BINARY HACKS™

黑客秘笈100选

高林 哲、鵜飼 文敏、佐藤 祐介、
浜地 慎一郎、首藤 一幸 著
蒋斌 杨超 译

O'REILLY®
中国电力出版社

BINARY HACKS™

黑客秘笈100选

高林 哲、鵜飼 文敏、佐藤 祐介、
浜地 慎一郎、首藤 一幸 著
蒋斌 杨超 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

图书在版编目（CIP）数据

BINARY HACKS™：黑客秘笈100选 /（日）高林哲等著；蒋斌，杨超译.
北京：中国电力出版社，2009
书名原文：BINARY HACKS™
ISBN 978-7-5083-8793-2

I. B… II. ①高… ②蒋… ③杨… III. 计算机网络－安全技术
IV. TP393.08

中国版本图书馆CIP数据核字（2009）第065928号

北京市版权局著作权合同登记
图字：01-2009-2300号

书　　名/ BINARY HACKS
书　　号/ ISBN 978-7-5083-8793-2
责任编辑/ 胡顺增
封面设计/ O'Reilly Japan，张健
出版发行/ 中国电力出版社（www.infopower.com.cn）
地　　址/ 北京三里河路6号（邮政编码100044）
经　　销/ 全国新华书店
印　　刷/ 汇鑫印务有限公司
开　　本/ 787毫米×1092毫米　18开本　21.25印张　379千字
版　　次/ 2010年1月第1版　2010年1月第1次印刷
印　　数/ 0001—3000册
定　　价/ 39.00元（册）

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的*The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在则编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目录

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

本书寄语

在2005年年末，以本书的作者高林先生等为中心举办了“Binary 2.0会议”。听到这个消息的时候，我不由笑了。表面上看这是一个用流行的Web2.0和轻量语言格式叙述低级别的技术、推崇不协调的策划，但是它的定位确实很有意义。

随着计算机高性能化的发展，编程的环境也大大改变了。想要编程首先必须要有编辑器和编译器的说法已经过时了。现在的程序员从一开始就拥有能够提供大量帮助的非常有效的开发工具。一旦有了新方案，就能方便地运用轻量级脚本语言、已有的库以及网络相关的各种服务轻松进行组合，使用非常方便。

若能熟练掌握最先进的工具，并能把巧妙的程序设计方法付诸实施的话，那实在是件很美妙的事情。然而，仅仅是查询一下编译器里跳出的数字信息，或是更新一下运行中的程序，确实没有太多意思。像让存储器内充满数据，从而耗尽机器周期这样的高超技术，尽管曾有人跃跃欲试，但若要真的付诸实施，则不是件容易的事情。

那么，为何当今社会中二进制如此重要？

工具的力量，或者说是抽象的力量。通过特定切入点对现实加以简化，并忽略与本质无关的复杂程序，只把精力集中到需要思考的问题上。但是，抽象化中必然存在其成立的前提。当系统正常工作的时候，前提的存在往往不会引人注意，但当系统的使用达到极限，前提便不复存在。抽象的界限会出现紊乱。在抽象的高级“世界”，不管构建如何强大的逻辑，根基松动则逻辑

也会随之土崩瓦解。如果对抽象界限所面对的世界缺乏起码的了解，要想重新构建抽象世界则完全不可能。

只使用高级工具的程序员，是在抽象化的世界里游刃有余的人。在抽象化世界可以做很多有趣的事情，但如果是光靠兴趣的话就并非那样容易了。如果对抽象化的范围没有把握好，就无法察觉自身世界的限制，对于专业编程人员而言这是致命的。因为不仅不能解决问题，还不能彻底理解抽象世界构建者的设想。例如，函数指令、回车等习惯用语都被作为真理接受，若不知道这个原理，想要熟练使用计算机就会很困难。如果你想拥有程序员的实力，那就好好利用本书吧。

表面上的介绍就到此为止。其实，本书的真正魅力并不在此。

孩提时代，在大件垃圾场看到被扔掉的电视机，也有过打开满是灰尘的电视机外壳往里面看的经历吧。看到像血管一样缠绕着的各种颜色的接线、隐藏在玻璃管中的奇怪的零部件，就像偷看了什么奇特的东西一样的兴奋。即使到了已经成年的现在，在Web上的关于个人电脑系列的新闻网站上看到最新的个人电脑分析文章，就不由自主地联想到了真实的线路板照片。

如果你和我以及本书的编者们一样拥有相同兴趣的话，在当前这种万物具备的开发环境中，就应该有种不满足的感觉。总是想着如何打开黑箱，然后把里面看个究竟。如果时间允许的话，甚至想自己组装出个什么东西来。

开发效率低也好，土里土气不怎么出色也罢，总之在抽象化的世界里我们总能够发现它令人惊叹的魅力。你选择了与专业编程相关的东西，那么你现在就站在这个原点上了。充满挑战的21世纪也好，五彩缤纷的大千世界也好，何不带着好奇心在这个看似低水平的世界搜寻一番看看呢？那么这本书就是你最好的参考选择。

Enjoy Hacking.

川合史朗

编写说明

关于编者

高林 哲　Satoru Takabayashi

软件工程师。1997年开发了全文检索系统Namazu，此后一直从事大量免费软件的开发。2005年负责IPA“未知软件创造事业”中“源码搜索引擎”的开发，被称为超级创造师。工学博士，兴趣是“破坏一切所知道的”。*http://0xcc.net/*。

鵜飼 文敏 Humitoshi Ukai

Debian 项目的官方成员、 Debian JP 项目前负责人、日本 Linux 协会前会长、The Free Software Initiative of Japan 副理事长、2004 和 2005 年度“未知软件创造事业”项目经理。自从攻读研究生期间研究用 PC98 架构操作 386BSD 和 Linux 以来，就一直沉浸在免费操作系统的世界里。作为 Debian JP 项目的创建人员，一直以 Debian 为活动中心，并进行着 *Debian.or.jp* 以及 *linux.or.jp* 等的运营管理。

佐藤 祐介 Yusuke Sato

软件工程师，早稻田大学理工部毕业后，从事软件开发，现在在某制造企业进行信息家电类的安全系统脆弱性检查。日本 SELinux 成员会（LIDS-JP）、JSSM 安全 OS 研究会、Linux 国际财团安全部成员。

浜地 慎一郎 Shinichiro Hamaji

喜欢将技术运用于不同寻常的领域，爱好广泛也有过很多尝试，但最终兴趣转移到了免费软件的批量生产上来。量子信息研究院研究生。

首藤 一幸 Kazuyuki Shudo

以“要做出别人做不出的东西”为格言的工程师，开发了Java线程转移系统、Just-In-Time编译器、覆盖构造工具系列等的软件。Utagoe株式会社董事、技术总监，信息科学博士。研究领域涉及广域分散处理、程序语言处理系统和信息安全等方面。

关于贡献

本书收录了许多Binary Hacker寄出的大量Hack。在此介绍一下他们的简况。

後藤 正德 Masanori Goto

在计算机领域中，从事Debian GNU C Library和Linux内核等开放源代码软件的开发工程的活动。Debian项目正式开发者、YLUG（Yokohama Linux Users Group）发起人。现在在制造研究所里参与数据存储器、PC Cluser等的研发。

中村 実 Minoru Nakamura

在命令集和ABI的夹缝中生存着的现实中的Binary Hacker。开发了MIPS、SPARC、Alpha用的编译器，x86、SPARC、Itanium2用的Java VM的优化调试器。

中村 孝史 Takashi Nakamura

参与本书编写的重要成员，在控制设备方面具备专业Binary Hacker的知识。因为PC的OS会将硬件部分隐藏起来，有些不方便，但是如果没有了OS会更加麻烦。

田中 哲 Akira Tanaka

经常使用Ruby的人，可能Conservative GC就是引他走向Binary Hacker的道路。

八重樫 剛史 Takeshi Yaegashi

每天都是新的开始。

野首 貴嗣 Takatsugu Nokubi

将文件指令库一体化的 Perl 模块、做着 File::MMagic 的维护工作。继续着在没有 Namazu、Kakasi 等库的情况下将内容强行库化的工作。

前言

本书的书名是底层的编程技术。所谓底层的意思是指接近于原始状态的电脑。

软件世界是随着抽象化的重叠而进步的。汇编程序是对机器语言的抽象，C语言则是对汇编程序的抽象。因此，在C语言的基础上还存在着对C进行封装的各种脚本语言。抽象化将底层的复杂部分隐藏起来，以高效率、高安全性的方法为开发者提供编程手段。

但是，如果把底层的技术完全忽略也是不能进行编程工作的。想要彻底追求性能、想尽可能地提高可信赖度、想要解决偶尔发生的“像谜一样的错误”等，在这些时候，底层的基础知识是必不可少的。这是因为抽象化并不能保证万无一失。

例如，一旦Ruby和Perl的脚本由于段异常而异常终止的问题发生之后，就有必要在C的标准环境下探求原因，有的时候为了解决特殊的问题，譬如像“转换运行中的机器语言的代码”这样的技巧就很重要。如果不了解底层的基础知识，这些问题就不能解决。

本书的目的是为了介绍能在这些场合使用的“Binary Hack”技能。Binary Hack的名称是由0或1也就是在编程中处于最底层的二进制而来的。本书将Binary Hack定义为“应用了底层软件技术的编程技能”，因为这是基本工具的使用方法，所以也涵盖了一些应用了OS安全编程以及处理器功能的高水平技巧。

一直以来，由于这样的技能没能被好好地总结和归纳，出现了“知道的人则知道，不知道的人则不知道”的情况。因此，本书尝试着将这些技能总结从而达到谁都可以使用的效果。本书的编写以在实践中确实有用的Hack为中心，但也包含了一些实用效果不是很大却相当有趣的Hack。希望能通过本书，让读者们在掌握实际可用技能的同时也能体会到底层技术的乐趣。

本书的能力与限制

本书中编录了在Binary Hack中必不可少的基本工具的使用方法，还有能够处理像GCC的扩展功能、OS的系统级调用以及内联汇编程序（Inline Assembler）等高技术水平的问题。本书的目标平台为UNIX，特别是以GNU/Linux为重点。虽然不能处理Win32API的Windows程序中存在的Binary Hack的问题，但是在采用了Cygwin的GNU基本的开发环境中，本书的大多数Hack都能适用。

必备知识和参考文献

本书的读者要能在UNIX的命令行上能够进行基本的操作。关于UNIX可以参考《UNIX环境编程》（Brian W.Kernighan、Rob Pike著）等书籍。

另外，本书虽介绍了C和C++编程语言的使用规则，但是没有涉及C程序设计语言如何操作这些编程语言的基本知识。关于C和C++可以分别参考《C程序设计语言》（B.W.Kernighan、D.M.Rich著）、《C++程序设计语言》（Bjarne Stroustrup著）等书籍。在**[Hack #100]**中介绍了很多参考文献。一部分的Hack中使用了汇编语言，由于在本书中有详细的解释说明，所以即便没有汇编语言方面的知识也能读懂。

本书的构成

第一章 介绍

围绕着Binary Hack的初步知识，解释并说明了在本书中使用的各类技术用语，还介绍了Binary Hack的最基本工具。

第二章 目标文件Hack

目的在于加深对作为可执行文件以及共有库的目标文件的理解。首先，对GNU/Linux等系统中使用的ELF进行了相关说明，其次介绍了与库

相关的Hack。也对作为目标文件Hack基本的GNU Binutils的使用方法进行了说明。

第三章 GNU编程Hack

GNU的开发环境，也就是以GCC、glibc为代表的软件，它们拥有很多方便的扩展功能。在这一章节中介绍了能够最大限度发挥GNU开发环境功能的技巧。

第四章 安全编程Hack

安全编程是现代程序设计中最重要的课题之一。在这一章中介绍了防御安全漏洞的技巧，以及发现并消除安全漏洞的方法。

第五章 运行时Hack

如果程序在运行时能够进行自动更新，并能自动调整状态的话是很有趣的。在这一章节中介绍了一些适用于运行中程序的技巧。

第六章 profile调试器Hack

本章介绍了对使用Profile检查程序瓶颈的方法，以及使用调试器的高级方法。Profile工具中的gprof、sysprof、oprfile以及作为调试器的GDB都有所涉及。

第七章 其他的Hack

本章中介绍了不能归为以上章节的其他Hack知识。最后以参考文献的方式介绍了一些对Binary Hack有指导作用的书籍和网站等。

本书的使用方法

可以将本书按顺序从头至尾阅读，也可以从目录中选择自己感兴趣的地方开始阅读。但是如果想要了解Binary技术基础知识的话，最好先看看第一章的内容。如果编程经验不是很多的话，还是从每章的初级Hack部分开始阅读比较好。

约定

下面是本书中使用的排版约定列表：

等宽字体（Constant Width）

表示样例代码、文件的内容、控制端的输出、变量名、命令以及其他的代码。

等宽黑体（**Constant Width Bold**）

表示用户输入和必须转换的命令和文本。

这个图标表示暗示、建议或是普通的备忘录，以及一些有用的补充信息。

这个图标表示提示或警告。

各个Hack左边的温度计图标，表示相对难易度，分别为初级、中级、高级。

初级水平

中级水平

高级水平

关于样例代码的使用

本书的目的在于向读者提供一些在实际操作中有用的信息。一般来说，本书中的编码在各自的程序或文档中都可以使用。除了大部分转载的情况之外，没有必要向O'Reilly公司要求授权。例如，使用本书中的几个代码片段来编写程序没有必要请求授权。但是，如果以CD-ROM的形式把O'Reilly公司出版的书籍的样例代码用作商业用途或是大量发布的话，就必须先得到O'Reilly公司的许可。要引用本书以及用本书的样例编码来解决问题的话，没有必要获得授权，但是要转载本书的大部分样例编码作为产品说明书的话，就必须获得授权。

在引用本书的内容时，虽然不必标明信息的出处，但是如果标明了出处的话那更好。在标明出处时，请使用下面的格式：高林哲、鵜飼文敏、佐藤祐介、浜地 慎一郎 、首藤一幸编写《Binary Hack》（O'Reilly Japan），清楚写明标题、作者、出版社以及ISBN等。

关于样例代码的使用，如果超出了正常使用许可的范围，请与*japan@oreilly.com*联系。

意见和问题

虽然我们尽力来检查和确认本书的内容，但是难免还会有不妥之处，甚至可

能会导致误解。若有错误之处，敬请广大读者予以指正，更欢迎您提出有利于我们改进的意见。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

日本：

Intelligent Plaza Building 1F
26 Banchi 27, Sakamachi
Shinjuku-ku, Tokyo 160-0002
Japan

中国：

100035 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
奥莱理软件（北京）有限公司

要询问技术问题或对本书提出建议，请发送电子邮件至：

info@mail.oreilly.com.cn
japan@oreilly.com

本书的网页登载有勘误表、补充信息等。

http://www.oreilly.co.jp/books/4873112885/
http://www.oreilly.com.cn/book.php?bn=978-7-5083-8793-2

有关本书以及其他图书的更多信息，请参阅 O'Reilly Web 网站：

http://www.oreilly.co.jp
http://www.oreilly.com.cn

致谢

感谢参与编写本书的鹈饲文敏、佐藤祐介、浜地慎一郎 、首藤一幸以及给予我们许多帮助的後藤正德、中村実、田中哲、八重樫 剛史、野首貴嗣等诸位先生。很荣幸他们这些优秀 Hacker 能参与本书的编写工作。

感谢川合史朗君给本书作序。川合君不仅是精通所有技术的Hacker，并且还是优秀的作家、翻译家和演员。川合君能给本书作序是我们的荣幸。

本书的开端要追溯到2005年末。提倡以当时流行的Web 2.0语言为手段将Binary 2.0语言用Blog表示是在2005年11月，“Binary 2.0 Conference”的召开是在2005年的12月。虽然Binary 2.0没有明确的定义，没有多少人完全理解它的涵义，但是仍有超过100人参加Binary 2.0会议，会场十分热闹。

记得当时在会场，出席的O'Reilly Japan的渡里君和田村君说，“出一本关于Binary Hacks的书吧。”这句话成为了本书开端的契机。在此对给本书的创作提供机会以及对编写过程给予热切关心的渡里君和田村君表示最诚挚的谢意。

高林哲

第1章

介绍

Hack #1～4

Binary Hack入门

将用于Binary hack的各种技巧分为“各种工具、库”、“二进制格式”、“系统调用”、“操作系统自带的功能”、“处理器自带的功能”、“编译器自带的功能”等部分进行了介绍。

欢迎来到Binary Hack世界，虽然笼统地称为Binary Hack，其实包括如何活用各种工具、使用编译器等内容。本书试图把能应用于Binary Hack技术上的各种技巧予以分类。

各种工具、库

我们的生活中已经有很多用于Binary Hack的工具、库。其中重要的有GNU项目提供的包含在GNU Binutils在内的各种工具、库。在“**[Hack #2]**目标文件Hack”中有很多使用了GNU Binutils技术的Hack。另外还有“(**[Hack #3]**)用File确认文件的种类”、“**[Hack #54]**用Valgrind检测内存溢出”、“**[Hack #73]**用Libunwind控制调用链”、“(**[Hack #82]**) 用strace追踪系统调用”、“**[Hack #83]**用ltrace追踪共享库的函数指令”、“**[Hack #89]**用oprofile获得详细的系统Profile”、“**[Hack #95]**用Protable Coroutine Library（PCL）进行轻量的并行处理”等Hack，介绍用于Binary Hack的工具和库。本书不仅介绍了各种工具、库的使用方法，也介绍了他们内部结构相关的知识，以及怎样活用前人的智慧和成果从而更有效地进行Binary Hack。

二进制格式（Binary Format）

可执行文件和库都是结构化的二进制文件。这种二进制文件的格式有很多，但GNU/Linux中主要使用ELF这种格式。理解二进制拥有怎样的构造和信息以及是怎样发挥这些功能的，在进行Binary Hack时非常有用。本书中，介绍了“**[Hack #8]**用readelf标明ELF文件的信息”、“**[Hack #9]**用objdump处理目标文件”、“**[Hack #12]**用nm核对目标文件里的符号”、“**[Hack #25]**不使用glibc编写Hello World”、“**[Hack #67]**用libbfd浏览符号”、“**[Hack #75]**独立装载目标文件”等活用二进制里的信息Hack的技巧。

系统调用（System Call）

大部分应用程序都不必直接调用操作系统的系统调用就可编写。C库里面有许多函数是对系统调用（System Call）的抽象，即使使用`fread`和`fwrite`函数代替`Read`和`Write`，不进行系统调用也可以进行文件的输入和输出。但是在内存相关的高级处理或信号处理等情况下，就必须使用系统调用了。本书中，以“**[Hack #34]**在堆栈上进行编码”、“**[Hack #76]**用sigaltstack处理堆栈溢出”、“**[Hack #78]**通过信号处理程序修改程序结构”、“**[Hack #81]**用SIGSEGV确认地址的有效性”等Hack的方式介绍了各种使用系统调用的技巧。另外，“**[Hack #59]**怎样使用系统调用”将介绍如何在GNU/Linux系统中进行系统调用。

操作系统自带的功能

通常，系统调用和库函数是由POSIX、ANSIC等标准来决定的API（应用程序接口），依据这些API，程序便可移植到其他的操作系统上。但是，使用不可移植的操作系统功能通常也可实现程序的不可复制的复杂处理。本书中，用“**[Hack #60]**用LD_PRELOD更换共享库”、“**[Hack #64]**检查运行程序的路径”、“**[Hack #65]**检测载入过程中的共享库”、“**[Hack #75]**取得堆栈空间的地址”等Hack的方式介绍操作系统自带的功能。

处理器自带的功能

通常，若使用C语言等高级语言，在不同处理器之间没有什么显著差别。但是各处理器在使用其自身的寄存器和指令时，使用汇编程序必须利用寄存器存取数据。本书中，用“**[Hack #79]**取得PC（程序计数器）的值”、“**[Hack #91]**使用硬件的调试功能”、“**[Hack #92]**在C语言程序中设置断点”、

“[Hack #96]计算 CPU 的时钟数”、“[Hack #98]x86 的特殊浮点数运算指令”等 Hack 技巧对处理器技术进行介绍。使用处理器自带的功能，从而可以充分发挥处理器的性能。

编译器自带的功能

编译器是C语言及C++语言编程不可或缺的工具。编译器不仅能把源代码翻译成机器语言，还具有增强安全性和优化程序等功能。在本书的“[Hack #22]GCC 的 GNU 扩展入门”、“[Hack #23]在 GCC 中使用内联汇编程序”、“[Hack #24]活用 GCC 的内置函数优化”、“[Hack #42]GCC 安全编程入门”、“[Hack #46]用 fstack-protector 保护堆栈”等 Hack 中应用 GCC 的功能。另外，在“[Hack #32]用 GCC 生成的运行代码”、“[Hack #38]理解 g++ 的异常处理(throw)”、“[Hack #41]理解 g++ 异常处理的开销”等 Hack 中介绍了 GCC 将生成什么样的代码。使用 GCC 功能的 Hack 占了本书的很大一部分。

总 结

这里把应用于 Binary Hack 的技巧分为六大类进行介绍：“各种工具、库”、“二进制格式”、“系统调用”、“OS 功能”、“处理器功能”、“编译器功能”。让我们来看看以下章节介绍的各个 Hack。

—— *Satoru Takabayashi*

HACK #2 Binary Hack 用语的基础知识

本 Hack 介绍在 Binary Hacks 中出现的用语。

在 Binary Hack 的领域里有许多专业术语，这里从本书频繁出现的用语中选择有代表性的用语进行简单解释。

应用二进制接口 ABI（Application Binary Interface）

应用程序应该遵守的二进制级别的规则集。如函数调用时堆栈和寄存器的使用方法与标识符名称的使用规则，OS 与处理器有关规定。

应用程序接口 API（Application Programming Interface）

应用程序的使用、库函数的功能以及数据结构的标准。利用 API 进行程序设计，具有可以在各种支持 API 的软件中互换源代码等优点。

初始符号段 BSS Segment（Block Started By Symbol Segment）

存放未初始化数据的程序段。包含有像用C语言定义的全局变量 `int global;` 一样，有未被设定初始值的变量。目标文件中不占空间，程序开始时被内核初始化为0。在ELF中称为bss。

