**。

syslog

记录内核或进程的信息（message）的文件。该文件的名称在各种发行版中有所不同，RedHat 家族中大多数为/var/log/messages，Debian 家族中多为/var/log/syslog。大多数情况下，发生错误时，该文件中会记录一些信息。

SysRq 键

在 101 键盘上（及兼容键盘上）相当于“Alt+PrintScreen”键。在 Linux 下的详细使用方法请参见**“HACK#18 使用 SysRq 键调试”**。

top

将进程按照 CPU 负载排序并显示的命令。除了 CPU 负载之外，内存使用量、优先级、执行时间、PID 等信息也会一同显示，可以方便地获知进程的状态。

UP

Uni Processing 的缩写，只包含一个 CPU 核心的系统。

VMware

VMware Workstation、VMware Server、VMware Fusion 是虚拟机软件，能在硬件上的宿主操作系统中创建虚拟机并执行。VMware ESX、ESXi 是 Hypervisor 类虚拟机软件，它不需要宿主操作系统，名为 VMkernel 的软件直接运行在硬件上（RING0），以此建立虚拟机环境。

x86_64

AMD 的 64 位架构，以及与之兼容的 Intel 64 位架构的通称。Intel 的 64 位架构 384
中还有个 ia64，但这两者是完全不同的。x86_64 架构的特征是它能执行 i386 指令，因此在使用 64 位环境的同时也能继续沿用 i386 的软件资源。

Xen

Hypervisor 类型的虚拟机软件，运行在 RING0 之上。Xen 将虚拟机的运行单位称为 Domain（域），Domain 中包含访问物理硬件和管理其他域的特权 Domain，称为 Domain 0；还有作为一般的虚拟机使用的 Domain U 和 HVM Domain 等。

汇编语言

将机器语言用人类容易理解的形式书写的低级语言，汇编语言程序使用名为助记符（mnemonic）的命令书写。助记符是与机器语言一一对应的，各种 CPU 架构的助记符都不相同。针对特殊用途而优化的程序中，即使是现在也仍然有使用汇编语言编写的。Linux 内核中的异常处理和处理器模式切换等功能也包含大量的汇编语言。本书的 hack 中也写做汇编语言、汇编代码。

attach、detach

利用调试器连接正在运行的进程称为 attach，切断连接称为 detach。本书中介

绍了利用 GDB 或 strace 等 attach 到进程上进行调试的方法。

内核

操作系统核心部分的程序。本书指 Linux 内核。

内核配置

给内核添加功能、删除不用的功能，或是设置内核参数的内核配置。默认情况下调试功能是禁用的，要使用调试功能，就要改变内核配置。**"HACK#34 kernel panic（链表破坏篇）"、"HACK#57 错误注入"** 介绍了内核配置的实例。

内核转储（kernel dump）、崩溃转储（crash dump）

将某个时刻的内核内存映像和寄存器内容等保存到文件中的行为，或是指保存
385 后的文件。许多实现中可以选择只保存内核内存映像，还是保存全部内存映像等条件。使用 crash 等分析工具和崩溃转储，就能获知问题发生时的内核状态。

内核参数、内核启动参数

GRUB、LILO 等启动器（boot loader）启动 Linux 时设定的参数。通过这些参数，无须重新编译内核即可改变内核的行为和设置。

内核信息（kernel message）

内核输出的信息，其内容从调试信息到致命错误信息等无所不包，某些重要的内容还会输出到控制台或 syslog 中。此外，通过 dmesg 命令也可以查看。

反汇编

将机器语言变成人类易于理解的汇编语言的软件。Linux 中的代表性工具为 objdump，一般的调试器也有反汇编功能。反汇编在本书中无处不在，是调试的必备工具。

崩溃（crash）

指用户应用程序或操作系统突然停止运行。绝大部分原因是软件 bug，也有是因为硬件的故障。如果是软件 bug，那么一般情况下，用户应用程序崩溃时要采集内核转储（core dump），操作系统崩溃时要采集崩溃转储（crash dump），然后进行调试。