动态共享对象 DSO（Dynamic Shared Object）

在GNU/Linux中，动态链接的共享库多被称为DSO，*.so* 为其扩展名。

任意记录格式调试 DWARF（Debug With Arbitrary Record Format）

存放调试信息的数据格式，在GNU/Linux+GCC环境中普遍使用。

可执行可链接格式 ELF（Executable and Linking Format）

用于可执行文件、目标文件、共享库、内核文件的一种数据格式。被GNU/Linux和FreeBSD等系统使用。

GCC（GNU Compiler Collection）

GNU编译器套件。原本是GNU C Compiler的意思。表示GNU C Compiler时缩写成GCC。

glibc（GNU C Library）

GNU的C程序库，被GNU/Linux和Hurd等OS使用。

GNU（GNU's Not Unix）

原指GNU程序开发的OS，但省略掉GNU程序直接写成GNU的情况较多。

GNU/Linux

Linux内核操作系统。把GNU写在前面是为了表达对开发内核的GNU程序员的敬意。

全局偏移表 GOT（Global Offset Table）

实现PIC的必要数据。在PIC中，使用GOT间接引用对全局数据进行存取。

LLP64

long long和pointer都占64位。int和long都占32位。64位Windows采用的是LLP64数据模型。

LP64

long和pointer都占64位。int占32位。若是64位Linux的话写成LP64。多数Unix系列的OS采用LP64。

位置无关代码 PIC（Position Independent Code）

可以装载到任意位置执行的代码。数据存取与转移在相对偏移量上进行。

位置无关的可执行程序 PIE（Position Independent Executable）

位置无关的可执行程序。用较新的 GNU/Linux 可以生成 PIE，它具有安全性高等优点。

程序链接表 PLT（Procedure Linkage Table）

实现动态链接的必要数据，与 GOT 同时使用，间接调用动态链接的共享库的函数。

POSIX（Portable Operating System Interface for UNIX）

决定系统调用和信号等 OS 的 API 标准。许多 Unix 系列的 OS 以 POSIX 为标准（或者以此标准为目标）。

SUS（Single UNIX Specification）

命名为 Unix OS 的规格。最新版的 SUSv3 可以在网上浏览。由于历史原因，SUSv3 包含 POSIX。

线程本地存储 TLS（Thread Local Storage）

任一线程即使使用相同的同名变量，其实际存储值根据线程本身保持独立。在 GCC 中使用 `__thread` 这个关键词使用 TLS。

预链接（prelink）

使动态链接高速化的一种方法，更新运行文件与共享库，降低大部分动态链接开销。多被 Linux 采用。

x86

Intel 公司的 8086 系列处理器的简称，80486 以后有 Pentium 和 Xeon 等产品，也被称作 IA-32。

x86_32

为了与 x86_64 区别开来，32 位的 x86 体系结构有时也写成 x86_32。

x86_64

AMD 公司设计的 64 位处理器，也叫做 AMD64。Intel 公司实际上也生产同样的处理器，只不过名称叫做 EM64T。

内联汇编代码（inline assembly code）

内嵌于 C 语言等高级语言程序中的汇编语言代码，用于依赖于特定结构的处理和优化等。

大小端

多字节的字数据以怎样的顺序储存的规则，也叫做字节顺序。Endian来源于《格利佛游记》[由于在小人国里的小人非常小（身高6英寸），所以总是碰到一些意想不到的问题。有一次就因为对水煮蛋该从大的一端（Big-End）剥开还是小的一端（Little-End）剥开的争论而引发了一场战争，并形成了两支截然对立的队伍：支持从Big-End剥开的人就称作Big-Endians，而支持从Little-End剥开的人就称作Little-Endians]。

目标文件（object file）

编译器生成的中间文件，可执行文件与库通过链接生成目标文件。在GNU/Linux中，.o为其扩展名。广义上来说，有时可执行文件与库也称为目标文件。

反汇编（disassemble）

把机器语言翻译成汇编语言。

共享库（shared library）

程序运行时内存上的多个程序共同使用的库。一般的共享库都是动态链接。虽然也有静态链接的共享库，但这种情况很少。本书中的共享库都是指动态链接的共享库，也可叫做共享目标。

重定位（relocation）

包含机器码地址的链接或是载入时的更新。

信号（signal）

传送过程中的同步、异步事件，在POSIX中SIGKILL和SIGSTOP以外的信号可用信号处理器处理。

信号处理器（signal handler）

处理信号的函数。可通过`sigaction()`或`signal()`函数来设定。

签名（signature）

由名字、参数以及返回值的型号决定函数的型号。通常在C、C++等编译语言中，调用名字相同但签名不同的函数时会显示警告或发生错误。

系统调用（system call）

在用户级的应用程序中调用操作系统内核功能的组合。例如：`read()`、`fork()`。

符号（symbol）

不同于一般“符号”的意思，在Binary Hacks上下文中指链接器在识别函数和变量的时候用的名字。

符号表（symbol table）

存在于目标文件等中的符号表，在没有明确删除的前提下会留在可执行文件以及库中。

堆栈（stack）

存放堆栈帧的内存区。在Binary Hack中，堆栈不仅是数据结构的堆栈，指代内存区的堆栈的情况也很多。

堆栈帧（stack frame）

函数调用时必须的信息的集合，简称帧。包含参数、本地变量、保存的寄存器、返回值地址等。

堆栈指针（stack pointer）

为操作堆栈帧而使用的指针。x86等有专用于堆栈指针的寄存器。RISC（精简指令集）处理器中把其中一个通用寄存器作为堆栈指针系统使用的情况较多。

线程（thread）

程序执行的最小单位之一。与进程（process）的主要区别是在于资源共享的方法。通常，在线程中比在进程中资源的共享更加容易，一个进程中可以包含多个线程。

线程安全（thread safe）

多线程环境下能安全运行。在多数情况下，含有静态变量的函数并不是线程安全的。

段错误（segmentation fault）

访问不能访问的地址，或在不能输入的地址上输入时发生的错误。C语言以及C++语言程序经常碰到这种情况，也叫做违规访问。

可执行文件（executable file）

可以运行的文件，也叫做可执行文件，在GNU/Linux中存放在*/usr/bin*等目录下。

静态链接库（static library）

处于静态链接下的库，在GNU/Linux中，扩展名为*.a*。

静态链接（static link）

生成可执行文件时与库链接。通常，为了把库的内容链接到可执行程序中，在运行时不需要共享库中的文件。

工具链（toolchain）

由编译器、链接器、解释器等组成，用于生成本地程序，必要的一系列工具的总称。

数据段（data segment）

存放初始化数据的程序段，在ELF中称为.data。

代码段（text segment）

存放机器语言代码的程序段，一般设定为只读状态，在ELF中称为.text。

调试器（debugger）

对检查程序出错原因很有帮助。通过在调试环境中运行程序来进行栈调用跟踪和变量监视等。

调试信息（debug information）

调试时必要的信息，存在于可执行文件和共享库中。编译器选项-g可以把必要的调试信息加入目标文件。

符号重组（demangle）

把命名较为复杂的符号，恢复到原来方便阅读的简单记号。例如：_ZN3Foo3BarE=>Foo:Bar。

动态链接（dynamic link）

程序在运行时与库链接。运行时需要调用库的文件，库不存在的话运行时会产生错误。

动态链接库（dynamic link library）

动态链接的库，在Windows中称DLL。在GNU/Linux中多称为DSO。

名称改编（name mangling）

将函数名、签名改写成唯一意义的符号。用于C++和Java等语言中。例如：Foo::Bar=>ZN3Foo3BarE。

Binarian

精通Binary Hack方面技术的工程师，也叫做Binary技术员。

栈跟踪（backtrace）

所有函数的一览表，也叫做堆栈跟踪。

堆（heap）

用malloc()等动态分配内存的空间。在Binary Hack中，堆不仅是数据结构的堆，也多指内存区中堆，也叫自由存储区。

断点（break point）

在调试时程序运行暂停的地方，由函数名及源代码的行数等指定。

程序计数器（program counter）

CPU中寄存器的一种。存放程序当前正在运行的命令的地址，通常简称PC。也称指令计数器。

进程（Process）

程序执行单位的一种，正在执行程序的实体。通常进程具有唯一的process ID 。

Profiler

解析程序性能的工具。本书介绍gprof、sysprof、oprofile。

函数调用约定（Calling Convention）

调用函数的过程中，对数据以何种方式进行传递给予规定，是ABI的一种，视OS和处理器情况而不同。

运行时（Runtime）

指程序正在被执行的时候。运行时发生的错误称为运行时错误。

链接（Link）

指链接目标文件以及库。执行重新配置等。

反射（Reflection）

在运行中对自身程序进行检查。在C语言中无法提供具有此种反应的功能，但使用Binary Hack中的技巧可以实现类似的功能。

装载（Load）

将可执行文件和库加载到内存中。

总结

本章所介绍的是在Binary Hacks中出现的术语。在本文中会对各类用语做出必要的解释。

—— *Satoru Takabayashi*

用 File 查询文件的类型

可以用 file 命令并根据其内容查询其文件类型。

通过使用 file 命令，可以查询任意文件的内容。为查询文件的类型，要事先根据文件种类决定其扩展名。一般情况下都是根据文件扩展名来推测文件类型，但 file 命令是通过详细阅读文件内容找出特征量来确定文件的类型。

以下列举的是 file 命令在 GNU/Linux 中对自身进行查询的例子，该例子是以选项 -i option 和 MIME Mediatype 字符串来表示的。

```
$ file /usr/bin/file
/usr/bin/file: ELF 32-bit LSB executable, Intel 80386, version
1 (SYSV), for GNU/
Linux 2.2.0, dynamically linked (uses shared libs), stripped
$ file -i /usr/bin/file
/usr/bin/file: application/x-executable, for GNU/Linux 2.2.0,
dynamically linked
(uses shared libs), stripped
```

file 命令是按以下的顺序判断文件的类别。

- 查询 Device（设备）、Directory（目录）、Symbolic（符号）等特殊文件。
- 查询压缩文件。
- 查询 tar 文件。
- 基于 magic 数据库文件的查询。
- 查询 ASCII、Unicode 等文本文件。

如果上述类别都不符合，则判断为二进制文件。

Magic 数据库中记录了标记信息。通常保存在 /etc/magic、/usr/share/misc/file/magic 等里面。

前面的例子，利用了下面的标记信息。

```
#------------------------------------------------------------------
# elf: file(1) magic for ELF executables
#
# We have to check the byte order flag to see what byte order all the
# other stuff in the header is in
#
# What're the correct byte orders for the nCUBE and the Fujitsu VPP500?
```

```
#
# updated by Daniel Quinlan (quinlan@yggdrasil.com)
0 string \177ELF ELF
>4 byte 0 invalid class
>4 byte 1 32-bit
(..)
>5 byte 1 LSB
(..)
>>16 leshort 2 executable,
```

Magic 文件由以下四个部分组成。

- 起始偏移量。
- 数据的类别。
- 值。
- 输出字符串。

所谓“Level”就是写在偏移量前面的“>”。在第一行加了“>”之后，下一行如果写成“>10”的话，那就可以读取从最开始的位置往后推 10 字节，再进行比较。若是相同数量的“>”排列出现的话，可以依次读出前一个级别的偏移量的相关内容。在配置好的条目后面如果还多了一个“>”的话，那么还要继续进行这个条目的配置处理。

例如上述判别 file 自身的例子，就是按以下顺序进行操作的。

- 从开始的 0 位确认“\177ELF”是否相配（第 1 行）。
- 相配的话就用“ELF”。
- 确认从开始算起到第 4 个字节的第 1 个 1 字节是否为“0”（第 2 行）。
- 不匹配的话继续往下。
- 确认从开始到第 4 个字节的第 1 个 1 字节，判断是否为“1”（第 3 行）。
- 相配的话用“32-bit”表示。
- 暂不匹配的话继续往下。
- 确认从开始算起到第 5 字节是否为“1”。
- 相匹配的话用“LSB”表示。
- 暂不匹配的话继续往下。

- 确认从开始算起到第16个字节是否为“2”。
- 匹配的话用“executable”表示。
- 重复以上操作至结束。

另外，GNU/Linux系统上广泛使用的指令文件中，包含能专门处理ELF文件的编码。“for GNU/Linux2.2.0”以下的部分由这个特别处理编码输出。

总结

使用命令文件，可以从文件的内容来查询它的类别。另外，即使出现新的文件格式，只要知道文件原有的签名，在magic文件里进行追加就可以识别这个文件。

相反，若在有必要确定新的二进制文件格式的情况下，将能够用指令文件查询的签名格式化就可以了。

有关magic格式的详细情况请参考man magic。

—— *Takatsugu Nokubi*

用od转储二进制文件

本Hack对转储二进制文件的工具 —— od的使用方法进行说明。

八进制转存

od就是octal dump的意思（“octal”是指八进制），参数缺省时用八进制转存输出二进制文件。

```
% od /etc/ld.so.cache | head -5
0000000 062154 071456 026557 027061 027067 000060 001430 000000
0000020 000003 000000 047440 000000 047452 000000 000003 000000
0000040 047475 000000 047505 000000 000003 000000 047526 000000
0000060 047547 000000 000003 000000 047601 000000 047616 000000
0000100 000003 000000 047644 000000 047662 000000 000003 000000
```

行首的第一列数字是用八进制表示最开始的偏移量。每行以2字节为一个单位，输出8个单位，也就是依次以16个字节输出。2个字节（short）在机器字节指令上用八进制表现。这种情况，最初用八进制表示是062154，十进制是25708，十六进制是646C，二进制则是0110010001101100。

指定输出格式

一般来说，即使是用八进制输出也很难理解。通常，每个字节都用十六进制表示的话相对较容易理解。在od中，能用-t选项（--format选项）指定输出格式。-t选项指定以下类型。

类型	含义
a	文字的名称（7bit ASCII）
c	ASCII文字或escape文字
d	带符号的十进制数
f	浮点数
o	八进制数
u	无符号的十进制数
x	十六进制数

a、c通常以字节单位的输出。对于d、o、u、x则在“a、c”后表示字节数，或者它们能够用于以下类型来指定大小。

类型	含义
C	char
S	short
I	int
L	long

关于f可用于以下类型指定大小。

类型	含义
F	float
D	double
L	long double

另外，加上“z”的话，右边可显示ASCII码。

格式的书写也可以使用八进制以外的方式。-A选项指定下面任何一项都可以改变其格式的基数。

类型	含义
d	十进制
o	八进制（default）
x	十六进制
n	无格式化

经常使用的还是用十六进制转存每个字节。在这种情况下就用“-t x1 -A x”的格式。

```
% od -t x1 -A x /etc/ld.so.cache | head -5
000000 6c 64 2e 73 6f 2d 31 2e 37 2e 30 00 18 03 00 00
000010 03 00 00 00 20 4f 00 00 2a 4f 00 00 03 00 00 00
000020 3d 4f 00 00 45 4f 00 00 03 00 00 00 56 4f 00 00
000030 67 4f 00 00 03 00 00 00 81 4f 00 00 8e 4f 00 00
000040 03 00 00 00 a4 4f 00 00 b2 4f 00 00 03 00 00 00
```

要显示ASCII码，在查询时也可以像下面一样在后面加上“-t x1z”和“z”。

```
% od -t x1z -A x /etc/ld.so.cache | head -5
000000 6c 64 2e 73 6f 2d 31 2e 37 2e 30 00 18 03 00 00 >ld.so-1.7.0.....<
000010 03 00 00 00 20 4f 00 00 2a 4f 00 00 03 00 00 00 >.... O..*O......<
000020 3d 4f 00 00 45 4f 00 00 03 00 00 00 56 4f 00 00 >=O..EO......VO..<
000030 67 4f 00 00 03 00 00 00 81 4f 00 00 8e 4f 00 00 >gO.......O...O..<
000040 03 00 00 00 a4 4f 00 00 b2 4f 00 00 03 00 00 00 >.....O...O......<
```

c也表示ASCII码。但是如果指定了“c”，就会通过换行表示。

```
% od -t x1c -A x /etc/ld.so.cache | head -5
000000 6c 64 2e 73 6f 2d 31 2e 37 2e 30 00 18 03 00 00
     l   d   .   s   o   -   1   .   7   .   0  \0 030 003 \0 \0
000010 03 00 00 00 20 4f 00 00 2a 4f 00 00 03 00 00 00
   003  \0  \0  \0       O \0  \0    *   O  \0  \0 003 \0 \0 \0
000020 3d 4f 00 00 45 4f 00 00 03 00 00 00 56 4f 00 00
```

不省略转存（dump）

od在默认的情况下，如果很多行的内容相同时，将会省略重叠部分的转存。

```
% od -tx1 -Ax /usr/share/apache/icons/deb.png | sed -ne '45,49p'
0002c0 fe f8 fd ff f9 ff ff fb fd ff fb ff fd fd fd fe
0002d0 fd fd ff fd fe fe fe fe 00 00 00 00 00 00 00 00
0002e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
000320 00 00 00 00 00 00 00 00 00 7a 6c 49 5c 00 00 00
```

像这样0x0002e0~0x000320中“00”一直持续出现，所以省略掉只用“*”代替。若不想省略则用-v选项（--output-duplicates选项）。

```
% od -tx1 -Ax -v /usr/share/apache/icons/deb.png | sed -ne '45,49p'
0002c0 fe f8 fd ff f9 ff ff fb fd ff fb ff fd fd fd fe
0002d0 fd fd ff fd fe fe fe fe 00 00 00 00 00 00 00 00
0002e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

字符串

od也有像strings一样的字符串转存功能。使用-s选项的话，显示最少有连续3个ASCII字母并以“\0”结束的字符串的格式和内容。

```
% od -Ax -s /etc/ld.so.cache|head -5
000000 ld.so-1.7.0
007450 libz.so.1
00745a /usr/lib/libz.so.1
00746d libz.so
007475 /usr/lib/libz.so
```

不想输出格式的话使用（-A n）可以得到与strings类似输出。

```
% od -An -s /etc/ld.so.cache|head -5
ld.so-1.7.0
libz.so.1
/usr/lib/libz.so.1
libz.so
/usr/lib/libz.so
```

但是，与strings的输出略有不同，字符串只能是连续的ASCII码，并以\0结束的格式。若是strings的话，只要连续出现ASCII码，即使不以“\0”结束也能输出。

```
% diff -u =(od -An -s /etc/ld.so.cache) =(strings /etc/ld.so.cache ) | head
--- /tmp/zsh2QzQnw     2006-02-21 01:22:08.657320876 +0900
+++ /tmp/zshesbFEZ     2006-02-21 01:22:08.666319531 +0900
@@ -1,4 +1,5 @@
ld.so-1.7.0
+glibc-ld.so.cache1.1J
libz.so.1
/usr/lib/libz.so.1
libz.so
```

若转存的话，“glibc-ld.so.cacha1.1J”后面是\03，不是以“\0”结束。

```
% od -Ax -tx1z /etc/ld.so.cache|sed -ne '596,597p'
002530 67 6c 69 62 63 2d 6c 64 2e 73 6f 2e 63 61 63 68 >glibc-ld.so.cach<
```

```
002540 65 31 2e 31 4a 03 00 00 96 89 00 00 00 00 00 00 >e1.1J..........<
```

像 strings 一样，也可以指定字符串的最小长度。

```
% od -Ax -s12 /etc/ld.so.cache|head -5
00745a /usr/lib/libz.so.1
007475 /usr/lib/libz.so
007486 libxvidcore.so.4
007497 /usr/lib/libxvidcore.so.4
0074b1 libxslt.so.1
```

但是，不能同时使用 -t 选项与 -s 选项。

使用 od 的例子

在源代码里包含有映像的二进制文件的情况，将二进制文件转存并转换为适当的C语言数组的情况，仅用 od 和 sed 就可以进行以下操作。

```
#!/bin/sh
# $0 objname < in > out
objname=${1:-objname}
od -A n -v -t x1 | sed -e '1i\
const unsigned char '$objname'[] = {
s/\([0-9a-f][0-9a-f]\) */0x\1,/g
$s/,$//
$a\
};
'
```

脚本的第一个参数作为数组名。从标准输入读取二进制文件，向标准输出C的数组。

如果使用了 od -A n -v -t x1，可将标准输入的内容就以如下所示方式转存。

- 不表示偏移量（-A n）。
- 不省略，全部转存（-v）。
- 将每一字节转存为十六进制（-t x1）。

将这个输出利用 set 转换成如下所示的C数组。

- 将“const unsignet char 数组名[] ={”填入首行。
- 将每一字节变换为“0xNN”的形式。
- 删除最后一行最后的“,”。

- 最后再添上一行“};”。

总结

在本Hack里解释了使用od的方法，不仅能用八进制转存，也可以用各种格式化进行转存（dump）。像strings这样的二进制文件中包含的字符串也可以进行转存。

—— *Fumitoshi Ukai*

第2章

目标文件Hack

Hack #5～21

通常，目标文件指的是编译器生成的中间目标文件，但在本章中，将从广义上理解目标文件的含义，也包括可执行文件和库。目标文件不仅包含了机器语言代码，还包括符号表、调试信息、再配置信息等各种信息。

在本章中，将介绍读取目标文件所包含信息的方法，以及改写目标文件并删掉不需要的信息，或输入数据的方法。熟悉目标文件，是掌握Binary Hack的第一步。

ELF入门

在本Hack里，将介绍作为二进制目标和可执行文件的ELF格式。

ELF

ELF是Executable and Linking Format的缩写，是可执行二进制文件和目标文件等格式的相关标准。ELF格式的文件里，ELF在文件的头部，而程序头文件表格以及节头表格放在ELF头之后。

这些头文件的构造在`elf.h`里都有记录。

使用ELF的类型

ELF二进制有32位和64位两种。ELF使用以下的类型：N的部分如果是32就作为32位二进制类型，如果是64就作为64位二进制类型。例如，如果是32位二进制就表示为`Elf32-Half`，如果是64位二进制就表示为`Elf64-Half`。

类型名	N=32	N=64	说明
ElfN_Half	uint16_t	uint16_t	无符号 16bit 值
ElfN_Word	uint32_t	uint32_t	无符号 32bit 值
ElfN_Sword	int32_t	int32_t	有符号 32bit 值
ElfN_Xword	uint64_t	uint64_t	无符号 64bit 值
ElfN_Sxword	int64_t	int64_t	无符号 64bit 值
ElfN_Addr	uint32_t	uint64_t	地址
ElfN_Off	uint32_t	uint64_t	偏移量
ElfN_Section	uint16_t	uint16_t	节序号
ElfN_Versym	uint16_t	uint16_t	符号版本信息

ELF 头

ELF 头必然存在于 ELF 文件的开头，表明这是一个 ELF 文件。ELF 头的内容可以在 readelf 的 -h 选项（--file-header 选项）中查看。

```
% readelf -h /bin/ls
ELF 头
 magic：7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 类：                               ELF32
 数据：                             2 的补码，little endian
 版本：                             1 (current)
 OS/ABI：                           UNIX - System V
 ABI 版本                           0
 类型：                             EXEC (可执行文件)
 机器：                             Intel 80386
 版本：                             0x1
 入口地址：                         0x8049a50
 程序的起始头：                     52 (byte)
 节头的起点：                       74948 (byte)
 标志：                             0x0
 头的大小：                         52 (byte)
 程序头的大小：                     32 (byte )
 程序头数：                         8
 节头：                             40 (byte)
 节头数：                           25
 节头字符串表索引：                 24
```

ELF 头的结构如下：

```
unsigned char e_ident[EI_NIDENT];     /*magic
                                      * 类 ，数据，版本
                                      * OS/ABI，ABI 版本
                                      */
```

```
ElfN_Half    e_type;          /*类型*/
ElfN_Half    e_machine;       /*机器*/
ElfN_Word    e_version;       /*版本 */
ElfN_Addr    e_entry;         /*入口地址*/
ElfN_Off     e_phoff;         /*程序头的起点*/
ElfN_Off     e_shoff;         /*节头的起点 */
ElfN_Word    e_flags;         /*标志*/
ElfN_Half    e_ehsize;        /*头的大小 */
ElfN_Half    e_phentsize;     /*程序头的大小*/
ElfN_Half    e_phnum;         /*程序头数*/
ElfN_Half    e_shentsize;     /*节头的大小*/
ElfN_Half    e_shnum;         /*节头数 */
ElfN_Half    e_shstrndx;      /*节名的字符串表格*/
```

e_ident保存着ELF的幻数和其他信息。ELF文件里最前面四个字节里有如下的幻数。

```
| 0x7F | 0x45 | 0x4C | 0x46 |
```

用字符串表示，即"\177ELF"。

其后的字节如果是32位则是ELFCLASS32 (1)，如果是64位则是ELFCLASS64 (2)。其后的字节表示endian。

little endian则用ELFDATA2LSB (1)；big endian则用ELFDATA2MSB (2)。在此之后，ELF版本和OS、ABI等信息则用一个比特位表示。

e_type可以用以下的某一种类型来表示。

类型名	值	说明
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	共享目标文件
ET_CORE	4	内核文件

e_machine表示架构类型，在EM_里定义为初始量。

e_version表示ELF版本，现在为EV_CURRENT (1)。

e_entry是在ELF中开始执行的虚拟地址。

e_ehsize表示ELF头的大小。

e_phoff、e_phentsize、e_phnum表示程序头表格的位置和数量。

e_shoff、e_shentsize、e_shnum表示节头表格的位置和数量。

e_shstrndx表示包含节名的存储器表格的节头索引。

程序头

程序头表格是由ELF头的e_phoff指定的偏移量和e_phentsize、e_phnum共同确定大小的表格组成。e_phentsize表示表格中程序头的大小，e_phnum表示表格中程序头的数量。程序头本身含有e-phentsize * e-phnum。

程序头用readelf的-l选项（--program-headers）表示。

```
% readelf -l /bin/ls

Elf文件类型是EXEC (可执行文件)。
entry point ·0x8049a50
8 个程序头、起点偏移量 52

程序头:
 类型          位移地址  虚拟地址    物理地址    文件大小 内存大小 标记 校验
 PHDR          0x000034 0x08048034 0x08048034 0x00100 0x00100 R     E
 INTERP        0x000134 0x08048134 0x08048134 0x00013 0x00013 R     0x1
    [所要求的程序解释器: /lib/ld-linux.so.2]
 LOAD          0x000000 0x08048000 0x08048000 0x11d08 0x11d08 R     E
 LOAD          0x012000 0x0805a000 0x0805a000 0x003f4 0x007b0 RW    0x1000
DYNAMIC        0x012184 0x0805a184 0x0805a184 0x000d8 0x000d8 RW    0x4
NOTE           0x000148 0x08048148 0x08048148 0x00020 0x00020 R     0x4
GNU_EH_FRAME   0x011cdc 0x08059cdc 0x08059cdc 0x0002c 0x0002c R     0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW    0x4

指向段映射的节:
段节…
 00
 01 .interp
 02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr
 03 .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
 04 .dynamic
 05 .note.ABI-tag
 06 .eh_frame_hdr
 07
```

程序头结构如下：

```
ElfN_Word     p_type;      /* 段类型 */
ElfN_Off      p_offset;    /* 偏移量 */
ElfN_Addr     p_vaddr;     /* 虚拟地址 */
ElfN_Addr     p_paddr;     /* 物理地址 */
```

```
ElfN_Word    p_filesz;    /*文件大小*/
ElfN_Word    p_memsz;     /*内存大小(MemSiz)*/
ElfN_Word    p_flags;     /*标志(Flg)*/
ElfN_Word    p_align;     /*对齐*/
```

这些分别对应readelf_1表示的程序头的各行。

类型（p_type）介绍如下。

p_type	值	说明
PT_LOAD	1	装载的程序段
PT_DYNAMIC	2	动态链接信息
PT_INTERP	3	程序解释器
PT_NOTE	4	辅助信息
PT_PHDR	6	程序头表格本身
PT_TLS	7	线程本地存储器
PT_GNU_EH_FRAME	0x6474e550	GNU .eh_frame_hdr段
PT_GNU_STACK	0x6474e551	堆栈的可执行性

“指向段映射的节”以后的内容，列出了在Program Headers的各程序头中表示的每个段的内存范围里所包含的节的名称。也就是说，最开始的（索引00）程序头里所表示的段为PHDR类，但没有包含的节。紧接着的程序头（索引01）里所表示的段为INTERP类，所包含的节为interp。另外（索引02）的程序里所表示的段为LOAD类，其中含有interp、note、ABI-tag、hash、dynsym、dynstr、.gnu.version、gnu.version_r、.rel.dyn、.rel.plt、.init、.plt、.text、.fini、.rodata、.eh_frame_hdr等节。

节头

节头表格是由ELF头的e_shoff指定的偏移量以及e_shentsize、e_shnum共同规定了大小的表格组成。e_shentsize指定表格中节头的大小，e_shnum表示表格中节头的数量。程序头表格自身含有e_shentsize*e_shnum字节部分。

readelf的-S选项（--section-headers选项）指定是否显示节头。

```
% readelf -S /bin/ls
25个节头，起点偏移量0x124c4:
```

```
Section Headers:
[编号] 名称              类型            地址     Off    大小   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0   0
[ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0   1
[ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0   4
[ 3] .hash             HASH            08048168 000168 000338 04   A  4   0   4
[ 4] .dynsym           DYNSYM          080484a0 0004a0 0006b0 10   A  5   1   4
[ 5] .dynstr           STRTAB          08048b50 000b50 00047b 00   A  0   0   1
[ 6] .gnu.version      VERSYM          08048fcc 000fcc 0000d6 02   A  4   0   2
[ 7] .gnu.version_r    VERNEED         080490a4 0010a4 0000b0 00   A  5   3   4
[ 8] .rel.dyn          REL             08049154 001154 000028 08   A  4   0   4
[ 9] .rel.plt          REL             0804917c 00117c 0002e0 08   A  4   b   4
[10] .init             PROGBITS        0804945c 00145c 000017 00  AX  0   0   4
[11] .plt              PROGBITS        08049474 001474 0005d0 04  AX  0   0   4
[12] .text             PROGBITS        08049a50 001a50 00c880 00  AX  0   0  16
[13] .fini             PROGBITS        080562d0 00e2d0 00001b 00  AX  0   0   4
[14] .rodata           PROGBITS        08056300 00e300 0039dc 00   A  0   0  32
[15] .eh_frame_hdr     PROGBITS        08059cdc 011cdc 00002c 00   A  0   0   4
[16] .data             PROGBITS        0805a000 012000 0000e8 00  WA  0   0  32
[17] .eh_frame         PROGBITS        0805a0e8 0120e8 00009c 00   A  0   0   4
[18] .dynamic          DYNAMIC         0805a184 012184 0000d8 08  WA  5   0   4
[19] .ctors            PROGBITS        0805a25c 01225c 000008 00  WA  0   0   4
[20] .dtors            PROGBITS        0805a264 012264 000008 00  WA  0   0   4
[21] .jcr              PROGBITS        0805a26c 01226c 000004 00  WA  0   0   4
[22] .got              PROGBITS        0805a270 012270 000184 04  WA  0   0   4
[23] .bss              NOBITS          0805a400 012400 0003b0 00  WA  0   0  32
[24] .shstrtab         STRTAB          00000000 012400 0000c3 00      0   0   1
Flags 可能包含的值:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 O (extra OS processing required) o (OS specific), p (processor specific)
```

节头的构成如下：

```
ElfN_Word      sh_name;      /* 名称(在存储器表格里的索引)*/
ElfN_Word      sh_type;      /* 类型*/
ElfN_Word      sh_flags;     /* 标志(Flg)*/
ElfN_Addr      sh_addr;      /* 地址*/
ElfN_Off       sh_offset;    /* 偏移(Off)*/
ElfN_Word      sh_size;      /* 大小*/
ElfN_Word      sh_link;      /* 链接(Lk)*/
ElfN_Word      sh_info;      /* 节信息(Inf)*/
ElfN_Word      sh_addralign; /* 对齐(Al)*/
ElfN_Word      sh_entsize;   /* 节为表格的时候，各个条目的大小*/
```

这些分别对应 `readelf -S` 里所显示的 Section Headers 的每列。

在ELF头的e_shstrndx指定的节里包含存储器表格的索引，索引里包含其名称。在这个 `/bin/ls` 的例子中，`e_shstrndx` 是 24 即为第 24 个节头，成为保存其存储器表格的节。

```
[24] .shstrtab     STRTAB      00000000 012400 0000c3 00   0   0 1
```

从这可以看出，sh_offset是0x012400、大小为0x0000c3字节的存储器表格。

作为节的类型有以下种类：

section type	值	说明
SHT_PROGBITS	1	程序数据
SHT_SYMTAB	2	符号表格
SHT_STRTAB	3	存储器表格
SHT_RELA	4	带加数的再配置条目
SHT_HASH	5	符号散列数据表格
SHT_DYNAMIC	6	动态链接信息
SHT_NOTE	7	Notes
SHT_NOBITS	8	文件上无数据部分（.bss）
SHT_REL	9	再配置条目
SHT_DYNSYM	11	动态链接所使用的符号表格
SHT_INIT_ARRAY	14	constructor的排列（.init）
SHT_FINI_ARRAY	15	destructor的排列（.fini）
SHT_GNU_verdef	0x6ffffffd	版本定义节
SHT_GNU_verneed	0x6ffffffe	版本要求节
SHT_GNU_versym	0x6fffffff	版本符号表格

存储器表格

存储器表格是单纯的字符串的表，如/bin/ls等，以下的节都是存储器表格。

```
[ 5] .dynstr           STRTAB      08048b50 000b50 00047b 00  A  0  0  1
[24] .shstrtab         STRTAB      00000000 012400 0000c3 00     0  0  1
```

让我们来看看例24的.shstrtab。.shstrtab的偏移量是0x012400，大小为0xc3，因而使用od会得出以下数据。

```
% od --skip-bytes 0x12400 --read-bytes 0xc3 -t x1z /bin/ls
0222000 00 2e 73 68 73 74 72 74 61 62 00 2e 69 6e 74 65 >..shstrtab..inte<
0222020 72 70 00 2e 6e 6f 74 65 2e 41 42 49 2d 74 61 67 >rp..note.ABI-tag<
0222040 00 2e 68 61 73 68 00 2e 64 79 6e 73 79 6d 00 2e >..hash..dynsym..<
```

```
0222060 64 79 6e 73 74 72 00 2e 67 6e 75 2e 76 65 72 73 >dynstr..gnu.vers<
0222100 69 6f 6e 00 2e 67 6e 75 2e 76 65 72 73 69 6f 6e >ion..gnu.version<
0222120 5f 72 00 2e 72 65 6c 2e 64 79 6e 00 2e 72 65 6c >_r..rel.dyn..rel<
0222140 2e 70 6c 74 00 2e 69 6e 69 74 00 2e 74 65 78 74 >.plt..init..text<
0222160 00 2e 66 69 6e 69 00 2e 72 6f 64 61 74 61 00 2e >..fini..rodata..<
0222200 65 68 5f 66 72 61 6d 65 5f 68 64 72 00 2e 64 61 >eh_frame_hdr..da<
0222220 74 61 00 2e 65 68 5f 66 72 61 6d 65 00 2e 64 79 >ta..eh_frame..dy<
0222240 6e 61 6d 69 63 00 2e 63 74 6f 72 73 00 2e 64 74 >namic..ctors..dt<
0222260 6f 72 73 00 2e 6a 63 72 00 2e 67 6f 74 00 2e 62 >ors..jcr..got..b<
0222300 73 73 00                                        >ss.<
0222303
```

此时，存储器表格如下。

索引	字符串
1	.shstrtab
11	.interp
19	.note

也就是说，.shstrtab最前面的偏移量，成为存储器表格里的索引。

符号表格

符号表格是为了使符号和其值等相对应的表格。如/bin/ls，因为被剥离化了，所以只有动态符号表格。符号表格可以在.readelf和-s选项（--syms选项）中看到。

```
% readelf -s /bin/ls

符号表格 '.dynsym' 由107个条目组成。
   序号            值  大小    类型    Bind    Vis索引名
   0: 00000000     0  NOTYPE  LOCAL   DEFAULT UND
   1: 08049484    60  FUNC    GLOBAL  DEFAULT read
   2: 08049494   283  FUNC    GLOBAL  DEFAULT
   3: 080494a4    42  FUNC    GLOBAL  DEFAULT
   4: 080494b4    58  FUNC    GLOBAL  DEFAULT
（略）
```

试着从ELF头来理解这个表。

首先，要理解在节头中作为符号表格的.dynsym 。

```
[ 4] .dynsym     DYNSYM     080484a0 0004a0 0006b0    10   A  5  1  4
```

经转储得到下面的结果。

```
% od --skip-bytes 0x4a0 --read-bytes 0x6b0 -t x1z /bin/ls
0002240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >................<
0002260 6c 01 00 00 84 94 04 08 3c 00 00 00 12 00 00 00 >l.......<.......<
0002300 7d 03 00 00 94 94 04 08 1b 01 00 00 12 00 00 00 >}................<
0002320 3b 01 00 00 a4 94 04 08 2a 00 00 00 12 00 00 00 >;.......*.......<
0002340 49 00 00 00 b4 94 04 08 3a 00 00 00 12 00 00 00 >I.......:.......<
0002360 9a 02 00 00 c4 94 04 08 53 00 00 00 12 00 00 00 >........S.......<
0002400 fd 00 00 00 d4 94 04 08 ef 00 00 00 12 00 00 00 >................<
(略)
```

符号表格具有如下结构。用ELF二进制的32位和64位分别表示的话，根据st_value对齐的不同，排列顺序会发生改变。

- 32 bit (16byte)

```
uint32_t          st_name;
Elf32_Addr        st_value;
uint32_t          st_size;
unsigned char     st_info;
unsigned char     st_other;
uint16_t          st_shndx;
```

- 64 bit (24byte)

```
uint32_t          st_name;
unsigned char     st_info;
unsigned char     st_other;
uint16_t          st_shndx;
Elf64_Addr        st_value;
uint64_t          st_size;
```

st_name表示在存储器表格上的索引，st_value是符号的值，st_size是符号的大小。st_info低4位表示符号类型等信息，表示如下：

符号类型	值	说明
STT_OBJECT	1	符号为数据对象
STT_FUNC	2	符号是执行代码
STT_SECTION	3	符号和节相关联
STT_FILE	4	符号名称是与该对象相关联的源代码的文件名
STT_COMMON	5	符号为共享数据
STT_TLS	6	符号为线程本地数据

st_info的高4位表示该符号的组合如何进行。

符号绑定（symbol binding）	值	说明
STB_LOCAL	0	局部符号
STB_GLOBAL	1	全局符号
STB_WEAK	2	弱符号

st_shndx 表示相关的节。

节（section）	值	说明
SHN_UNDEF	0	未定义
SHN_ABS	0xfff1	有绝对值的符号
SHN_COMMON	0xfff2	共享符号

看例子 /bin/ls。第 0 项的符号信息如下所示，全部为空。

```
0002240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >................<
```

接下来的符号信息如下：

```
0002260 6c 01 00 00 84 94 04 08 3c 00 00 00 12 00 00 00 >l.......<.......<
```

地址从 0002260 开始的这个字节有如下定义。最开始的 6c 01 00 00 对应着 st_name = 0x16c，所以写成 6c 01 00 00 而不是 00 00 00 01 6c。

```
st_name = 0x16c
st_value = 0x08049484
st_size = 0x3c == 60
st_info = 0x12 == (STB_GLOBAL | STT_FUNC)
st_other = 0
st_shndx = 0 == SHN_UNDEF
```

在这里，指向符号偏移量的值 st_name 是 0x16c。

从 .dynstr 节可以看出，存储器表格是从第 0xb50 个字节开始的。

```
[ 5] dynstr        STRTAB        08048b50 000b50 00047b 00  A  0  0  1
```

所以，查看 0xb50 与 0x16c 的和 0xcbc 为开头的字符串符号。可以看到如下的 readlink 字符串符号。

```
% od --skip-bytes 0xcbc --read-bytes 16 /bin/ls
0006274 72 65 61 64 6c 69 6e 6b 00 5f 5f 6f 76 65 72 66 >readlink.__overf<
0006314
```

这样一来，就可以明白对应st_name=0x16c的符号是"readlink"。这和readelf输出的以下部分相对应。

```
1: 08049484  60 FUNC  GLOBAL DEFAULT UND readlink@GLIBC_2.0 (2)
```

在readelf中，版本信息之类也可以查询，所以表示出来的字符串符号也相对集中。

再配置信息

SHT_RELA或是SHT_REL类型的节，带有再配置信息。SHT_RELA带有如下的Rela构造的表格。

```
ElfN_Addr r_offset;
uintN_t r_info;
intN_t r_addend;
```

SHT_REL包含如下的不带有r_addend的Rel结构的表格。

```
ElfN_Addr r_offset;
uintN_t r_info;
```

r_offset表示进行再配置的位置，表示节头的偏移量。r_info包含再配置的类型以及符号表格的索引等信息。而Rela，在再配置的时候通常把加法的值作为r_addend保存着。

总结

在UNIX的OS中一般都会介绍可执行文件和对象文件所使用的ELF格式的内容。虽然使用readelf命令和objdump命令可以进行方便的查询，但进行binary hack的时候就可以自己读取ELF格式，这样就可以进行各式各样的HACK。

—— *Fumitoshi Ukai*

HACK #6 静态链接库和共享库

本Hack将介绍静态链接库和共享库的区别，及各自的特征。

静态链接库

静态链接库是包含了各种程序中所使用的函数等模块的各个目标文件，这样一个集合。

编写程序的时候，把源文件分割后的每一小块编译成目标文件，最后再链接成可执行文件。在这种情况下，如果多个程序都使用的模块是多个目标文件的话，把这些目标文件当成一个块来操作就有些麻烦了。

由此想到了存档文件。它把多个目标文件归纳整理成一个文件，通过使用ar(1)命令，可以把多个目标文件都放在一个存档文件里。通过OS，由于使用了ranlib(1)可以找出该存档内的对象所提供的符号信息的散列值，所以可以高效地进行存档中提供符号的目标文件的检索（注1）。这样的存档文件被称为静态链接库。

静态链接库的编写通常如下：

```
% cc -c -o foo.o foo.c
% cc -c -o bar.o bar.c
% ar ruv libfoo.a foo.o bar.o
生成ar: libfoo.a
a - foo.o
a - bar.o
```

可以用ar命令查看库的内容。

```
% ar tv libfoo.a
rw-r--r-- 1000/1000 639 Mar 1 02:48 2006 foo.o
rw-r--r-- 1000/1000 639 Mar 1 02:48 2006 bar.o
```

链接静态库时，链接器进行以下工作：先从其他目标文件中查找未定义的符号，再从指定的静态链接库中读取定义符号的目标文件的副本，加入到可执行文件中，完成链接。

在库的目录中没有Libfoo.a时，进行如下链接：

```
% cc -o baz.o -lfoo
```

这种情况下，对于baz.o中未定义的符号，包含在libfoo.a中的对象中若存在定义该符号的对象，将其读取出来，复制到可执行文件baz中去，完成链接。

这里需要注意的是，操作是以库中的目标文件为单位进行的，以及链接时可执行文件里包含目标文件的副本。

注1：在GUN/Linux中，ranlib命令的内容和ar命令相同，运行ranlib和运行ar-s的操作完全相同。

使用链接静态库，执行生成的可执行文件时，即使没有静态链接库也可以正常进行。因为必要的代码副本包含在可执行二进制文件里了。

共享库（共有库）

共享库在可以共享这一点上与静态库有所不同。随着OS虚拟内存技术的发展，使用1个文件例如mmap(2)等，能够在多个进程中共享内存进行引用。能够有效运用这个功能的就是共享库。共享库也称共享目标。静态库是多个目标文件的存档，而共享库则把多个目标文件复制成一个巨大的目标文件中进行共享。

以前的OS存储管理，巨大文件形成后，程序运行前还需要把它们加载到内存中，所以效率不高。

现在的OS只要先设定好内存映射，就能延迟在实际中引用其存储内容的磁盘存取，因而即使形成一个巨大的对象文件，也不会出现什么问题。

制作共享库时，通常如下进行：

```
% cc -fPIC -c -o foo.o foo.c
% cc -fPIC -c -o bar.o bar.c
% cc -shared -Wl,-soname,libfoo.so.0 -o libfoo.so foo.o bar.o
```

制作共享库时，加入-shared选项，生成共享目标。另外，一般通过-Wl、-soname选项在链接器中已经指定了该共享目标的SONAME。有关内容将在后面详细说明，这个SONAME决定了执行时将链接哪个共享目标。

共享库的链接，可按与静态链接库的同样的顺序进行。