内核转储（core dump）

指将某一时刻的处理器的内存映像和寄存器内容等保存到文件中的动作，或指

保存后的文件，保存后的文件也称为内核转储。一般来说，程序异常结束时，某些操作系统会生成内核转储。GDB 等调试器与内核转储结合使用，就能获知问题发生时的处理器状态。详情参见 **“HACK#4 获取进程的内核转储”**。

调度器（scheduler）

多任务操作系统中按照进程或线程等运行单位来分配 CPU 使用权的功能，Linux 中也称为任务调度器（task scheduler）。由于调度器能大幅度影响系统的响应性能，可以说是操作系统的核心。调度器中发生 bug 的话，出现程序无法结束、系统挂起甚至崩溃等致命故障的可能性就会大大提高，而且调试起来非常困难。

386

栈溢出（stack overflow）

在栈上保存的数据量超过栈空间的大小。发生栈溢出后，操作系统就会给进程发送 SIGSEGV 信号。执行递归的程序很容易发生该现象。

停止响应（stall）、挂起（hang）

系统或进程不响应的状态，也称为“死机”、“当机”、“宕机”等。[注1]

stock kernel（标准内核）

由 Linus Torvalds 发布的、作为标准的 Linux 内核，也称为 main line、vanilla kernel（纯正内核）等。

自旋锁（spinlock）

锁机制的一种，是 Linux 内核中用得最多的一种锁。由于该锁的处理代价很低，常在临界区（critical section）等短小的地方使用。自旋锁引发死锁时，通常可以由 NMI watchdog 或 IPMI watchdog 获取崩溃转储。详情参考**“HACK#37 内核停止响应（自旋锁篇之一）”**、**“HACK#38 内核停止响应（自旋锁篇之二）”**。

线程、进程

在 Linux 中，线程是内核管理的最小的程序运行实例，进程是由一个或多个线程组成的程序运行实例。进程内的各个线程共享内存空间，所以某个线程发生内存非法访问等 bug，经常会影响到其他线程的运行。

注 1： 此处原文介绍了日语中的别称，不再原样翻译。——译者注

section（节）

ELF、COFF 等目标文件格式中的一部分，本书指 ELF 格式。在 ELF 中，程序的只读数据为.rodata section，无初始值的数据为.bss section，可执行代码为.text section，编译器或连接器会为每个 section 按照其功能进行定位。本书的
387
“HACK#28 数组非法访问导致内存破坏”和“HACK#59 Linux 内核的 init 节”中介绍了与 section 有关的调试案例。

信号量（semaphore）

锁机制的一种。Linux 用户应用程序可以使用 Posix 信号量和 IPC 信号量。Linux 内核内部也使用信号量，称为内核信号量（kernel semaphore），或者仅仅称为信号量。内核信号量上发生死锁时，通常通过 SysRq 键来采集崩溃转储。详情参考“HACK#39 内核停止响应（信号量篇）”。

任务（task）

本书指进程、线程之外的一般处理。

死锁（dead lock）

互斥处理的故障的一种，指陷入永远等待锁的状态。多次尝试给同一个锁加锁，或者按照错误的顺序加锁而导致多个线程互相等待锁的情况下，就会发生持有锁的同时等待锁事件发生的情况，是系统停止响应的典型原因之一。

缓冲区溢出

将数据写到缓冲区的区域之外的 bug。例如，在 C 语言中使用数组时经常会发生这种 bug。如果缓冲区旁边就是其他数据区域，就会导致其他数据被破坏，引发程序的异常行为。如果缓冲区旁的内存空间没有映射，就会引发 segmentation fault。

panic、kernel panic（内核挂起）

内核中发生致命错误，导致操作系统完全停止。Linux 则会显示 Oops 等错误信息，然后停止。发生 panic 后只能重新启动，但如果设置了崩溃转储（crash dump），就会采集转储文件。本书也说明了该错误信息及分析崩溃转储的案例。

奇偶校验错（parity error）

由于噪声等硬件原因，导致内存中的一个比特出错的故障。从原理上来说，所有 PC 都有可能发生该故障，但检测该故障则需要内存控制器等硬件的支持。

388

进程/任务状态

Linux 中的重要进程状态包括 RUNNING、INTERRUPTIBLE、UNINTERRUPTIBLE、STOPPED、TRACED、ZOMBIE 等。RUNNING 为程序正在运行的状态；STOPPED 为停止状态；TRACED 为被调试的状态；ZOMBIE 为进程结束后，父进程在等待结束状态被读取的状态。