```
% cc -o baz baz.o -floo
```

只是在实际执行的操作很不一样。在这种情况下，对于baz.o中未定义的符号，如果在共享目标中已被定义，那么只要把该共享目标的SONAME设定成执行文件的NEEDED，包含在共享目标里的代码自身并不复制。

和静态链接库不同，基本上不存在“*.so中存在什么样的对象文件”这种问题。

这里需要注意的是，操作是以共享库为单位进行的，链接时，只要把需要的共享库SONAME作为NEEDED登记到可执行文件里就可以了。

执行链接了共享库的可执行文件时，动态链接加载器使用NEEDED的信息找到必要的共享库，执行的时候，操作该进程中的内存映射，可以在同样的空间里使用共享库和可执行二进制文件。因此，执行链接了共有库的可执行文件时，必须保证在系统中已经存在共享库。这是因为实际的库代码没有包含在可执行文件中，而只在存在共享库中。

文件大小

从文件大小上看，作为整个系统，共享库小一些也无妨。这是因为使用静态库时，各可执行文件要利用包含在库中的代码，这些代码被复制，所需容量便要增加，因为库代码本身不复制，只在共享库中存在，即使有很多要使用代码的可执行文件，每个可执行文件的代码部分却只是自身所必需的，所以不会发生库的部分随可执行文件数量增加的情况。

内存大小

从执行时必要的内存大小来说，在最近的OS中也是共享库占优势。这是因为，特别是PIC代码，无论配置在什么地址也不需要更改代码，从而只要把共享库输入到1个物理存储页面，即可从别的存储空间里存在的进程中共享那个共享库的存储页面。

对于库的补丁

最近多次在库中看到安全漏洞。这就需要把有漏洞的库替换成已修正的库。

使用静态库的话，只重做静态库是不够的。因为在使用库进行编译后的可执行二进制文件中，被复制的东西会被保留下来，从而需要完全重新编译一次使用静态库的可执行二进制文件。

使用共享库的话，在库中存在问题的代码，然后只要替换一下共享库就行了。当然如果是长期执行的程序，以前的共享库已经载入内存中，要引用新的共享库则需要重新启动一次。

总结

本Hack介绍了静态库与共享库的不同以及各自的特征。静态库仅仅是目标文件的存档，库链接时目标文件都被复制一次。副本包含在可执行文件里，所以没有静态库也可以进行文件的运行。共享库是归纳了目标文件的巨大目

标文件，库链接时在执行文件里只包含SONAME的引用信息。因为执行文件里只有使用哪一个库的信息所以执行时必须存在链接时所使用的共享库。

—— *Fumitoshi Ukai*

通过 ldd 查阅共享库的依赖关系

本 Hack 里将介绍查阅共享库依赖关系的方法。

共享库的依赖关系

使用共享库的可执行文件以及共享库本身，在执行它们时，其他必要的共享库的信息会记录在 ELF“动态节”NEEDED 上。

例如 /bin/ls，使用 objdump 命令，可以看到如下结果：

```
% objdump -p /bin/ls
/bin/ls         文件形式elf32-i386

程序头
(略)

动态节:
  NEEDED        librt.so.1
  NEEDED        libacl.so.1
  NEEDED        libc.so.6
  INIT          0x804945c
(略)
```

若是使用 readelf 命令，可以看到如下结果：

```
% readelf -d /bin/ls

Dynamic segment at offset 0x12184 contains 22 entries:
 标签 (tag).   类型                    名称 / 值
 0x00000001   (NEEDED)                共享库: [librt.so.1]
 0x00000001   (NEEDED)                共享库: [libacl.so.1]
 0x00000001   (NEEDED)                共享库: [libc.so.6]
 0x0000000c   (INIT)                  0x804945c
```

可以看出这样的 /bin/ls，librt.so.1，libacl.so.1，libc.so.6这三个共享库是必需的。

不过执行 /bin/ls，只有这3个共享库是不行的。这3个共享库本身也各自需要其他的必需的共享库。

NEEDED里记录的是SONAME，所以需要从SONAME中找到实际的文件。特别是之前未设定的情况下，在/usr/lib和/lib中存在对应其SONAME的文件的话，那就是其共享库了（注2）。在环境变量LD-LIBRARY-PATH里如果设定了库的路径，则引用该库。若是在/etc/ld.so.cache里有信息，则引用此信息。/etc/ld.so.cache在使用/etc/ld.so.conf的设置来执行ldconfig时会被更新一次。

例如librt.so.1因为存在/lib/librt.so.1文件（symbolic link），它即是对应SONAME librt.so.1的共享库文件。这个共享库本身依赖的库也同样可以在objdump和readelf中查看到。

```
% readelf -d /lib/librt.so.1

Dynamic segment at offset 0x61b8 contains 25 entries:
标签        类型                         名称/值
0x00000001 (NEEDED)                     共享库:[libc.so.6]
0x00000001 (NEEDED)                     共享库:[libpthread.so.0]
0x00000001 (NEEDED)                     共享库:[ld-linux.so.2]
0x0000000e (SONAME)                     库的soname: [librt.so.1]
0x0000000c (INIT)                       0x177c
(略)
```

不难看出，在这样的librt.so.1里面，libc.so.6，libpthread.so.0，ld-linux.so.2是必需的。

同样的，查阅其他共享库如下：

```
/bin/ls NEEDED librt.so.1  NEEDED libc.so.6
                           NEEDED libpthread.so.0  NEEDED libc.so.6
                                                   NEEDED ld-linux.so.2
                           NEEDED ld-linux.so.2
        NEEDED libacl.so.1 NEEDED libattr.so.1     NEEDED libc.so.6
                           NEEDED libc.so.6
        NEEDED libc.so.6   NEEDED ld-linux.so.2
```

因此，可以看出，要执行/bin/ls，必须的共享库有librt.so.1，libacl.so.1，libc.so.6，libpthread.so.0，libattr.so.1，ld-linux.so.2。

使用ldd查阅共享库的依赖关系

如上所述，使用objdump和readelf查询共享库的依赖关系不是不可能，但是因为要查看与各个共享库相关的全部依赖关系，这样就很麻烦了。另外，

注2：　实际上，有时也会用到/lib/tls和/lib/tls/i686/cmov等中的共享库。

实际执行时，在使用哪个目录上的共享库这一点上，并不一定能得到正确的结果。

使用ldd命令的话，能集中完成以上的执行操作。

```
% ldd /bin/ls
    librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7fd2000)
    libacl.so.1 => /lib/libacl.so.1 (0xb7fcb000)
    libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e96000)
    libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e86000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb7fea000)
    libattr.so.1 => /lib/libattr.so.1 (0xb7e82000)
```

这样的话可执行文件所需的共享库中的SONAME以及它的路径名和所分配的存储地址，都全部表现出来。

对于可执行文件以外的共享库，ldd也可以使用。

```
% ldd /lib/librt.so.1
  libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ea7000)
  libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e97000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

在GUN/Linux里，ldd实际上仅是shell脚本。重点是环境变量LD-TRACE-LOADED-OBJECTS。把环境变量LD-TRACE-LOADED-OBJECTS设定为1后执行程序，开始执行程序时解释器（Runtime loader /bib/ld.linux.so.2）将在执行实际的程序之前查看程序必要的共享库，将其载入内存并把它的信息显示出来。因此，不用ldd，只用环境变量LD-TRACE-LOADED-OBJECTS，也可以得到同样的结果。

```
% LD_TRACE_LOADED_OBJECTS=1 /bin/ls
  librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7fd2000)
  libacl.so.1 => /lib/libacl.so.1 (0xb7fcb000)
  libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e96000)
  libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e86000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb7fea000)
  libattr.so.1 => /lib/libattr.so.1 (0xb7e82000)
```

如果不是可执行文件，共享库也不能执行，不能进行同样的操作。

```
% LD_TRACE_LOADED_OBJECTS=1 /lib/librt.so.1
zsh: 没有许可：/lib/librt.so.1
```

这时，执行Runtime loader /lib/ld.linux.so.2。

```
% LD_TRACE_LOADED_OBJECTS=1 /lib/ld-linux.so.2 /lib/librt.so.1
  libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ea7000)
```

```
libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e97000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

总结

在共享库的依赖关系中，通过可执行文件以及记录在共享库的ELF的动态节的NEEDED上的信息，管理所需共享库的SONAME。各个文件的NEEDED虽然可以通过使用objdump和readelf查看，但要显示出贯穿依赖关系的所有共享库，则要使用ldd命令，ldd命令使用环境变量LD-TRACE-LOADED-OBJECTS实现这个过程。

—— *Fumitoshi Ukai*

用 readelf 表示 ELF 文件的信息

本 Hack 将介绍显示 ELF 文件信息的工具 readelf 的有关内容。

readelf是不通过BFD库而直接读取ELF的工具。由于存在不依赖BFD的程序，就可以轻易区分是ELF文件的问题还是BFD的问题。readelf因为是不经过BFD查看ELF文件，就可以得到比objdump更加详细的信息。比如，可以查看DWARF调试信息等。readelf要读出哪条信息，必须指定任意选项中的一个，没有选项的时候，就会显示出使用帮助。

读出 ELF 头

读出ELF头的选项如下

需要的头	选项（option）	long option
ELF文件头	-h	--file-header
程序头	-l	--program-headers, --segments
节头	-S	--section-headers, --sections
以上3个头	-e	--headers

读出 ELF 信息

需要的信息	选项	long option
符号表	-s	--syms, -symbols
再配置信息	-r	--relocs

需要的信息	选项	long option
动态段	-d	--dynamic
版本信息	-V	--version-info
依赖设计	-A	--arch-specific
长的直方图	-I	--histogram
所有的头和以上全部	-a	--all
核标记	-n	--notes
unwind 信息	-u	--unwind

符号信息通常使用符号节里的符号信息，但在使用-D选项（--use-dynamic选项）时，则要使用 dynamic 节里的符号信息。

ELF 节的转储处理（dump）

用 -x 选项（--hex-dump 选项）可以转储指定节的内容。

节通过节编号指定，节编号即是用 -S 选项表示的、跟在节头后面的数字。

```
% readelf -S /bin/ls
25 个节头，起点偏移量 0x124c4:

Section Headers:
[号码] 名称              类型        地址      Off    大小   ES Flg Lk Inf Al
[ 0]                    NULL        00000000 000000 000000 00      0   0   0
[ 1] .interp            PROGBITS    08048134 000134 000013 00  A   0   0   1
[ 2] .note.ABI-tag      NOTE        08048148 000148 000020 00  A   0   0   4
[ 3] .hash              HASH        08048168 000168 000338 04  A   4   0   4
[ 4] .dynsym            DYNSYM      080484a0 0004a0 0006b0 10  A   5   1   4
[ 5] dynstr             STRTAB      08048b50 000b50 00047b 00  A   0   0   1
（略）
```

.interp 的节编号是 1，为了阅读它的内容，要进行以下操作。

```
% readelf -x1 /bin/ls

'.interp' 节的十六进制转储
 0x08048134 6f732e78 756e696c 2d646c2f 62696c2f /lib/ld-linux.so
 0x08048144                                    00322e .2.
```

.note.ABI-tag 的节编号是 2，.hash 的节编号是 3。

读出 DWARF2 的调试节

-w选项表示（--debug-dump选项）DWARF2调试节的信息。（关于DWARF2请参照 [Hack #40]）

-w	--debug-dump=	选项
l	line	.debug_line
i	info	.debug_line
a	abbrev	.debug_abbrev
p	pubnames	.debug_pubnames
r	aranges	.debug_aranges
R	Ranges	.debug_ranges
m	macro	.debug_macinfo
f	frames	.debug_frame
F	frames-interp	debug_frame
s	str	.debug_str
o	loc	.debug_loc

长符号也都能表示

在缺省时，为了把长符号的书写控制在一行以内，要把后面的剪掉。

使用-w选项（--wide选项）的话，就可以进行长度超过80个字符的输出。

总结

使用readelf，可解释ELF和DWARF的信息，并将它们表示出来。

—— *Fumitoshi Ukai*

用objdump来转储目标文件

在本Hack中，将介绍转储目标文件工具 —— objdump的使用方法。

用objdump转储ELF Binary

对于objdump而言，有必要指定所显示信息的选项。只是单纯地进行转储时，要使用-s 选项（--full-contents选项）。

```
% objdump -s /bin/ls
/bin/ls:     文件形式elf32-i386
节 .interp的内容:
8048134  2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
8048144  2e3200                             .2.
```

```
节       .note.ABI-tag的内容：
8048148  04000000 10000000 01000000 474e5500 ............GNU.
8048158  00000000 02000000 02000000 00000000 ................
节       .hash的内容：
8048168 61000000 6b000000 00000000 3d000000 a...k.......=...
8048178 3c000000 31000000 00000000 00000000 <...1...........
（略）
```

像这样，把指定的ELF二进制文件的形式以及各节的内容进行转储，并显示出来。各节的转储，以如下的形式输出。

```
内存地址 16进制转储(4 字节 * 4)     ASCII 表示
```

在这里，即使是运行在如x86这样little endian的架构上，十六进制也用big endian的方式输出。例如在上述情况中，interp最初的4字节用ASCII表示是"/lib"，它相应的十六进制是2f 6c 69 62而不是little endian。虽然objdump有--endian选项，但这个选项只影响objdump上的disassemble（反汇编）阶段，不影响其输出。

像这种指定了-s选项的情况下，结果就变为以目标格式（target format）进行转储。通常以elf32-i386的形式，所以在转储时每个节都要首先进行识别。这个目标格式通过指定-b 选项（--target选项）来更改。

通过-i选项（--info选项）来对目标格式进行查询。

```
% objdump -i
BFD头文件版本 2.15
elf32-i386
 (header little endian, data little endian)
 i386
a.out-i386-linux
 (header little endian, data little endian)
 i386
（略）
```

用 objdump 只对 ELF Binary 的特定节进行转储。

只想转储特定节的话，则用-j选项（--section选项）指定节名即可。

```
% objdump -s -j .interp /bin/ls
/bin/ls:     文件形式 elf32-i386
节.interp的内容：
  8048134 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
  8048144 2e3200                              .2.
%
```

为了查询节的内容，可以只用-s选项表示“节××的内容：”的那一行，也可以选用-h选项（--section-headers选项、--headers选项）来表示。

```
% objdump -h /bin/ls
/bin/ls:     文件形式 elf32-i386
   节
  索引名          大小      VMA       LMA       File off Algn
0 .interp         00000013 08048134 08048134  00000134 2**0
        CONTENTS, ALLOC, LOAD, READONLY, DATA
1 .note.ABI-tag   00000020 08048148 08048148  00000148 2**2
        CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .hash           00000338 08048168 08048168  00000168 2**2
        CONTENTS, ALLOC, LOAD, READONLY, DATA
(略)
```

这里的索引名（.interp 和 .note.ABI-tag 等）也可以作为节名使用。

用 objdump 对地址进行指定和转储

使用--start-address选项和--stop-address选项，可以指定转储地址范围。例如：可以看到上面输出的 .interp 的地址是从 0x08048134 到 0x08048147，也可以进行如下操作：

```
% objdump -s --start-address=0x08048134 --stop-address=0x08048147 /bin/ls
/bin/ls:     文件形式 elf32-i386
节 .interp 的内容：
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
 8048144 2e3200 .2.
节 .note.ABI-tag 的内容：
节 .hash 的内容：
节 .dynsym 的内容：
节 .dynstr 的内容：
节 .gnu.version 的内容：
节 .gnu.version_r 的内容：
节 .rel.dyn 的内容：
节 .rel.plt 的内容：
节 .init 的内容：
节 .plt 的内容：
节 .text 的内容：
节 .fini 的内容：
节 .rodata 的内容：
节 .eh_frame_hdr 的内容：
节 .data 的内容：
节 .eh_frame 的内容：
节 .dynamic 的内容：
节 .ctors 的内容：
节 .dtors 的内容：
节 .jcr 的内容：
节 .got 的内容：
%
```

跨节指定地址等情况，会变成以下格式：

```
% objdump -s --start-address=0x08048134 --stop-address=0x08048150 /bin/ls
/bin/ls:     文件形式 elf32-i386
节 .interp 的内容:
8048134 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
8048144 2e3200                              .2.
节 .note.ABI-tag 的内容:
8048148 04000000 10000000                   ........
节 .hash 的内容:
节 .dynsym 的内容:
节 .dynstr 的内容:
节 .gnu.version 的内容:
节 .gnu.version_r 的内容:
节 .rel.dyn 的内容:
节 .rel.plt 的内容:
节 .init 的内容:
节 .plt 的内容:
节 .text 的内容:
节 .fini 的内容:
节 .rodata 的内容:
节 .eh_frame_hdr 的内容:
节 .data 的内容:
节 .eh_frame 的内容:
节 .dynamic 的内容:
节 .ctors 的内容:
节 .dtors 的内容:
节 .jcr 的内容:
节 .got 的内容:
%
```

用 objdump 转储简单二进制文件

把原本不是ELF 的文件，以及把ELF 文件当作简单二进制文件进行转储时，将目标格式指定为 binary。

```
% objdump -s -b binary /bin/ls
/bin/ls:     文件形式 binary
节 .data 的内容:
00000 7f454c46 01010100 00000000 00000000 .ELF............
00010 02000300 01000000 509a0408 34000000 ........P...4...
00020 c4240100 00000000 34002000 08002800 .$......4. ...(.
00030 19001800 06000000 34000000 34800408 ........4...4...
00040 34800408 00010000 00010000 05000000 4...............
00050 04000000 03000000 34010000 34810408 ........4...4...
00060 34810408 13000000 13000000 04000000 4...............
00070 01000000 01000000 00000000 00800408 ................
00080 00800408 081d0100 081d0100 05000000 ................
(略)
```

这样文件格式变成 binary，按节分割则不能输出。

不是ELF文件的话将变成以下的形式。

```
% objdump -s -b binary  /etc/ld.so.cache
/etc/ld.so.cache:    文件形式 binary
节 .data 的内容：
0000 6c642e73 6f2d312e 372e3000 18030000 ld.so-1.7.0.....
0010 03000000 204f0000 2a4f0000 03000000 .... O..*O......
0020 3d4f0000 454f0000 03000000 564f0000 =O..EO......VO..
0030 674f0000 03000000 814f0000 8e4f0000 gO.......O...O..
0040 03000000 a44f0000 b24f0000 03000000 .....O...O......
0050 c94f0000 d54f0000 03000000 ea4f0000 .O...O.......O..
0060 f84f0000 03000000 0f500000 22500000 .O.......P.."P..
0070 03000000 3e500000 4a500000 03000000 ....>P..JP......
0080 5f500000 6c500000 03000000 82500000 _P..lP.......P..
（略）
```

二进制文件是不能自标识的，所以必须指定选项。

```
% objdump -s /etc/ld.so.cache
objdump: /etc/ld.so.cache: 不能识别文件格式
```

这种情况下，地址和文件格式对等。

总结

在本 Hack 中，说明了用 objdump 对 ELF 二进制文件以及普通的文件进行转储的方法。

—— *Fumitoshi Ukai*

用 objdump 反汇编目标文件

在本 Hack 中，介绍了用 objdump 来反汇编目标文件的方法。

用 objdump 来反汇编目标文件

objdump不仅可以转储目标文件，在ELF binary中还可以反汇编目标文件。在反汇编文件时使用 -d 选项（-disassemble 选项）

```
% objdump -d hello.o

hello.o:    文件形式 elf32-i386

节 .text 的反汇编：

00000000 <main>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp, %ebp
```

```
 3: 83 ec 08                sub    $0x8, %esp
 6: 83 e4 f0                and    $0xfffffff0, %esp
 9: b8 00 00 00 00          mov    $0x0, %eax
 e: 29 c4                   sub    %eax, %esp
10: c7 04 24 00 00 00 00    movl   $0x0, (%esp)
17: e8 fc ff ff ff          call   18 <main+0x18>
1c: c7 04 24 00 00 00 00    movl   $0x0,(%esp)
23: e8 fc ff ff ff          call   24 <main+0x24>
```

像这样在使用 -d 选项时，通常将可执行代码形式的文件部分（.text 等）作为反汇编的对象。若要把所有的节都作为反汇编的对象，则可使用 -D 选项（--disassemble-all选项）。这时，会将 .debug_abbrev等非代码部分也当作代码来解释，然后输出反汇编的结果。

反汇编结果通常如以下所示

```
地址 <符号 >:
地址：代码的字符串    反汇编代码
```

若不需要代码的字符串时，则用 --no-show-raw-insn 选项。

另外，使用--prefix-address选项，反汇编的地址会和符号的相对地址同时输出。这种情况下会自动转化为 --no-show-raw-insn。

```
% objdump -d --prefix-address hello.o

hello.o:     文件形式 elf32-i386

节 .text 的反汇编：
00000000 <main> push   %ebp
00000001 <main+0x1> mov    %esp,%ebp
00000003 <main+0x3> sub    $0x8,%esp
00000006 <main+0x6> and    $0xfffffff0,%esp
（略）
```

顺便提一下，在 --prefix-address 中，还想查阅代码的字符串的话，可以同时使用 --show-raw-insn 选项。

```
% objdump -d --prefix-address --show-raw-insn hello.o

hello.o:     文件形式 elf32-i386

节 .text 的反汇编：
00000000 <main> 55                      push %ebp
00000001 <main+0x1> 89 e5               mov %esp,%ebp
00000003 <main+0x3> 83 ec 08            sub $0x8,%esp
00000006 <main+0x6> 83 e4 f0            and $0xfffffff0,%esp
（略）
```

用 objdump 反汇编特定的节、地址范围

指定节也可以只对指定的节进行反汇编。节的指定与转储一样，只需使用 -j 选项（--section选项）即可。

```
% objdump -d -j .init hello

hello:     文件形式elf32-i386

节 .init 的反汇编：

 0 804829c <_init>:
   804829c:     55                  push %ebp
   804829d:     89 e5               mov %esp, %ebp
   804829f:     83 ec 08            sub $0x8, %esp
   80482a2:     e8 7d 00 00 00      call 8048324 <call_gmon_start>
   80482a7:     e8 e4 00 00 00      call 8048390 <frame_dummy>
   80482ac:     e8 ff 01 00 00      call 80484b0 <__do_global_ctors_aux>
   80482b1:     c9                  leave
   80482b2:     c3                  ret
```

地址范围与转储一样，用 --start-address选项和 --stop-address选项就可以指定。

与源文件对照

目标文件里含有调试信息的情况下，使用-l选项（--line-numbers选项），会输出各种汇编代码与源代码的行号相对应的信息。在不含调试信息的情况下使用 -l 选项无效。

```
% objdump -d -l hello.o

hello.o:     文件形式elf32-i386

节 .text 的反汇编：

00000000 <main>:
main():
/tmp/hello.c:5
   0:   55                      push %ebp
   1:   89 e5                   mov %esp,%ebp
   3:   83 ec 08                sub $0x8,%esp
   6:   83 e4 f0                and $0xfffffff0,%esp
   9:   b8 00 00 00 00          mov $0x0,%eax
   e:   29 c4                   sub %eax,%esp
/tmp/hello.c:6
  10:   c7 04 24 00 00 00 00    movl $0x0,(%esp)
  17:   e8 fc ff ff ff          call 18 <main+0x18>
/tmp/hello.c:7
  1c:   c7 04 24 00 00 00 00    movl $0x0,(%esp)
  23:   e8 fc ff ff ff          call 24 <main+0x24>
```

另外，指定-S选项（--source选项）的话，若其源文件存在，将在它里面插入表达与-l选项的行号码相对应的源代码。

```
% objdump -d -S hello.o

hello.o:      文件形式elf32-i386

节.text 的反汇编:

00000000 <main>:
#include <stdio.h>

int
main(int argc, char *argv[])
{
   0:   55                      push %ebp
   1:   89 e5                   mov %esp,%ebp
   3:   83 ec 08                sub $0x8,%esp
   6:   83 e4 f0                and $0xfffffff0,%esp
   9:   b8 00 00 00 00          mov $0x0,%eax
   e:   29 c4                   sub %eax,%esp
     printf("Hello, world\n");
  10:   c7 04 24 00 00 00 00    movl $0x0,(%esp)
  17:   e8 fc ff ff ff          call 18 <main+0x18>
     exit(0);
  1c:   c7 04 24 00 00 00 00    movl $0x0,(%esp)
  23:   e8 fc ff ff ff          call 24 <main+0x24>
```

当然，也可以同时使用-S选项和-l选项。-S选项也与 -l选项一样，对于不含调试信息的目标文件无效。目标文件里的调试信息，由于它含有源代码的路径名与行号码，所以它的源文件必须放在由它的路径所指定的地方。若没有相对应的源代码，源程序则无法表示，若放入不同的源程序的的话，也将输出不同源程序的行。

在链接前的目标文件中，再配置的地址会有变成0的情况。例如："Hello, world\n"的地址，本应该存在于13~16的4字节里，但因为处于未连接状态，所以一直为0。

```
c7 04 24 00 00 00 00
```

在链接好的可执行文件相应的地方，存入了如下地址。

```
c7 04 24 04 85 04 08

% objdump -d --start-address=0x080483c4 --stop-address=0x80483ec
-S hello
(略)
80483d4:        c7 04 24 04 85 04 08 movl $0x8048504,(%esp)
(略)
```

总结

objdump可以用于目标文件和可执行文件的反汇编。若有源代码的话，还可混合进行相应源程序的反汇编并输出。

—— *Fumitoshi Ukai*

用 objcopy 嵌入可执行文件的数据

在本 Hack 中，介绍了用 objcopy 嵌入可执行文件中数据的方法。

有时候为了程序的运行必须要从文件中提取必要的数据并加以利用。用这种方法，可以方便地进行数据的更新，但是不能运行单独的可执行文件，有可能导致数据文件的丢失。

在本 Hack 中，介绍了用 objcopy 对可执行文件中的数据进行嵌入的方法。

数据的嵌入

将少量数据存储在源代码中是很简单的。包含在源代码中像 "hello, world" 这样的信息就可以称为存储在源代码中的数据。

但是，要将图像或字典这样庞大的数据输入源代码就不那样简单了。首先，要将数据转换为字符串等，变换后庞大的源代码可能会超出编译器所能处理的范围。

objcopy

在此介绍的是附属在 GNU binutils 中的 objcopy 命令。使用 objcopy 可以将任意文件转换为可以链接的目标文件。

例如：可以用如下方式将 foo.jpg 转换为 x86 用的 ELF32 形式的目标文件 foo.o 。

```
% objcopy -I binary -O elf32-i386 -B i386 foo.jpg foo.o
```

foo.jpg 的数据可以引用以下的链接了 foo.o 的 C 程序的变量名。

```
extern char _binary_foo_jpg_start[];
extern char _binary_foo_jpg_end[];
extern char _binary_foo_jpg_size[];
```

这些变量的使用方法如下所示：

```
const char *start = _binary_foo_jpg_start;  // 获得数据的起始地址
const char *end = _binary_foo_jpg_end;  // 获得数据的末尾地址+1
int size = (int)_binary_foo_jpg_size;  // 获得数据大小
```

要注意的是，最后的_binary_foo_jpg_size中的&_binary_foo_jpg_size[0]表示的不是地址而是值（数据的大小）。

总结

介绍了用objcopy存储可执行文件中数据的方法。不仅仅是像本Hack中举出的那些普通的数据，如果还能将源代码本身或是其他程序的二进制等奇妙的东西也能存储的话就非常有意思了。

—— *Satoru Takabayashi*

用 nm 检索包含在目标文件里的符号

在本节中将会介绍用于查看包含在对象文件中的符号的工具——nm的使用方法。

nm 的使用方法

对目标文件中使用nm，就可以将包含在目标文件中的符号一一列出。

```
% nm cabin.o
00003530 t .L1207
00003557 t .L1208
0000356c t .L1209
00003581 t .L1210
00003596 t .L1211
000035ab t .L1212
000044b9 t .L1427
00004561 t .L1428
```

nm可以将包含在目标文件中的符号按字母表的顺序一行行的输出。缺省的话则为bsd输出格式，bsd格式的每行将分别输出符号的值、符号类型以及符号名。出现值未定义的符号（符号类型为U）时，将不输出符号值，表现为空格。

用-r选项（--reverse-sort选项）可以进行倒序输出。

```
% nm -r cabin.o
         U write
         U times
         U time
         U sysconf
         U strstr
         U strrchr
         U strlen
```

使用了--size-sort选项则表示将符号的大小（该符号所表示的对象的大小）按从小到大的顺序排列。同时使用了-r选项则表示按从大到小的顺序进行排列。

```
% nm --size-sort cabin.o
00000001 T cbstdiobin
00000004 d asiz.7703
00000004 B cbfatalfunc
00000004 b farray.7701
00000004 b onum.7702
00000004 b parray.7699
00000007 T cbdatumptr
...
% nm --size-sort -r cabin.o
00000b5b T cbstrmktime
00000637 T cbmimebreak
000004ec T cburlbreak
0000036a T cbhsort
00000365 T cbxmlbreak
0000033f T cbdatestrhttp
0000033a T cbsprintf
000002b2 T cbmapputcat
0000026d T cbxmlattrs
...
```

但是，不能输出未定义符号。在缺省的输出格式中，若指定了--size-sort选项，则在最开始的一栏里输出符号大小。在按大小分类表示符号值的时候，要同时使用-S选项。此时，每行按符号的值、符号大小、符号类型、符号名称的顺序排列。这样便可以查询包含在目标文件里的庞大的函数或数据。

```
% nm --size-sort -r -S cabin.o
00007070 00000b5b T cbstrmktime
000053e0 00000637 T cbmimebreak
00005a20 000004ec T cburlbreak
00003970 0000036a T cbhsort
000035d0 00000365 T cbxmlbreak
00004350 0000033f T cbdatestrhttp
00006d30 0000033a T cbsprintf
00004c80 000002b2 T cbmapputcat
00005170 0000026d T cbxmlattrs
```

输出格式可用-f选项（--format选项）来指定。除了缺省的bsd之外还有sysv和posix。

```
% nm -f sysv foo.o
foo.o中的符号:

Name                  Value   Class   Type         Size    Line  Section
change              |0000a130|  T  |      FUNC|0000009d|       |.text
check_buffer        |000081a0|  t  |      FUNC|00000045|       |.text
check_type          |        |  U  |    NOTYPE|        |       |*UND*
check_target        |00000124|  d  |    OBJECT|00000004|       |.data
clear_buffer        |000000f0|  D  |    OBJECT|00000004|       |.data

% nm -f posix foo.o
change T 0000a130 0000009d
check_buffer t 000081a0 00000045
check_type U
check_target d 00000124 00000004
clear_buffer D 000000f0 00000004
```

在目标文件被多个参数指定时，先在每个目标文件里分别将目标文件的符号分类再输出。若加入了-A选项，（或是-o选项、--print-file-name选项）因为每行的最开始的符号在输出时包含了所在文件的文件名，所以在用grep查找符号时，就可以轻易地找出该符号包含在哪个目标文件中。

```
% nm -A *.o | grep check_target
foo.o:00000124 d check_target
```

即使是在可执行文件中使用nm，也可以检索符号。

```
% nm a.out
080494e4 D _DYNAMIC
080495c0 D _GLOBAL_OFFSET_TABLE_
080484c0 R _IO_stdin_used
         w _Jv_RegisterClasses
080495b0 d __CTOR_END__
....
```

但是，若在nm中省略了变量时，当前目录中的a.out则变为指定的对象。a.out是生成的可执行二进制文件的缺省文件名。在目录中没有a.out时，程序则会报错并终止运行。

使用strip(1)去掉可执行文件中的符号之后，用nm查看则表示如下：

```
% strip foo
% nm foo
nm: foo:没有符号
```

但是，在不是静态二进制文件的情况下，会保留为动态链接共享库的符号。由于这些符号是动态的符号，所以有必要使用 -D 选项（--dynamic 选项）。

```
% nm -D foo
080484c0 R _IO_stdin_used
         w _Jv_RegisterClasses
         w __gmon_start__
         U __libc_start_main
         U printf
```

关于静态链接库，会输出包含在库文件中的每个目标文件符号。

```
% nm /usr/lib/libc.a

init-first.o:
         U __environ
         U __fpu_control
         U __init_misc
00000004 C __libc_argc
00000004 C __libc_argv
00000090 T __libc_init_first
         U __libc_init_secure
00000000 D __libc_multiple_libcs
         U __setfpucw
         U _dl_non_dynamic_init
000000a0 T _dl_start
         w _dl_starting_up
         U abort
00000000 t init

libc-start.o:
         U __close
         U __cxa_atexit
         U __environ
```

在这种情况下，如果同时使用了 -A 选项的话，则在输出每行时会标明是哪个库的哪个目标文件。

```
% nm -A /usr/lib/libc.a
/usr/lib/libc.a:init-first.o:          U __environ
/usr/lib/libc.a:init-first.o:          U __fpu_control
/usr/lib/libc.a:init-first.o:          U __init_misc
/usr/lib/libc.a:init-first.o:00000004 C __libc_argc
/usr/lib/libc.a:init-first.o:00000004 C __libc_argv
/usr/lib/libc.a:init-first.o:00000090 T __libc_init_first
```

若是共享库，则看起来会像一个庞大的目标文件。

```
% nm libqdbm.so.11.5.0
00005588 t .L10
0000d148 t .L10
```

```
0000558f t .L11
0000d14f t .L11
00005596 t .L12
0000d156 t .L12
```

共享库和可执行二进制文件一样，在安装时多会被删除。

在系统里安装共享库时通常会输出如下。

```
% nm /usr/lib/libqdbm.so.11.5.0
nm: /usr/lib/libqdbm.so.11.5.0: 没有符号
```

在这种情况下也可以使用 -D 选项来查看动态符号。

```
% nm -D /usr/lib/libqdbm.so.11.5.0
00027028 D VL_CMPDEC
00027020 D VL_CMPINT
0002701c D VL_CMPLEX
00027024 D VL_CMPNUM
0002703c D VST_CMPDEC
00027034 D VST_CMPINT
00027030 D VST_CMPLEX
00027038 D VST_CMPNUM
00027068 A _DYNAMIC
00027154 A _GLOBAL_OFFSET_TABLE_
         w _Jv_RegisterClasses
```

符号类型

想要读取nm的输出，就必须要分清符号类型。符号有很多种类型，在nm中用一个字符表示。各种符号类型如下所示。大字符和小字符在含义上有区别，大字符表示全局（可供外部引用）的符号，小字符则表示局部文件符号。

若按字母表顺序排列则如下所示：

符号类别	说明
A	符号值为常量值，在链接时不会发生变化
B	符号在未初始化区域（BSS）里
C	公共符号，未初始化的数据
D	符号在已初始化的数据节里
G	符号在可使用的已初始化后的数据节里(附近的符号可以被高效的存取)
I	被别的符号间接引用的符号。a.out 的 GNU 扩展
N	调试用的符号

符号类别	说明
R	专门用于读取数据节里的符号
S	符号位于能在查询到的未初始化的数据节里
T	代码节里的符号
U	未定义符号，在其他目标文件或共享库中可能存在符号定义
V	弱对象符号
W	不能决定为弱对象符号的弱符号
-	a.out 目标文件里的 stabs 符号（调试信息等）
?	未知的符号类型

按节分类则如下所示：

节	符号类别	作用范围
文本节	T	全局
	t	局部
数据节	D	全局
	G	全局（用于小对象）
	d	局部
	g	全局（用于小对象）
专用读取数据	R	全局
	r	局部
BSS（未初始化数据）	B	全局
	S	全局（用于小对象）
	b	局部
	s	局部（用于小对象）
弱对象	V	全局
	v	局部
弱符号	W	全局
	w	局部
公共	C	全局
调试用	N	全局
	n	局部
	-	stabs

节	符号类别	作用范围
常量值	A	全局
	a	局部
未定义	U	全局
间接引用	I	全局
	i	局部
未知类型	?	

总结

在本 Hack 中，对用于查询包含在目标文件中符号的基本工具 nm 进行了说明。

—— *Fumitoshi Ukai*

HACK #13 用 strings 从二进制文件中提取字符串

在本节中介绍了 strings 的使用方法及组成

strings 是用于从二进制文件中提取字符串的工具，包含在 GNU Binutils 中。在本 Hack 中，对 strings 的使用方法及其组成进行了介绍。

strings 的使用方法

strings 的基本用法很简单。只要把从二进制文件中取出的字符串转换为变量就可以了。至于二进制文件，可以是像 /bin/ls 这样的可执行文件，也可以是像 foo.jpg、bar.mp3 这样的任意二进制文件。

```
% strings /bin/ls | head -5
/lib/ld-linux.so.2
librt.so.1
clock_gettime
_Jv_RegisterClasses
__gmon_start__
```

也可以从标准输出中读取二进制数据。strings 和 grep 命令结合起来的话会更方便，在下面的例子中正在检索包含 ignoring 的错误信息。

```
% cat /bin/ls | strings | grep ignoring
ignoring invalid tab size in environment variable TABSIZE: %s
ignoring invalid width in environment variable COLUMNS: %s
ignoring invalid value of environment variable QUOTING_STYLE: %s
```

若使用了 -tx 选项则用16进制来表示字符串的位置。若想使用10进制则指定 -td，8进制则指定 -to，关于其他的选项，请参考 man strings 上的使用手册。

```
% strings -tx /bin/ls | head -5
  134 /lib/ld-linux.so.2
  b51 librt.so.1
  b5c clock_gettime
  b6a _Jv_RegisterClasses
  b7e __gmon_start__
```

字符串的判定

strings 在缺省时以“可用ASCII表示的字符（7位）构成的4字节以上的可打印字符串”的规则进行字符串的处理。因此，在程序里输入的简体中文字符串（8位）在缺省时就无法用 UTF-8 显示。

```
const char *p = "简体中文的信息";
```

要表示 UTF-8 的字符串的话，则必须使 strings 通过 -eS 选项。-e 是 encoding（编码），S 代表 8 位的意思。但是，这样就可能产生许多垃圾字符串，所以在用 -n 选项表示时，有必要增加必要的字节数（缺省时为 4）。

目标文件的处理

strings 在内部对 BFD 库是否为可解释的目标文件（可执行文件和库等）进行判断，若是目标文件则将每个数据节作为选取字符串的对象，这是因为包含在 C 程序里的字符串通常以以下方式包含在数据节里。

```
const char *p = "hello, world";
```

通过指定 -a 或 - 可以强行将文件整体作为对象。若该文件不是目标文件的话则该文件整体成为对象。而且，从标准输入读取时，由于依据 BFD 不能进行判定，文件整体也将成为对象。利用 -T 选项对系统标准以外的目标文件的格式进行指定。

总结

在本 Hack 里介绍了 strings 的使用方法及其组成。strings 可进行错误信息的检索，可用于简单的程序解析，是一个非常有用的工具。

—— *Satoru Takabayashi*


用 C++filt 对 C++ 的符号进行转储

在本 Hack 中，将介绍根据命令系统对 C++ 符号进行转储的方法——nm-demangle 和 c++filt。

C++编译器对符号进行名称递变换的处理，让每个符号只拥有一种名称。在本Hack中，介绍了从命令系统中将C++ 符号进行名称逆变换的方法。运行时的名称逆变换方法请参照“[Hack #68]C++ 符号运行时的名称逆变换”。

nm 的使用方法

如果对 C++ 目标文件使用 nm，则在缺省时会以变换过的难以辨认的方式进行符号输出。

```
% nm foo.o
00000000 T _Z3fooi
```

想要读懂它的话，就要在管道上使用 c++filt，或在 nm 中使用 --demangle 选项。c++filt 不仅可以用于 nm 的输出，还可以作为多用的过滤器，所以非常方便。

```
% nm foo.o | c++filt
00000000 T foo(int)

% nm --demangle foo.o
00000000 T foo(int)
```

总结

在本 Hack 中把 nm --demangle 和 c++filt 作为从命令系统中将 C++ 符号进行名称逆变换的方法进行了介绍。nm 和 c++filt 都是 GNU Binutils 里含有的工具。

—— *Satoru Takabayashi*


用 addr2line 从地址中获取文件名和行号

addr2line 是从地址中获取文件名和行号的工具，主要用于调试。

addr2line 的使用方法

addr2line是利用调试信息来获取文件名和行号信息的。因此，程序在编译时有必要附上调试信息，在 GCC 中指定为 -g 选项。

假设有以下程序 test.c：

```
#include <stdio.h>
void func() {
}

int main() {
    printf("%p\n", &func);
    return 0;
}
```

将其附上调试信息进行编译，运行后便可出现函数 func 的地址。

```
% gcc -g test.c
% ./a.out
0x8048364
```

使用addr2line，便可从地址中获取相应的文件名和行号码。在-e选项上，指定了对象的可执行文件。

```
% addr2line -e a.out 0x8048364
/tmp/test.c:2
```

的确在 test.c 的第 2 行显示出来了。加上 -f 选项后也可知道函数名。

```
% addr2line -f -e a.out 0x8048364
func
/tmp/test.c:2
```

addr2line在标准输入中可进行地址的传送。这在对多个地址进行处理时很方便。

addr2line 的组成

addr2line在获取调试信息时用到了 BFD。在“[Hack #67]用libbfd获取符号信息”中有关于addr2line以及自动进行相同处理的介绍。

总结

在本 Hack 中，介绍了用addr2line从地址中获取文件名和行号的方法，这是一种有用的调试技巧。

—— *Satoru Takabayashi*

HACK #16 用 strip 删除目标文件中的符号

本节中介绍了 strip 的使用方法

strip是用于删除目标文件中符号的工具，通常用于删除已生成的可执行文件和库中不需要的符号。

strip 的使用方法

strip的使用方法很简单。基本上只需要在想要删除符号的目标文件里指定变量就可以了。以下例子为从可执行文件 test 中删除符号的情况。

```
% strip test
```

按顺序来看看这个例子吧。

```
void test() {}
int main() {
    return 0;
}
```

和上面一样在有test.c时，只要用gcc进行一般的编译，出来的test里面就会包含符号的信息。

```
% gcc -o test test.c
% nm test | grep test
08048334 T test
```

这个时候，test 变为 11,356 字节。

对 test 进行 strip 操作，便可删除符号。

```
% strip test
% nm test
nm: test: 没有符号
```

符号删除后变为 2,796 字节，为何很多明明是空的程序，却含有 8,000 字节之多的符号呢？这是因为test大部分的空间由一个名为/usr/lib/crt*.o的目标文件占据着的。关于不链接这些的方法请参照 **“[Hack #25]不使用 glibc 来编写 Hello World”**。

使用 strip 的诀窍

在 strip 中有各式各样的选项，总的来说最方便的还是 -d 选项。使用 -d 选项后，可以删除不使用的信息（文件名和行号码等），并可以保留函数名等一般的符号。

用gdb进行调试时，只要保留了函数名，即便不知道文件名和行号，也可以进行调试。在想要减小文件的大小、并保留对调试有用的信息时，这个选项非常方便。-R 选项是可删除其他任意信息的选项。在运行了 strip -R .text program后，程序的text 部分（代码部分）会被完全删除，从而导致程序的无法运行。

实际上，对 .o 文件以及 .a 文件使用strip后，就不能进行和其他目标文件的链接操作。这是由于文件对链接器符号有依赖性，所以最好不要从 .o 和 .a 文件中删除符号。

再者，在已上市制品的二进制程序中运用了strip，可以从开发者方面获得包含了调试信息的二进制程序版本。这样的话，便可以在开发环境中调试在用户环境中产生的核心文件。

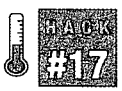
strip 的安装

strip是用BFD库来进行安装的。通过调用BFD的API，来对目标文件进行操作。遇到了Binutils的源代码后，objcopy和代码就会共通。实际上在objcopy 上使用 -strip-* 选项后也能进行与 strip 同样的处理。

总结

在本 Hack 中介绍了 strip 的使用方法。虽然在磁盘容量足够大的 PC 中，可能不会出现想要将可执行文件变小的情况。在容量有限的环境中安装程序时，以及想要通过网络复制并运行程序时，strip 却是一个方便的工具。

—— *Satoru Takabayashi*



用 ar 操作静态链接库

ar命令是用于从静态链接库生成阅览、编辑、展开的命令。在本Hack中介绍了这个ar 命令。

ar 的使用方法

其实 ar 命令的生成与存档并不仅限于静态链接上，与 tar(1)等一样，可以广泛地作为非压缩的存档使用，但是通常用于操作静态链接库。

在建成存档时，只要建成ar rcus libhoge.a foo.o bar.o baz.o就可以了。r 选项在新建时表示插入，在寄存时则表示置换。选项c为libhoge.a不存在时将禁用发出警告信息的命令。u则是遇到时间戳就替换为新东西的命令。s选项可与ranlib(1)进行相同处理并建立索引。

若不建立索引的话，链接的速度就会降低，在不同的环境中可能会产生不同的错误。这种索引可用nm -s浏览。

用ar tv libhoge.a可以浏览存档。t为浏览的命令，v表示浏览方式，利用这些选项可以检查文件大小或更新时刻等信息。

用ar xv libhoge.a展开存档，附加v选项可以边确认边展开文件。

ar不仅可以编成一般的静态链接库，也可用于更改静态链接库的部分信息。在由于不能同时打开所有库代码等原因不能重建静态链接库时，使用这个工具非常方便。

ar虽然还含有存档内部移动、删除等对存档进行操作的命令，但这里所介绍的命令在一般的情况下不会出现什么问题。必要的时候，用man或者help来查询。

总结

在本Hack中，将ar作为操作静态链接库的工具进行了介绍。

—— *Shinichiro Hamaji*

在链接C程序和C++程序时要注意的问题

在本Hack中介绍了从C中调用C++函数，以及反过来从C++中调用C函数的方法。

经常会有想从C++中调用C函数的情况，反过来也会出现要从C中调用C++函数的情况。在本Hack中，介绍了实现这些操作的方法以及要注意的问题。

C/C++和符号名

让我们试着用C编译器和C++编译器来对下面的dbg函数进行编译。