机器语言

CPU 可以直接理解的指令。实际上为由 0 和 1 两个值表示的二进制串，但处理器手册上通常写为十六进制数。人类很少直接使用机器语言，通常使用与机器语言一一对应的汇编语言，或使用 C 语言等高级语言进行编程和调试。

死循环

程序中一系列无限反复执行处理的状态，但本书中仅指代出乎程序员意料的死循环，即由于 bug 引起的死循环。其原因有很多，从简单的数据使用错误到复杂的忘记互斥处理等。这是系统停止响应的代表性原因之一。详情参考**“HACK #32 应用程序停止响应（死循环篇）”**和**“HACK#36 内核停止响应（死循环篇）”**。

用户模式、内核模式

CPU 的执行模式，也称为特权级别。CPU 至少有两种以上的运行模式，不同运行模式对能访问的地址空间和 CPU 指令有不同的限制。内核模式为特权模式，可以访问所有地址空间，执行所有 CPU 命令。用户模式为非特权模式，无法访问内核空间，也不能执行 CPU 的特权命令。包括 Linux 在内的一般操作系统的内核代码运行在内核模式下，用户应用程序的代码运行在用户模式下。

用户空间、内核空间

支持虚拟内存的操作系统中所用的、对虚拟地址空间的分类方法，像 Linux 和 Windows 等常见操作系统都使用了虚拟内存。用户应用程序能访问的地址空间称为用户空间（或称为 userland），只有内核能访问的地址空间为内核空间。

389

实时进程

这种进程比一般进程的优先级更高，在主动释放 CPU 或被优先级更高的实时进程抢占（preempt）之前，一直占有 CPU。实时进程一旦失控，就会发生系统停止响应等故障。详情参见**“HACK#40 实时进程停止响应”**和**“HACK#65 使用 RT Watchdog 检测实时进程失去响应”**。

异常

CPU 在运行中检测出了异常或特殊状态。例如，被零除、非法内存访问、无效指令执行、调试指令执行等。

竞态条件（race condition）、冲突

指多个线程或进程对共享资源进行某种操作时，由于执行时机而导致无法预测的状态。这种问题大多数十分复杂，相关的共享资源越多，确定问题原因就越困难。大多数原因是忘记互斥处理或数据访问顺序不一致。

锁

互斥控制中使用的一种同步机制。多个线程之间共享资源时，通过锁可以保证数据的一致性，锁保护的操作区间称为临界区（critical section）。用户应用程序常用的代表性的锁有 mutex（互斥量）、信号量（semaphore）等。Linux 内核常用的有自旋锁（spinlock）、信号量、RCU、顺序锁（sequential lock）等。忘记加锁，会很容易发生数据被破坏、死锁等 bug。除了锁之外，其他同步机制还有通过原子指令操作资源的方法。

中断

主要指外围设备用非同步方式向 CPU 发送信号的机制。包括 Linux 内核在内的许多操作系统在中断发生时会中断现行操作，转而执行该中断信号的处理（中断处理程序）。软件中如果不考虑这种随时可能发生的中断，就容易混入 bug。

总结

390

上面介绍了《Debug hacks 中文版》一书中出现的术语。各个术语在正文中出现时也根据需要做出了解释。除了此处的介绍之外，还可以参考《BINARY HACKS》一书中的 **"hack #2 Binary hacks 术语的基础知识"**。

参考文献

- 《BINARY HACKS——ハッカー秘伝のテクニック 100 選》（O'Reilly Japan 出版）[注2]

——安部东洋、大和一洋、吉田俊辅

注 2：译者注：该书的中文版为《Birary hacks——黑客秘笈 100 选》，译者蒋斌，中国电力出版社，ISBN：9787508387932。

索引

数字·符号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y、Z

あ行（a 行）

か行（ka 行）

さ行（sa 行）

た行（ta 行）

な行（na 行）

は行（ha 行）

ま行（ma 行）

や行（ya 行）

ら行（ra 行）

わ行（wa 行）