```
//
// dbg.c
```

```
//
#include <stdio.h>
void dbg(const char *s) {
  printf("Log: %s\n", s);
}
```

生成的对象中包含的符号名如下所示：

编译器	符号名
C	dbg
C++	_Z3dbgPKc

用C编译器对函数进行编译时，一般函数名会成为符号名。不同环境中可能会出现符号名为"_dbg"而不是"dbg"，这只是编译环境的不同造成的。另一方面用C++编译器来编译函数时，如同**[Hack #14]**中说的一样，符号中就会包含函数所属的命名空间的信息和函数的参数类型信息。

从C++中调用C函数

用C编译器来编译dbg.c，然后从C++编写的函数中对其进行调用。首先要准备下面的sample.cpp。

```
//
// sample.cpp
//
extern "C" void dbg(const char *s);
int main() {
  dbg("foo");
  return 0;
}
```

将其用如下的方式进行编译、链接之后，就会生成正常的可执行文件。这样就可以安全地从C++的函数中调用C函数。

```
% gcc -Wall -c dbg.c
% g++ -Wall -c sample.cpp
% g++ -o sample dbg.o sample.o
% ./sample
Log: foo
```

sample.cpp的第4行，extern "C"。让我们来看看在有和没有extern "C"时sample.o的内容有何变化。关于nm命令的使用方法请参考“**[Hack #12]** 用nm来检索包含在目标文件中的符号”。

(1) 有extern "C"

```
% nm sample.o
         U __gxx_personality_v0
00000000 T main
         U dbg
```

(2) 没有 extern "C"

```
% nm sample.o
         U __gxx_personality_v0
00000000 T main
         U _Z3dbgPKc
```

在(2)中，虽然sample.o是引用了"_Z3dbgPKc"这个符号，但包含在dbg.o中的符号只有"dbg"，"_Z3dbgPKc"不存在于任何地方。像这样，不带有extern "C"时，便会导致像下面一样链接失败的结果。

```
% g++ -o sample dbg.o sample.o
sample.o(.text+0x25): In function `main':
: undefined reference to `dbg(char const*)'
```

就像这样，从C++中调用C函数时，extern "C"发挥了巨大作用。

注意：使用 extern "C" 时要检查变量的类型是否一致

在此，我们试着将sample.cpp做如下改写：

```
//
// sample.cpp
//
extern "C" void dbg(int i); //函数原型声明错误
int main() {
  dbg(1);
  return 0;
}
```

试着将这个文件进行编译、链接，无论怎样都会链接成功。当然生成的可执行文件不能正常运行。

```
% g++ -Wall -c sample.cpp
% g++ -o sample dbg.o sample.o
% ./sample
Segmentation fault
```

为什么会链接成功呢？这是因为sample.o引用的是不含类型信息的"dbg"。它和包含在dbg.o里的符号名一致。

C++用惯了之后，会很容易认为“链接成功了的话就不会调用错误的函数”。即使是在C++中使用了extern "C"，也不仅是部分相关。C语言的安全

性非常低，为了防止这种现象的产生，要准备好在C++语言中也能使用的头文件。例如：准备好如下 dbg.h 这样的头文件。

```
//
// dbg.h
//
#ifdef __cplusplus
  extern "C" {
#endif
  void dbg(const char *s);
#ifdef __cplusplus
  }
#endif
```

从 C 中调用 C++ 函数

和上次的正好相反，这次是从C中调用C++函数。可以使用求最大公约数的函数 Boost C++ Library（*http://www.boost.org/*）这个著名的库在C++上可以简单地安装，从C中调用其函数需要注意以下两点。

- C++的函数要结合C（extern"C"）进行编译
- 在链接时，要运行 g++ 命令而不是 gcc

这样操作后，C++的安装便如下所示。

```
//
// gcd.h
//
#ifdef __cplusplus
extern "C"
#endif
int gcd(int v1, int v2);
```

```
//
// gcd.cpp
//
#include <boost/math/common_factor.hpp>
#include "gcd.h"

extern "C" {
  int gcd(int v1, int v2) {
    return boost::math::gcd(v1, v2);
  }
}
```

调用C++函数方面，没有太大的变化。

```
//
// sample.c
```

```
//
#include <stdio.h>
#include "gcd.h"

int main() {
  printf("%d 和 %d 的最大公约数为%d  \n", 14, 35, gcd(14, 35));
  return 0;
}
```

那么可以运行编译了。

```
$ g++ -Wall -c gcd.cpp
$ gcc -Wall -c sample.c
$ g++ -o sample sample.o gcd.o
$ ./sample
14 和35 的最大公约数是7
```

安全运行完毕。从C 中读出了 C++ 函数。

注意：不允许 C++ 函数向 C 函数抛出异常

在写从C语言中调用C++函数时，要注意避免C++中的异常向C语言抛出。若是C++中的异常函数抛出到C函数中的话，就不能明确规定C++的规格。若是在GCC中会导致路径的异常终止。

要充分注意不能让 C++ 函数显示的异常抛出，还要充分注意以下两点：

- new运算符有抛出 throw std::bad_alloc异常的可能
- std::vector的构造函数at有抛出std::out_of_range异常的可能

如下所示，如果将所有的 C++ 函数用 try/catch 控制的话就好了。

```
extern "C" {
  int cpp_func() try {
    //处理有可能抛出的异常
    return 0; // 成功
  } catch(...) {
    return -1; // 失败
  }
}
```

注意：处理函数指针的 C 函数

C 函数有可能不能阻止 C++ 异常的通行。可以考虑标准 C 的 qsort 函数使用在以下所示的在 C++ 上的情况。

```
//
// sample2.cpp
```

```
//
#include <cstdlib>
using namespace std;

// qsort 的比较函数
int compar(const void *, const void *) {
  throw -1;
}

int main() {
  int array[] = {3, 2, 1};
  try {
    qsort(array, 3, sizeof(int), compar);
  } catch(...) {
    return 1;
  }
  return 0;
}
```

由qsort函数过渡而来的比较函数compar为了抛出异常，让C++的异常从qsort函数中通行。在使用GCC时，为了防止路径的异常终止，可以像如下所表示的那样，必须在qsort上附上"-fexceptions"进行编译。

```
% gcc -fexceptions qsort.c
```

一般地，对于处理函数指针的C函数而言，是不难附上-fexceptions选项进行编译的。见到glibc的Makefile时，这样的函数（qsort、bsearch之外）就会附上该选项进行编译。

总结

在本Hack中，分别介绍了从C中调用C++函数以及反过来从C++中调用C函数的方法。还介绍了在进行这些操作时必须注意的两点。

- extern "C"的使用方法
- 对C++异常处理的方法

对于后者，GCC的-fexceptions选项虽然是不怎么熟知的选项，但却在编译处理函数指针的C函数时起着重要作用，因此要牢记它。

另外，有关调用函数的高水平Hack，在“**[Hack #69]**用ffcall读出动态决定签名的函数”等中有介绍。

—— *Yusuke Sato*

HACK #19 注意链接时的标识符冲突

在本Hack中，介绍了在动态和静态链接时会发生的标识符冲突的问题以及如何避免冲突的方法。

同名标识符的冲突

在C和C++程序中存在两个以上的同名global符号。

对.o文件进行整理链接的情况

首先，有如下的a.c文件。在a.c中定义了global函数func()。

```
#include <stdio.h>
void func() {
    printf("func() in a.c\n");
}
```

接下来，在b.c中同样定义为func()。虽然和a.c的很像，但用printf表示的信息是不同的。

```
#include <stdio.h>
void func() {
    printf("func() in b.c\n");
}
```

最后，在main.c中调用func()。

```
void func();
int main () {
    func();
    return 0;
}
```

如果分别将这3个文件进行编译并静态链接的话，因为有多个func()，检索器会发现错误。由于出现了错误，就可以调用非预期func()以避免这种情况。

```
% gcc -c a.c
% gcc -c b.c
% gcc -c main.c
% gcc -o main a.o b.o main.o
b.o(.text+0x0): In function `func':
: multiple definition of `func'
a.o(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

在生成由a.o和b.o组成的共享库libfoo.so里，链接器会检查出符号的冲突，从而发生错误。

```
% gcc -fPIC -c a.c
% gcc -fPIC -c b.c
% gcc -shared -o libfoo.so a.o b.o
b.o(.text+0x0): In function `func':
: multiple definition of `func'
a.o(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

生成库并进行链接时的情况

在相同源代码 a.c和b.c中使用ar生成静态链接库并进行链接时不会发生错误。这是因为ar和链接器不同，作为目标文件的存档的ar不检查符号冲突。

```
% gcc -c a.c
% gcc -c b.c
% gcc -c main.c
% ar cr libfoo.a a.o b.o
% gcc main.o libfoo.a
```

运行的话，便可以调用a.c的func()。这是因为在链接时从libfoo.a的两个 func()中，a.c的func()会比b.c的func()更早的被发现。

```
% ./a.out
func() in a.c
```

用ar建成静态链接库llibfoo.a时，可使用b.o里的func()将变量的顺序由a.o b.o变为b.o a.o。而且对a.c和b.c分别建成静态链接库的情况下，进行链接也不会发生错误。此时再向gcc传递时只要将liba.a和libb.a的顺序颠倒，便可以调用b.c的func()。

```
% ar cr liba.a a.o # liba.a生成
% ar cr libb.a b.o # libb.a生成
% gcc main.o liba.a libb.a
# ./a.out
func() in a.c
```

同样的，分别将a.c和b.c生成动态的共享库a.so和b.so并将它们链接时，也不会发生错误。下面所示的例子中，可以从a.so中调用func()。

```
% gcc -fPIC -shared -o a.so a.c
% gcc -fPIC -shared -o b.so b.c
% gcc -fPIC -shared -o main.so main.c
% gcc -o main-shared ./a.so ./b.so ./main.so
```

在链接时如果颠倒了命令行变量a.so和b.so的顺序，便会调用b.so的func()。在GNU C Library的主要开发成员Ulrich Drepper写的《How to Write Shared Libraries》(*http://people.redhat.com/drepper/dsohowto.pdf*)中有如下的描述。

> scope中包含两个以上的同名标识符的定义也没有关系。标识符查找的算法，只需采用最先发现的来定义就好了 —— 这种概念非常有用。使用LD_PRELOAD的功能就是其中的一例。

关于LD_PRELOAD请参考“**[Hack #60]**用LD_PRELOAD更改共享库”。

这样一来无论是静态链接库、共享库、还是其他的库中，即使定义了同名标识符，程序也可以正常的进行链接和运行。在进行这个操作时可能会发生如“莫名其妙地调用非预期的函数”这样的bug，所以必须引起注意。

C++和同名类

标识符相同的问题，在C++中可能会引起调用非预期的构造函数的问题。

假设存在计算消费税的源代码a.{h,cpp}和使用了这个源代码的main.cpp。

a.h

```
class Tax {
public:
    int tax(int price);
    Tax();
private:
    double consumption_tax_;
};
```

a.cpp

```
#include "a.h"
Tax::Tax() : consumption_tax_(1.05) {}

int Tax::tax(int price) {
    return price * consumption_tax_;
}
```

main.cpp

```
#include <iostream>
#include "a.h"
```

```
int main() {
    Tax tax;
    int apple_price = 100;
    std::cout << "apple: " << tax.tax(apple_price) << std::endl;
    return 0;
}
```

将这些源代码进行编译、链接之后运行，会显示出apple: 105这样的信息。

```
% g++ -fPIC -shared -o a.so a.cpp
% g++ -fPIC -shared -o main.so main.cpp
% g++ -o main-shared ./a.so ./main.so
% ./main-shared
apple: 105
```

在这里，存在着与a.{h.cpp}独立编写的b.cpp，假设其被放置了同名的类Tax，但是b.cpp里的Tax类的用途与a.cpp完全不同。

```
class Tax {
public:
    int deduct(int income);
    Tax();
private:
    double deduction_rate_;
};

Tax::Tax() : deduction_rate_(0.1) {}

int Tax::deduct(int income) {
    return int(income * (1.0 - deduction_rate_));
}
```

然后，如果将b.cpp作为b.so来编译，并不小心按b.so、a.so、main.so的顺序链接的话，就会产生麻烦。

```
% g++ -o main-shared2 ./b.so ./a.so ./main.so
% ./main-shared2
apple: 10
```

这次的运行结果为apple:10。因为这是用于计算包含消费税的价格的，原来的90%会被折扣掉。

这是由于在b.so里调用了作为Tax对象的构造函数的Tax::Tax()，在a.so里调用了作为Tax对象的组成函数的Tax::tax()。也就是说，调用了非预期的构造函数成为了bug出现的原因。

由于同名类的冲突而引起的该问题，虽然在小程序中不会发生，但随着程序的扩大会很容易发生。所以有必要使用命名空间等方法来避免冲突。

只有将符号控制在一个文件夹中，无名命名空间才会有效。如果b.cpp的所有都被控制在namespace { ... }，就不会发生上面的错误。

弱符号

在刚开始时，介绍了如何检测出在将.o文件进行链接时的同名符号冲突。但如果弱符号存在的话，就要另当别论了。

让我们来看看下面的程序main.cpp。

```
#include <iostream>
class Foo {
public:
    Foo(int x) : x_(x) {}
    void func() {
        std::cout << x_ << std::endl;
    }
private:
    int x_;
};

int main () {
    Foo foo(256);
    foo.func();
    return 0;
}
```

将这个程序进行编译并运行，则结果显示为256。

```
% g++ -c main.cpp
% g++ -o main main.o
% ./main
256
```

在这里，存在着与main.cpp独立编写的a.cpp，a.cpp里也放置了Foo类。

```
#include <iostream>
class Foo {
public:
    Foo(int x);
private:
    int x_;
};

Foo::Foo(int x) : x_(x * x) {}
```

此时，对a.cpp编译之后产生的a.o如果运气不好链接到main上的话，会出现很大的麻烦。

```
% g++ -c a.cpp
```

```
% g++ -o main main.o a.o
% ./main
65536
```

这次运行的结果为 65536，是 256 的平方而不是 256。这时，无论你怎样改变 main.o、a.o、main.o 的链接顺序，结果也不会有任何改变。

很明显是使用了 a.cpp 里的 Foo::Foo(int) 而不是在 main.cpp 里定义的 Foo::Foo(int)，但为何会产生这样的结果呢？

关于弱符号

原因是由于在 main.o 里包含的 Foo::Foo(int) 是弱符号，而在 a.o 中的 Foo::Foo(int)....则不是弱符号。以下用 nm 输出的数据中用 W 表示的则为弱符号。

```
% nm --demangle main.o | grep Foo::
00000000 W Foo::func()
00000000 W Foo::Foo(int)
% nm --demangle a.o | grep Foo::
00000012 T Foo::Foo(int)
00000000 T Foo::Foo(int)
```

关于弱符号，在由 John R. Levine 著，榊原一矢翻译的《Linkers & Loaders》的第 6 章里有如下介绍。

> 在 ELF 中，除了弱引用，还增加了另外一个弱符号 —— 弱定义。所谓弱定义，即在不存在一般定义的情况下进行大范围符号定义。若一般定义存在，weak 定义则被忽略。

也就是说，因为 a.o 存在一般定义（非弱定义），所以 main.o 的 Foo::Foo(int) 的弱定义被忽略。

不管怎样，要避免这种问题，利用无名命名空间即可。具体来说，就是把 main.cpp 的 class Foo { ... } 的范围用 namespace { ... } 包围就可以了。

弱定义与重复代码的删除

然而，为什么 main.o 的 Foo::Foo(int) 以及 Foo::func() 会变成弱定义呢？这是因为 g++ 把内联函数作为弱定义的形式进行编码。类定义里的函数定义由 C++ 的规范决定，都被看成内联函数。并且，弱定义在 .o 文件里存放在 .gnu.linkonce.t.* 这个节中。

gnu.linkonce.t.*用于链接时删除重复编码。大多数情况下，类定义在.h文件中，.h文件被多个.cpp文件包含。因此，必须把各个.o文件里的弱定义链接成一个。内联函数会变成弱定义也是由于这个原因。

总结

在本Hack中，介绍了关于进行静态链接与动态链接操作时发生的标识符冲突问题及其对策。通常，不注重链接等操作也可以进行编程工作，但若发生上述问题，链接知识的有无将很大影响到修正错误所需的时间。

—— *Satoru Takabayashi*

HACK #20 建立GNU/Linux的共享库，为什么要用PIC编译？

在本Hack中，将研究建立共享库时，为什么要用PIC编译。

通常，建立GNU/Linux的共享库时，把各个.c文件编译成PIC（Position Independent Code，位置无关代码）。但是，实际上不用PIC编译的话也可建立共享库。那么使用PIC还有意义吗？

那让我们来进行实验。

```
void func() {
    printf("");
    printf("");
    printf("");
}
```

用PIC编译必须把-fpic或-fPIC传递给gcc。-fpic可以生成小而高效的代码，但是不同的处理器中-fpic生成的GOT（Global Offset Table，全局偏移表）的大小有限制，另一方面，使用-fPIC的话，任何处理器都可以放心使用。在这里，使用-fPIC。（在x86中-fpic、-fPIC没有区别）。

```
% gcc -o fpic-no-pic.s -S fpic.c
% gcc -fPIC -o fpic-pic.s -S fpic.c
```

阅读上述生成的汇编程序源，则可知道PIC版通过PLT（Procedure Linkage Table）调用printf。

```
% grep printf fpic-no-pic.s
        call printf
```

```
        call printf
        call printf
% grep printf fpic-pic.s
        call printf@PLT
        call printf@PLT
        call printf@PLT
```

下面，建立共享库。

```
% gcc -shared -o fpic-no-pic.so fpic.c
% gcc -shared -fPIC -o fpic-pic.so fpic.c
```

这些共享库的动态节（dynamic section）用readelf阅读的话，非PIC版中有TEXTREL输入方法(需要在text内进行再配置)，并且假设RELCOUNT（再配置的数量）为5则比PIC版多出3个。多出3个是因为printf()的调用要进行3次。

```
% readelf -d fpic-no-pic.so|egrep 'TEXTREL|RELCOUNT'
 0x00000016 (TEXTREL)                    0x0
 0x6ffffffa (RELCOUNT)                   5
% readelf -d fpic-pic.so|egrep 'TEXTREL|RELCOUNT'
 0x6ffffffa (RELCOUNT) 2
```

PIC版的RELCOUNT非0是由于gcc在缺省时使用的是包含在启动文件里的代码。若把-nostartfiles选项传递给gcc则转为0状态。

PIC与非PIC共享库的性能对比

上面的例子述说了非PIC版运行时（动态运行时）需要5个地址的再配置。那么，若在配置的数量大增时会出现什么样的情况呢?

运行下面的shell script，用非PIC版和PIC版建立含有1000万次printf()调用的共享库，并制作链接了这些的运行文件fpic-no-pic和fpic-pic。

```
#! /bin/sh
rm -f *.o *.so
num=1000
for i in `seq $num`; do
  echo "void func$i() {" > fpic$i.c
  ruby -e "10000.times { puts 'printf(\"\");' }" >> fpic$i.c
  echo "}" >> fpic$i.c
  gcc -o fpic-no-pic$i.o -c fpic$i.c
  gcc -fPIC -o fpic-pic$i.o -c fpic$i.c
done
gcc -o fpic-no-pic.so -shared fpic-no-pic*.o
gcc -o fpic-pic.so -shared fpic-pic*.o
echo "int main() { return 0; }" > fpic-main.c
```

```
gcc -o fpic-no-pic fpic-main.c ./fpic-no-pic.so
gcc -o fpic-pic fpic-main.c ./fpic-pic.so
```

生成的运行文件的运行结果如下：非PIC版首次2.15秒、第2次以后大约0.55秒，而PIC版则首次用了0.02秒、第2次以后用了0.00秒。

```
.% repeat 3 time ./fpic-no-pic
2.15s total : 0.29s user 0.48s system 35% cpu
0.56s total : 0.25s user 0.31s system 99% cpu
0.55s total : 0.30s user 0.25s system 99% cpu

% repeat 3 time ./fpic-pic
0.02s total : 0.00s user 0.00s system 0% cpu
0.00s total : 0.00s user 0.01s system 317% cpu
0.00s total : 0.00s user 0.00s system 0% cpu
```

`main()`本身是空的，可以知道非PIC版动态链接时的再配置需要2.15~0.55秒。运行环境为Xeon 2.8 GHz+Debian GNU/Linux sarge+GCC 3.3.5 。

非PIC版的缺点不仅是在运行时的再配置上花时间。为了更新再配置部分的代码，将会发生这样的情况（下载text segment里需要再配置的页→更新→进行copy on write→不能共享其他的路径与text）。也就是将失去如其他的路径共享text（程序的代码）的共享库这一主要优点。

但是，比较非PIC版的`fpic-no-pic.so`与PIC版的`pic.so`的大小，前者是268MB，后者是134MB，差别很明显。用`readelf -S`阅读节头，会有以下的区别。

	.rel.dyn	.text
非PIC	152MB	114MB
PIC	0MB	133MB

非PIC版的代码（.text）比PIC版的小，但再配置所需的信息占很大的空间。

总结

在本Hack中，研究了建立共享库为什么要用PIC编译的原因。虽然也能建立非PIC的共享库，但运行时的再配置不但花费时间，还存在不能和其他路径与代码（`.text`）共享这一大缺点。建立共享库请用PIC编译`c`文件。

—— *Satoru Takabayashi*

用 statifier 对动态链接的可执行文件进行模拟静态链接

使用statifier，可以把动态链接下的可执行文件和共享库制作成一个文件。

所谓statifier，是把动态链接下的可执行文件和共享库制作成一个文件的一种GNU/Linux工具。也可用于把动态链接下的可执行文件复制到别的主机运行等情况。

因为statifier在原稿中没有成为Debian package，在Debian里使用时要从源代码中选择make && make install。

```
% tar zxf statifier-1.6.7.tar.gz
% cd statifier-1.6.7
% make
% sudo make install
```

statifier 的使用方法

使用方法比较简单，只需通过命令选项指定目标文件和新的文件名，然后执行statifier就OK了。例如要把/usr/bin/php共享库整理成一个名为php2的文件，则执行如下：

```
% statifier /usr/bin/php php2
```

如此形成的php2文件，复制到别的主机上执行。试着把/usr/bin/php和php2复制到没有安装PHP的主机上，php2可以顺利得到执行。单单复制/usr/bin/php的文件，会由于库的不足而无法执行。

```
% scp /usr/bin/php php2 foo.example.com:
% slogin foo.example.com

% ./php -v
./php: error while loading shared libraries: libzzip-0.so.12:
cannot open shared
object file: No such file or directory

% ./php2 -v
PHP 4.3.10-16 (cli) (built: Aug 24 2005 20:25:01)
Copyright (c) 1997-2004 The PHP Group
Zend Engine v1.3.0, Copyright (c) 1998-2004 Zend Technologies

% ldd php |grep 'not found' # ..............................
        libzzip-0.so.12 => not found
        libgssapi_krb5.so.2 => not found
```

```
libkrb5.so.3 => not found
libk5crypto.so.3 => not found
```

/lib/libnss*.so 的问题

在glibc系统中，为了解决使用了DNS等域名系统时编译的问题，/lib/libnss*.so是必需的。由于在最初执行域名解析的时候，这些共享库被加载了，依存关系在静态条件下无法得知，所以statifier的生成文件中无法包含lib/libnss*.so。

试着在statifier中执行getent命令，也就是在执行过程中加载了/lib/libnss*。如果是通过在statifier中生成的二进制相异的Linux机器来执行的话，/lib/libnss*的加载就有可能失败，从而导致程序执行的异常终止。

```
% statifier /usr/bin/getent getent2
% LANG=C strace ./getent2 hosts >/dev/null 2>&1 |grep /lib
open("/lib/libnss_files.so.2", O_RDONLY) = 3
open("/lib/libnss_dns.so.2", O_RDONLY) = 4
open("/lib/libresolv.so.2", O_RDONLY) = 4
```

想把这些共享库包含在二进制里的话，则要使用statifier的--set选项。LD_PRELOAD环境变量中，将要含有二进制的共享库在空白段设定就可以了。

```
% statifier --set=LD_PRELOAD="/lib/libnss_files.so.2 /lib/
libnss_dns.so.2 /lib/libnss_dns.so.2" /usr/bin/getent getent3
```

由于在执行时会加载/usr/lib/gconv/*.so，iconv(3)也会发生同样的问题。glibc的gettext是在内部调用iconv的，所以在使用statifier处理已gettext化的程序时需使用--set=LD_PRELOAD把.so文件包括在其中。

statifier 的组成

statifier必须在ELF和glibc的环境中建成。在ELF和glibc环境中，执行已经动态链接的程序时，ld.so(/lib/ld-linux.so.2等)会进行必要的共享库加载。这个操作分为以下两大部分。

1. 对所有的库进行动态链接所必需的再配置等处理，然后加载到内存中。
2. 调用各库内的初始化函数，将控制权转交到原来的程序入口地址上。

statifier是指在2的临界点上取得内存中的映像，是否能生成在单一文件中可执行的二进制文件。在ld.so中，通过_dl_start()函数进行1的处理，通过_dl_start_user()函数进行2的处理。以.ld.so这个操作为前提，statifier在生成二进制文件时进行如下的处理。

- 在_dl_start_user()中设定断点，执行目标程序
- 以/proc/PID/maps为基础了解到达_dl_start_user()的时刻，如何把目标文件映射到内存中
- 把已经映射好的各个域，转储为文件并保存。
- 最后，链接这些文件，使之成为单一的二进制文件。

实际上，诸如有无TLS查阅、寄存器保存等，虽也进行其他的各种操作，但基本的流程如上所述。

要执行生成的二进制文件，则要在进行寄存器的复原和变量的设置等处理之后，才能进入_dl_start_user()开始操作。这一点，可以认为是从2重新开始刚才在2的临界点的映像，这样理解会容易一些。

如前所述，由于statifier是以ELF和glibc的环境为前提的，将来的话，如果glibc的版本升级，ld.so的操作会发生改变，statifier的结构也有可能发生变化。

总结

使用statifier，可以把动态链接的可执行文件和共享库打包成一个文件。在statifier的站点上，介绍了其构造的文件。想了解详细结构的读者，请务必参考一下。

—— *Satoru Takabayashi*

第3章

GNU 编程 Hack

Hack #22～41

GNU/Linux 就如同它的名称所表示的一样，包含了 GNU 工程的许多成果。实际上，GNU/Linux 除了 Linux 内核之外，还有像 C 库、编译器、链接器、调试器等这些重要的部分都使用了 GNU 工程的成果。在本章中，以 GCC 提供的扩展功能为中心，介绍了适用 GNU 的工具链所提供的各种单独功能的方法。熟练掌握 GNU 的工具链可以说是在 GNU/Linux 上进行 Binary Hack 的最重要课题。

HACK #22 GCC 的 GNU 扩展入门

本部分介绍了 GCC 的具有代表性的 3 个 GNU 扩展功能 [built-in 函数（内置函数）、attribut（属性）、label 引用（标签引用）]。

在本 Hack 中，介绍了 GCC 的 GNU 扩展功能中具有代表性的东西。

内置函数（built-in 函数）

GCC 中的标准函数有很多是作为 built-in 函数存在的，根据优化的不同会生成与源程序中不同的代码。在 `printf(3)` 等情况中，有像下面这样的输出字符串的代码。

```
printf("hello, world\n");
```

程序运行时，由于不能解析 `printf(3)` 的格式字符串，将会生成如下所示 `puts(3)` 的调用代码。

```
puts("hello, world");
```

基本上，虽然在同一运行环境中，生成更高效的可执行文件不会出现什么问题，但有必要留心在LD_PRELOAD等环境变量重写后改变运行行为的情况。gcc/builtins.c中的程序将会决定优化时生成什么样的代码。

此外还存在获得运行时环境的built-in函数，以及在编译时提供提示信息的built-in函数。

`void *__builtin_return_address(unsigned int LEVEL)`

函数的返回地址，也就是返回该函数当前的调用源的地址。虽然LEVEL会用某些常量来指定调用源，但通常指定为0（当前函数）。一般在x86的情况下，变为保存在%ebp + 4上的指针值。

`void *__builtin_frame_address(unsigned int LEVEL)`

返回到函数的frame pointer(指定本地变量和寄存器存储区域的指针)。LEVEL虽会用一些常数来指定调用源，但通常指定为0(当前的函数)。一般在x86的情况中变成%ebp。

`int __builtin_types_compatible_p (TYPE1, TYPE2)`

检测TYPE1和TYPE2是否相容，一般用于在宏中根据种类的不同分别调用适当的函数。

`TYPE __builtin_choose_expr(CONST_EXP, EXP1, EXP2)`

这个功能虽然和CONST_EXP ? EXP1 : EXP2差不多，但在编译时，决定用哪一个却有所差异。在编写选择适当函数调用的宏时，多在CONST_EXP中使用__builtin_types_compatible_p。

`int __builtin_constant_p (EXP)`

判断EXP是否为常量。一般用于如下情况，当知道参数为常量时，选择优化函数代码。

`long __builtin_expect(long EXP, long C)`

用于在判断EXP的值大多数情况下会取C的基础上，对分支进行优化。

`void __builtin_prefetch (const void *ADDR, int RW, int LOCALITY)`

用于将ADDR所指向的数据预读取到缓存上。RW为1时，将显示附近存在的写入信息，为0时则显示近处存在的可读信息。LOCALITY可以是0~3这几个值。若使用了0则显示出无用数据，若使用了3则显示出不久要继续使用的数据。

Attribute(__attribute__)

在函数声明时加上 attribute，可对函数进行特殊优化。

以如下的方式声明 attribute。

```
int foo(int n) __attribute__((attribute));

int foo(int n)
{
...
}
```

还可以写成：

```
__attribute__((attribute)) int foo(int n)
{
}
```

对于函数，attribute 包含如下所示的内容：

`constructor`

用于调用 `main` 前和加载共享目标时必须运行的函数。

`destructor`

用于在进行 `exit` 前和直接运行为加载的共享目标时必须运行的函数中。

`cleanup`

指定局部变量从周期中销毁时调用的函数。

`section`

在特定的节上配置代码。

`used`

即使是在未被调用的情况下也一定会生成代码。用于只从汇编程序代码中被调用的情况。

`weak`

生成弱符号的代码。

`alias`

作为其他符号的别名，一般和 `weak` 一同使用。

`visibility`

用于控制宏符号的可视性，有 "default"、"hidden"（共享目标之外

不可见）、"protected"（共享目标内的调用即使是LD_PRELOAD也不能被覆盖）、"internal"（不能进行从共享目标外部将包含有函数指针的调用）等。

stdcall

指定x86的调用规范，堆栈由被调用方负责维护。

cdecl

指定x86的调用规范，堆栈由调用函数负责维护。

fastcall

指定x86的调用规范，用寄存器%ecx、%edx调用最开始的2个变量。

regparm

控制是否将变量传送到几个寄存器上。

vector_size

指定变量的维数大小

dllimport

指定引入动态链接库中的代码。

dllexport

指定将代码导出到动态链接库中。

pure

根据全局变量和参数（argument）决定返回值，可用于无副作用的情况。在某些情况下可以省略对无用函数的调用。

const

仅用变量决定返回值，可在无副作用的情况中使用。

malloc

用于NULL以外的返回值不和其他的指针共享的情况。

noreturn

用于像exit(2)这样的不能返回的函数。

noinline

用于不想将内联（inline）展开时。

always_inline

即使是优化水平很低时也展开内联。

nothrow

用于不抛出异常的情况。

format

显示格式字符串与printf、scanf、strftime、strfmon中的哪个样式相同，并进行格式字符串与可变参数的对应检索。

format_arg

显示哪个变量为格式字符串。

nonnull

显示不为NULL的pointer参数。

unused

即使是不能使用的情况也不发出警告。

deprecated

在使用时出示警告。

warn_unused_result

在未获取返回值时出示警告。

no_instrument_function

即使是在-finstrument-functions时，也不调用profile函数。

关于数据的attribute有以下几种。

aligned

控制变量的区间对齐。

packed

在structure内部根据对齐生成的代码变为最小。

common

将变量放置在公有区域。

nocommon

不将变量放置在公有区域。

shared

在使用DLL的所有路径中用于共享变量。

标签（label）引用

虽然在C中不能像这样使用，但在goto要跳转时为了指定跳转目标必须要使用标签。

```
    goto error;
    ....
error:
    /* 错误处理 */
```

在GCC中，这个标签可用 && 在引用时代入到 void * 型的变量中。

```
    void *label;
    label = &&error;
    ...
    goto *label;
    ....
error:
    ...
```

由于这样可以将标签的地址存入变量中，根据变量的值来改变跳转目标的代码可将对标签的引用作为数组的元素。由于将用&&引用过的标签存入void *型的指针，所以根据减法来得到偏移量。然后也可以写出如下所示的那样的代码。

```
    static int labals[] = { &&laba10 - &&label0, &&label1 - &&label0, ... };
    ....
    goto *(&&laba10 + labels[i]);
    ....
label0:
```

总结

GCC是以标准的C99为基础运行的，并提供多个可轻松记录源代码的扩展，理解了GCC的扩展功能之后，使用这些功能就很方便了。例如：Linux的内核存在使用GCC扩展功能的代码，要想读取这些代码就必须要对GCC的扩展功能有所了解。

—— *Fumitoshi Ukai*

在GCC上使用内联汇编（inline assembler）

使用了GCC的asm命令就可以使用C中的inline assembler

在本Hack中，对在GCC上inline assembler的使用方法进行了说明。

寄存器变量

可以将某变量分配到指定的寄存器中，若进行如下操作，便可用C中的变量stack_pointer、frame_pointer引用%esp、%ebp。

```
register void *stack_pointer asm ("%esp");
register void *frame_pointer asm ("%ebp");
```

内联汇编

在GNU/Linux的<string.h>中定义了像如下所示的这样用内联汇编定义高速的strcpy()等。

```
26 #define __HAVE_ARCH_STRCPY
27 static inline char * strcpy(char * dest,const char * src)
28 {
29 int d0, d1, d2;
30 __asm__ __volatile__(
31         "1:\tlodsb\n\t"
32         "stosb\n\t"
33         "testb %%al,%%al\n\t"
34         "jne 1b"
35         : "=&S" (d0), "=&D" (d1), "=&a" (d2)
36         :"0" (src),"1" (dest) : "memory");
37 return dest;
38 }
```

__asm__命令用[:] 可出现以下的分段形式。

```
__asm__ ("assembler template "
        : 输出operand的设定
        : 输入operand的设定
        : 在assembler的运行中变更了的部分)
```

上面strcpy()的情况，会变成如下所示的这样：

汇编程序模板（assembler template）

```
1: lodsb
   stosb
   testb %al,%al
   jne 1b
```

输出 operand

```
"=&S" (d0), "=&D" (d1), "=&a" (d2)
```

输入 operand

```
"0" (src),"1" (dest)
```

变更部分

```
"memory"
```

关于operand，会用表示制约条件的字符串和与其对应的C语言方式来指定。

首先指定输出operand，在"=&S"(d0)中变量d0有着"=&S"制约。它的意思如下：

- [=].asm完成之后，改变的结果将会被带入到指定的变量（d0）里
- 在运行之前变更[&]asm
- [S]分配到%esi寄存器中

由于这是初始的寄存器制约，所以在汇编程序模板上可引用%0，也可以用于其他operand的"0"制约。

然后在"=&D"(d1)中变量d1有"=&D"，这也和[=&]相同，[D]表示的是分配到%edi寄存器上的意思。由于这是接下来的寄存器指定，所以可引用%1，然后在operand中变为"1"指定。

其次，"=&a"(d)中变量d2带有"=&a"制约。[a]分配到%eax寄存器中。变为在%2上可参照的"2"制约。

然后指定输入operand，在"0"(src)中变量src带有"0"制约。"0"制约，就像已经介绍了的那样，由于被分配到%esi寄存器中，所以将src的值设定到%esi寄存器上。

在"1"(dest)中变量dest带有"1"制约。"1"制约也像上面所说的一样，由于被分配到%edi寄存器上，所以将dest的值设定到%edi寄存器上。

由于将变更后的部分设定为"memory"，所以运行了这个asm命令后GCC将会接到所传达的关于内存的内容有变更的信息，这样最优化的内存上的内容便不会和寄存器里所保持的一样。此外，如果存在有影响的寄存器，还会记录下这些过程。

综上所述，可像下面这样来解释：

```
将src变量的值设定到 %esi寄存器中
    根据输入operand中的"0" (src)和输出operand 的"=&S" (d0)
dest的值设定到%edi寄存器中。
根据在输入operand的"1" (dest)里输出operand的"=&D" (d1)
```

```
下面的汇编程序代码将被隐藏。
  1:  lodsb
      stosb
      testb %al,%al
      jne 1b
%esi寄存器被代入到变量d0里
 根据输出operand的"=&S" (d0),
%edi寄存器将被代入到变量d1里
 根据输出寄存器的"=&D" (d1),
%eax寄存器被代入到变量d2中
 根据输出operand的 "=&a" (d2)
```

__asm__后面的__volatile__是防止编译器的最优化的关键，防止由于最优化而导致代码的消除。

寄存器制约

用这样的GCC的asm命令，可以通过在operand上指定制约条件来指定使用哪个寄存器。

使用"g"等将在适当的寄存器中分配GCC。这种不指定特定寄存器也可以的情况下，GCC会充分考虑在前后的C的代码中寄存器的使用状况，不会分配无用的寄存器。

在x86中运用下面的制约可以进行特定的寄存器分配。

寄存器	制约
eax	a
ebx	b
ecx	c
edx	d
edx:eax	A
edi	D
esi	S
fp	f
st(0)	t
st(1)	u
xmm SSE	x
MMX	y

这些会由于构造的不同而不同。这由GCC的源gcc/config/<architecture>/< architecture >.h.(gcc/config/i386/i386.h等）的宏REG_CLASS_FROM_LETTER(C)来决定。C是制约字符，将选择合适的与字符相对应的寄存器成为返回宏。

详细信息请参照gcc.info的Machine Constraints等。

总结

使用了GCC的asm命令便可以在C中进行inline assembler。对用operand制约字符控制C的变量和汇编程序的寄存器的对应的分配进行了说明。

—— *Fumitoshi Ukai*

活用在GCC的built in函数上的最优化

在本Hack中，对编写使GCC更优化、更有效的代码的方法进行了说明。

GCC上的最优化的例子

将下面的C语言的程序用gcc进行编译，将会生成什么样的代码呢？要怎样来编译呢？特别是函数开头的2个strlen函数的调用？

```
//
// 建成User-Agent头
//
#define USER_AGENT_HDR_NAME "User-Agent: "
char *get_user_agent_hdr(const char *hdr_value) {
  assert(hdr_value != NULL);
  const size_t name_len = strlen(USER_AGENT_HDR_NAME);
  const size_t value_len = strlen(hdr_value);
  char *hdr = malloc(sizeof(char) * (name_len + value_len + 1));
    // 省略对整数溢出的检索
  if (hdr)
    sprintf(hdr, "%s%s", USER_AGENT_HDR_NAME, hdr_value);
  return hdr;
}
```

事实上试着用gcc -fverbose-asm -S将程序变换为汇编语言时，便会得到下面的结果。

```
1: get_user_agent_hdr:
2:        pushl   %ebp    #
```

```
3:      movl    %esp, %ebp      #,
4:      subl    $24, %esp       #,
5:      movl    $12, -4(%ebp)   #, name_len
6:      subl    $12, %esp       #,
7:      pushl   8(%ebp) # hdr_value
8:      call    strlen  #
...
```

请注意第5行，虽然是第1行strlen函数调用，在编译时会变成值（$12）。

Built in 函数

实际上GCC是以strlen为首的，用C语言规格来规定的主要的函数，与libc不同的是保存在内部。这在GCC中称为built in函数。然后在“将文字列literal传送到strlen函数中”等特定的场合，GCC会利用built in 函数将输出代码最优化（高速化）。

成为最优化对象的函数

GCC会将这些成为最优化对象的函数向多方面传送。对于最近的地方，会用多种形式对printf函数施行最优化。除此之外还有以下所示的函数也会成为最优化的对象。

- 与数学相关的函数（abs、acos、asin...）
- 判定字符种类的函数（isalnum、isalpha、iscntrl...）
- 操作字符串的函数（strcat、strchr、strcmp...）

GCC 在线使用手册（on-line manual）的《Other built-in functions provided by GCC》这一章节中有详细地总结，有兴趣的话可以参看一下。

另外，在调试的状态下不想进行这种最优化时，可以像下面这样使用-fno-builtin选项来编译。

```
% gcc -fno-builtin foo.c
```

在源代码中直接使用 built in 函数

不是为了GCC 的最优化而最优化，而是像《Other built-in functions provided by GCC》中介绍的那样在源代码内直接使用函数__builtin_foobar()，并能让GCC输出高速的代码。这里列举出Hacks经常使用的处理办法。

```
int __builtin_clz(unsigned int x);
```

试着“将变量 x 开头的不知是否为"0"的几个 bit 变为返回函数”。clz 是 count-leading-zero 的省略。

```
const unsigned int x = 0x0fffffffU; // 最初的4bit为0，余下的 28bit为1
printf("%d\n", __builtin_clz(x));
```

用 gcc -O2 将这个代码进行编译，便会输出如下的已最优化过的代码。GCC 在编译时，会将 .printf 的第 2 个变量计算为 4。

```
.LC0:
      .string "%d\n"
(略)
      pushl   $4
      pushl   $.LC0
      call    printf
```

关于这个 Hack“**[Hack #22]** GCC 的 GNU 拓展入门”的相关解说也非常有参考价值。

注意指向字符串 literal 的指针

有“尽量不要使用 #define”这样的最新编程手法。据此，试着将最初的程序改写如下。

```
//不正确的例子
static const char *USER_AGENT_HDR_NAME = "User-Agent: ";
  char *get_user_agent_hdr(const char *hdr_value) {
  size_t name_len = strlen(USER_AGENT_HDR_NAME);
...
```

这样操作后会发生意外情况，GCC 没有将第 1 个 strlen 函数读出并变为即值，读出了 libc 的 strlen 函数。

```
...
pushl USER_AGENT_HDR_NAME
call strlen
...
```

实际上，这是在想将这种最优化运用到指向字符串 literal 的指针上的情况时，对变量 const 修饰的另一必要条件。进行如下的操作后，结果会和最开始的程序大致相同。

```
// 正确的例子
static const char* const USER_AGENT_HDR_NAME = "User-Agent: ";
//（另外的）正确的例子
static const char USER_AGENT_HDR_NAME[] = "User-Agent: ";
```

“指针参照的内存内容”并不是一定的，如果不明确指出一定的“指针参照”，GCC就不会将strlen函数的读出安全地变换为即值。

总结

定义没有预定变更的变量时，一定要用const来修饰。特别是在公布指向字符串literal的指针时，要附上2个const。GCC是用built in函数最大限度地将代码高速化。另外，在源代码中公开地使用built in函数，在GCC中还可能输出高效的代码。

—— *Yusuke Sato*

HACK #25 不使用glibc写Hello World

在本Hack中，介绍了只使用系统调用的编程。使用了这个Hack，便可以把小程序当成有效的二进制来输出。

所谓的“Hello World”，若在C语言中用5行左右的程序便可以写出，但是那样写出来的程序用gcc进行编译、链接后，即便是动态链接也能产生5KB左右的二进制。在本Hack中，用简浅易懂的形式介绍了“为何5行的程序会变成5KB？”、“如何输出更小的二进制？”。

5KB的内部解析

来看看像谜一样的5KB的内部解析。在gcc上附上-v选项，进行Hello World程序的编译、链接。

```
% gcc -o hello -v hello.c
...
/usr/libexec/gcc/i386-redhat-linux/3.4.4/collect2 --eh-frame-hdr
  -m elf_i386 -dynamic-linker /lib/ld-linux.so.2
  /usr/lib/gcc/i386-redhat-linux/3.4.4/../../../crt1.o
  /usr/lib/gcc/i386-redhat-linux/3.4.4/../../../crti.o
  /usr/lib/gcc/i386-redhat-linux/3.4.4/crtbegin.o
  -L/usr/lib/gcc/i386-redhat-linux/3.4.4 -L/usr/lib/gcc/i386-redhat-linux/3.4.4
  -L/usr/lib/gcc/i386-redhat-linux/3.4.4/../../.. /tmp/ccOPEg3W.o -lgcc
  --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -
lgcc_s --no-as-needed
  /usr/lib/gcc/i386-redhat-linux/3.4.4/crtend.o
  /usr/lib/gcc/i386-redhat-linux/3.4.4/../../../crtn.o
% wc -c hello
4712 hello
```

这样一来，便可以得到这样的输出。不难看出，gcc 命令不仅将动态链接 libc.so，还将许多 .o 文件链接在 hello 二进制上。

在这些crt*.o文件中，简而言之包含有使glibc初始化的代码。因此，不仅可以不使用包含在glibc(libc.so)中的函数来记录 Hello World程序，而且只要不链接 crt*.o，便可得出非常小的二进制。

编写小型 Hello World

如果是 Hello World 程度的程序，不仅可以使用包含在printf等的 libc 里的函数，还可以直接运用 write 系统调用。只是，像下面这样，直接使用 write 函数和 syscall 函数会行不通。

```
#include <unistd.h>
...
  write(1, "Hello World\n", 12);
```

这是由于这里的write和syscall和“系统调用的本质”不同，它们是包含在 glibc (libc.so) 里的单纯“函数”。由于调用 system call 的方法也就是进入 kernel mode 的方法，即使在相同的 Linux 中也会因构造的不同而不同，glibc则解决了这个差异。虽然很难，但在这里一定要考虑不是依靠 glibc 而是直接运用 system call 的方法。

用内核的源代码掌握调用 system call 的方法

实际上在 x86 的情况，读了“**[Hack #59]**怎样调用 system call”后，便可以明白直接调用 system call 的方法，包括寄存器使用方法。但是，由于利用了x86以外的东西，即使不知道system call的调用方法也不用担心，只要复制并链接Linux内核的源代码中的一部分，便可以简单地调用出system call (不使用glibc)。

参考文件是“linux-2.6.x/include/asm-< architecture >/unistd.h.”。打开这些文件，存在名叫 syscallN的宏 (N=0,1,2...)。将这些宏复制并链接到源代码 (hello.c) 中，暂时保留 _syscall1和_syscall3以及它们所依存的宏比较好。

仅用系统调用来写 Hello World

这样生成的源代码如下所示：

```
#include <asm/unistd.h> // __NR_xxx
static int errno;

//从内核的源代码中，复制并链接下来的部分
#define _syscall1(type,name,type1,arg1) \
   (略)
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
   (略)
// 复制和链接到此

_syscall1(int, exit, int, status);
_syscall3(int, write, int, fd, const void*, buf, unsigned long, count);

void hello() {
  write(1, "Hello World!\n", 13);
  exit(0);
}
```

接下来的2行，像下面介绍的“直接调用系统调用的函数”那样展开。

```
_syscall3(int, write, int, fd, const void*, buf, unsigned long, count);
_syscall1(int, exit, int, status);
```

看看x86_64中的例子，似乎只要使用syscall这个CPU命令便可以调用系统调用。

```
int write(int fd, const void *buf, unsigned long count) {
  long __res;
  __asm__ volatile ("syscall"
        : "=a"(__res)
        : "0"(__NR_write), "D"((long)fd), "S"((long)buf), "d"((long)count)
        : "r11","rcx","memory");
  if ((unsigned long)__res >= (unsigned long)-127) {
    errno = -(__res);
    __res = -1;
  }
  return (int)__res;
}
```

hello函数，只运用上面展开的函数来记录，将这个hello.c用如下方式进行编译。

```
% gcc -Os -fno-builtin -fomit-frame-pointer -fno-ident -c hello.c
```

通过省略frame.pointer进行大小优先的优化（-Os），将生成的目标文件变小。

链接

不是用gcc命令而是用ld命令来执行链接。虽然ld命令在缺省时经常将

_start 函数变为入口，但由于它包含在 _start crt*.o 里，所以现在不存在。因此，指定 hello.c 的 hello 函数为入口。

```
% ld --entry=hello -o hello hello.o
```

进行静态链接便出现了942字节的二进制，试用objdump将它（hello）进行反汇编（-d），则会变成很简单的构造。

努力将其精简

首先将链接产生的二进制进行 strip -s，然后用 readelf 命令输出节的所有内容，将大小为0的节或判断为无用的节用 strip -R 删除。

```
% strip -s hello
% readelf -S hello
Section Headers:
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 0    0  0   0  0
[ 1] .text             PROGBITS        08048094 000094 00005b 00  AX  0   0  4
[ 2] .rodata           PROGBITS        080480ef 0000ef 00000e 01 AMS  0   0  1
[ 3] .data             PROGBITS        08049100 000100 000000 00  WA  0   0  4
（略）
% strip -R .data hello
```

最后用动态链接将其精简为488字节的二进制。

```
% wc -c hello
488 hello
% file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked,
stripped
% ldd hello
not a dynamic executable
```

虽然这些二进制很小，但也可以安全地运行。

```
% ./hello
Hello World!
```

获得命令行变量

虽然在先前 Hello World 程序中，入口函数的变量是作为 void，但像一般的 main 函数那样，会出现想要知道命令行变量和环境变量列表的情况。在这种情况下，要参考 glibc 的源代码 glibc-2.4/sysdeps/<true>/elf/start.S，用汇编程序来记录像将 hello 函数作为 argc、将 argv 作为变量读出那样的成为了入口的函数就 OK 了。

start.S的_start函数，会将包含着argc或argv的一些变量作为参数，从而读出C函数__libc_start_main，所以将其简略化成为“仅将argc 和argv作为参数，读出hello”那样就好了。虽然掌握若干的汇编程序语言是一件难事，但这个操作起来并不难，关于这个操作，请参考“**[Hack #58]**程序在到达main()之前”。

总结

在Hack中，介绍了只用系统调用来编程的方法。使用了这种Hack，可以将极小的二进制的程序当作极小的二进制来输出，在汇编过程中，这个方法也有实际的用处（*http://www.selinux.gr.jp/LIDSJP/document/general/web_lids_busybox/main.html*）。

由于介绍的是以Linux和glibc的源代码为参考的方法，所以无论是在怎样的构造中，只要是参考了本Hack均可输出极小的二进制。甚至不理解该构造的汇编语言也没有关系。

—— *Yusuke Sato*

使用TLS（Thread-Local Storage）

在gcc中，可通过运行__thread关键字来使用TLS

虽然每个线程都可以使用同名变量，但实际上TLS（Thread-Local Storage）指的是所存放的值能独立保存的每个线程区域。

在gcc中，可通过使用__thread关键字来使用TLS。例如：将线程分成3部分、每个线程都分成3部分、在每个在线程中参照了全局变量和TLS变量的代码表示如下：

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

#define THREADS 3

__thread int tls;
int global;

void *func(void *arg)
{
  int num = (int) arg;
  tls = num;
```

```
  global = num;
  sleep(1);
  printf("Thread=%d tls=%d global=%d\n", num, tls, global);
}

int main()
{
  int ret;
  pthread_t thread[THREADS];
  int num;

  for (num = 0; num < THREADS; num++) {
    ret = pthread_create(&thread[num], NULL, &func, (void *) num);
    if (ret) {
      printf("error pthread_create\n");
      exit(1);
    }
  }
  for (num = 0; num < THREADS; num++) {
    ret = pthread_join(thread[num], NULL);
    if (ret) {
      printf("error pthread_join\n");
      exit(1);
    }
  }
  exit(0);
}
```

运行这个程序会得到如下结果：

```
% ./threadlocal
Thread=0 tls=0 global=2
Thread=2 tls=2 global=2
Thread=1 tls=1 global=2
```

在上述程序中，由于全局声明变量在线程间可共享，所以将保留最后的赋值结果（在这里为2）。与此相对，TLS变量tls虽可以使用与全局变量相同的变量名，但实际上可以保存每个线程的不同值。

使用 TLS 的情况

__thread关键字，例如在glibc中，可针对错误值errno来使用。这是为了保持每个线程的不同错误值。

__thread关键字自身是gcc所固有的。作为portable使用的方法，存在着pthread的_thread固有数据（TSD: Thread-Specific Data），其中的例子有pthread_key_t类型和pthread_key_create函数等。

TLS 的安装

TLS 在 Linux 的情况中，可在 Linux 内核 2.6 +gcc 3.3 +glibc 2.3+NPTL 之后活用。这样的话，为了实现 TLS，有必要将内核 glibc、gcc 等各个部分协调起来。

另外，由于体系结构的不同，安装方法也有很大的不同。例如：TLS 在 x86 中可以通过 %gs 实现线程局部存储，也存在着像 mips 那样由于没有保留多余的线程寄存器，所以不能实现 TLS 的构造。关于安装的详细资料，可参照以下网站："ELF Handling For Thread-Local Storage"（*http://people.redhat.com/drepper/tls.pdf*）、"The Native POSIX Thread Library for Linux"（*http://people.redhat.com/drepper/nptl-design.pdf*）。

总结

本 Hack 中，介绍了在 GNU/Linux 的环境中 TLS（Thread-Local Storage）的使用方法。

—— *Masanori Goto*

HACK #27 根据系统不同用 glibc 来更换加载库

Linux 中运行的 glibc 的动态加载中，每一平台和 HWCAP 都能更新库。

运行于 Linux 中的 glibc 的动态加载，在 x86 和 sparc 等构造中每个细小的平台（i386、i486、i586、i686）和 HWCAP（HardWare CAPabilities x86 的 MMX 和 SSE 等）都能更换（更新）库。

例如：在**[Hack #62]**用 dlopen 运行时的动态链接介绍的关于 dlopen(3) 的程序 dlsay 在以不存在的库为对象运行时，加上栈跟踪时如下所示：

```
% strace -f ./dlsay non-existed symbol
...
open("/lib/tls/i686/sse2/cmov/non-existed", O_RDONLY) = -1 ENOENT
(No such file or directory)
stat64("/lib/tls/i686/sse2/cmov", 0xbfe29570) = -1 ENOENT (No
such file or directory)
open("/lib/tls/i686/sse2/non-existed", O_RDONLY) = -1 ENOENT (No
such file or directory)
stat64("/lib/tls/i686/sse2", 0xbfe29570) = -1 ENOENT (No such file
or directory)
```

```
open("/lib/tls/i686/cmov/non-existed", O_RDONLY) = -1 ENOENT
(No such file or directory)
...
open("/lib/sse2/non-existed", O_RDONLY) = -1 ENOENT (No such
file or directory)
stat64("/lib/sse2", 0xbfe29570) = -1 ENOENT (No such file or directory)
open("/lib/cmov/non-existed", O_RDONLY) = -1 ENOENT (No such file
or directory)
stat64("/lib/cmov", 0xbfe29570) = -1 ENOENT (No such file or directory)
open("/lib/non-existed", O_RDONLY) = -1 ENOENT (No such file or
directory)
```

很显然在中间加上了很多 look up，仔细观察，就能发现 "tls"、 "i686"、"sse2"、"cmov" 这些关键字。这些是在检索普通的库目录之前检索的特别目标，每个都有意思上的区别。

tls

在与内核 2.4 和 2.6 两者相应时，要在与 thread-local-storage 相对应的情况中检查分配。例如在使用了内核 2.6+TLS 并安装了 LinuxThreads 的系统中，通过在 LD_ASSUME_KERNEL 这一变量上设定值与否来控制是否使用了 TLS。另外，在最近的 glibc 中，LinuxThreads 的支持可能变得不能运行（随使用中处理器类型名而变化名称）。

i686（根据使用中的 processorclass(platform) 变化的名字）

由于在 i686 处理器中运行，所以要进行检索，在老 i586 机器中，要检索 i586 的而不是 i686 的目录。

sse2、cmov

sse2 和 cmov（Conditional MOVe），用于在构造中既存在支持的处理器与不支持的处理器的情况，这便称为 HWCAP。

平台和 HWCAP 信息传送结构

上述的平台名和 HWCAP，实际上是在以内核向程序中传送控制权时，作为 ELF 信息的一部分和 AUXV（AUXiliary Vector）数据一起被传送的，这种表示信息的方法由程序启动时 LD_SHOW_AUXV 环境变量来决定。这样，glibc 的动态加载可输出各种信息。

```
% LD_SHOW_AUXV=1 /bin/echo
AT_SYSINFO:      0xffffe400
AT_SYSINFO_EHDR: 0xffffe000
AT_HWCAP: 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
AT_PAGESZ:        4096
AT_CLKTCK:        100
AT_PHDR:          0x8048034
AT_PHENT:         32
AT_PHNUM:         7
AT_BASE:          0xb7fd0000
AT_FLAGS:         0x0
AT_ENTRY:         0x8048a90
AT_UID:           4107
AT_EUID:          4107
AT_GID:           400
AT_EGID:          400
AT_SECURE:        0
AT_PLATFORM:      i686
```

在这里，AT_PLATFORM表示平台，AT_HWCAP表示HWCAP，使这些信息在内核启动时从CPU中读出并储存，关于HWCAP，在x86系的处理器中，通过读出CPU标识来实现。

根据平台与HWCAP来区分使用

分别作成i386和i686+SSE2用的2级库。然后，用带有SSE2的机器和不带它的机器更换用dlopen()加载的库。

通常推荐的方法，是用库中的当前代码来检查是否支持SSE和i686。但是，在分别组成这些代码的时候比较麻烦(例如在编译水平的最优化时配置分解等)。前面提到的平台HECAP信息就很有用。

使用方法比较简单，通常在放置了dlopen和库的目录上（/lib、/usr/lib、LD_LIBRARY_PAHT等）放置普通的通用库。例如在这里则放上i386用的库。然后，在这个目录之下建成i686/sse2目录，并放置用i686+SSE2优化的库。

这样一来，glibc可以很简单地优先载入到i686+SSE2最优库中。

设定LD_HWCAP_MASK变量

上述的HWCAP，由于实际上将所有的flag标志进行汇总检索后会发生组合，然而，在加载路径名中限定编入的名称要素，并在内部将其作为检索对象。特别是在最新的glibc 2.3.9x系列中，只有sse2成为检索对象。

有时候会想检查i386、i686+MMX和i686+SSE2这3个种类，在将这些的

标签作为对象时，可由 LD_HWCAP_MASK 环境变量来控制。下面举个例子来证明将 MMX 和 SSE2 作为检查对象的情况。

```
LD_HWCAP_MASK=0x04800000
```

这个标记位置，是依赖于处理器种类的值。在 Pentium 4 中，bit23 相当于 MMX、bit26 相当于 SSE。

自身进行 dlopen() 的程序是比较好，但在由 ld.so 加载动态库时，glibc 在未发现 LD_HWCAP_MASK 的设定时会加载被 */etc/ld.so.cache* 隐藏的路径名的库，此时，要注意不会读出当前优化过的库。

总结

在本 Hack 中介绍了利用平台和 HWCAP 来更换库的方式，在这里介绍了两种方法，是为系统准备的绝招，在使用时要注意它们间的互换性。

Debian GNU/ Linux sarge 是活用了平台和 HWCAP 进行发布的一种。一般的 glibc 中不能被检索，在缺省 cmov 的 HWCAPmask 中进行追加时，不带有像 VIA C3 那样的 cmov 命令也没有关系，平台中加入了这样的组成，他们可以在 i686 那样的不寻常的 CPU 中安全地运行二进制。

—— *Masanori Goto*

由链接后的库来变换程序的运行

本部分介绍了用 GNU 扩展利用弱符号的方法。

运用了弱符号后，便可以通过链接库变换程序的运行。在本 Hack 中，介绍了运用了 GNU 扩展利用弱符号的方法。

看看样例代码，在这个程序中，包含在 libm.so 里有 sqrt() 这个函数时会利用这个方法，没有时会表示它的大致信息。

weak.c

```
#include <stdio.h>
extern double sqrt(double x) __attribute__ ((weak));

void func() {
    if (sqrt) {
        printf("%f\n", sqrt(10.0));
    } else {
```

```
        printf("sqrt isn't available\n");
    }
}

int main() {
    func();
    return 0;
}
```

运行结果如下所示：

```
% gcc -fPIC -shared -o weak.so weak.c # weak.so
% gcc weak-main.c ./weak.so -lm; ./a.out
3.162278
% gcc weak-main.c ./weak.so; ./a.out
sqrt isn't available
```

这个程序的重点是weak.c的第2行的那一部分使用了__attribute__((weak))GNU扩展，声明了弱符号。

用nm来读包含在weak.so里的符号的话，sqrt会变成弱符号。

```
% nm a.out |grep sqrt
         w sqrt
```

弱符号在链接符号定义时却不能找到的情况下，会被初始化为0，上面的程序中，sqrt函数在链接时不被发现则变为sqrt = 0，执行if (sqrt)的else的块。

弱符号的优点

和上例中相同的是，在像被运行了autoconf的#ifdef HAVE_SQRT那样的(macro)宏中也可以办到。

```
#ifdef HAVE_SQRT
        printf("%f\n", sqrt(10.0));
#elif
        printf("sqrt isn't available\n");
#endif
```

但是，在这种情况中，weak.so的运行在编译时是被固定的，这一点是不同的。因此，在用HAVE_SQRT = 1编译weak.so时，为了让利用了weak.so的程序（客户端）读出sqrt()，一定要链接libm.so。

像libm.so这样的库没有什么问题，但在特殊的库的情况下（非一般的库或libpthread等对程序的性能产生影响的库等），客户端有时会选择是否链接这个库，而弱符号在这些情况中很有用。

总结

在本 Hack 中，介绍了通过弱符号用链接后的库来变换程序运行的方法，由于利用了GNU扩展而使可移植性下降是个难点但在提高库的依存关系的弹性上非常有用。

—— *Satoru Takabayashi*

控制对外公开库的符号

使用了GNU 链接器的版本控制脚本以及GCC扩展的话，则可控制对外公开库的符号。

在C语言中，只能从文件内（编译单位）访问的静态函数也有能从其他文件中访问的非静态函数。但是有的时候，在生成库方面，只有这两个 scope 是不够的。

在本 HACK 中介绍了通过使用 GNU 链接器的版本控制脚本以及 GCC 扩展来控制向库外公开符号的办法。

版本控制脚本的情况

使用了GNU链接器的版本控制脚本就可以控制向外公开的符号。版本控制脚本正如其名，也用于在符号上附上版本信息。关于这点将参照“**[Hack #30]**，在向外公开库的符号上附上版本控制该运行”。

看下面的例子：

```
% cat a.c
// 由于foo()是libfoo的主要函数，所以想公开
void foo() {
  bar();
}

% cat b.c
// 由于bar()只能在库中使用的，函数本身不想公开
// 但由于被包含在别的文件中的foo()使用
// 不得不成为非static
void bar() {
}
```

将这些代码 a.c 和 b.c 分别进行编译，链接从而生成 libfoo.so，通常，foo()和bar()这两方面函数符号会向库外公开，但是本来 bar()是不想向外公开的函数。

```
% gcc -fPIC -c a.c; gcc -fPIC -c b.c; gcc -shared -o libfoo.so a.o b.o
% nm -D libfoo.so |grep -v '_'
000005e4 T bar
000005d4 T foo
```

只要使用了GNU链接器的版本控制脚本，便可控制向外公开函数。下面的例子中将foo变为全局的（将库向外公开），前面的例子中定义为局部的（封锁在库中）

```
% cat libfoo.map
{
  global: foo;
  local: *;
};
```

将这样的版本控制脚本用libfoo.map传送到gcc并用-Wl链接，便会隐藏bar以及向外公开foo。-Wl是用来隔开两参数反传送到链接器上的选项。

```
% gcc -fPIC -c a.c; gcc -fPIC -c b.c; gcc -shared -o libfoo.so a.o
b.o \ -Wl,--version-script,libfoo.map
% nm -D libfoo.so |grep -v '_'
000004d8 T foo
```

这时，由于符号bar会被链接器隐藏，所以从foo()到bar()的调用会依据PIC代码的规范，经由PLT。也就是说，即使是符号隐藏起来，函数的读出值也不会发生变化。

优点

控制公开的符号有以下两个优点：

- 库的利用者不能看到非公开API
- 共享库的符号表格变小，降低动态链接的开销

虽然在小软件中大致可以忽略动态链接的成本，但在Firefox和OpenOffice.org这些巨大的软件中却是个很大的问题，关于动态链接开销的问题，请参照“**[Hack #85]**用prelink将程序高速化启动”

C++的情况

C++的情况和C的大致相同，但版本控制脚本的书写方式稍微有些变化，请看下面的例子：

```
{
  global:
    extern "C++" {
```

```
        some_class::some_func*
      };
    local: *;
}
```

所谓的不同点，即是指将符号名放在 extern "C++" {}中，以及在函数名后面附上*。加上extern "C++"后便可用demangle型的C++来匹配C++的符号，因为demangle的C++的符号中包含着变量的类型信息，所以若是函数名后没有附上*则不会匹配。关于C++符号的demangle请参照“**[Hack #14]**，用c++filt将C++符号demangle”。

GCC扩张的情况

也有使用GCC扩张控制公开符号的方法。从最优化的观点来看，比起在链接时控制，在编译的时候控制更有效。具体而言，通过使用GCC的__attribute__((visibility("hidden")))以及__attribute__((visibility ("default")))属性可以进行对公开符号的控制，这些扩张在GCC 4.0之后就可以利用（虽然从GCC 3.x始便开始支持，但适用于C++类的还是可以从4.0开始）。

来看看例子，在下面的程序中，对func1和类Foo来说，明确表示有公开符号的属性，但func2和Bar上什么也没有附上。

```
#define EXPORT __attribute__((visibility("default")))

EXPORT void func1() {}
void func2() {}

struct EXPORT Foo {
   void func();
};
void Foo::func() {}

struct Bar {
   void func();
};
void Bar::func() {}
```

来看看将这些程序进行一般的组建从而得出的共享库，很明显所有的符号都被公开了。

```
% g++ -o test.so -shared test.cc
% nm --demangle -D test.so |grep func
000006ec T func1()
000006f2 T func2()
000006fe T Bar::func()
```

```
000006f8 T Foo::func()
```

将-fvisibility=hidden选项传送到g++上，把缺省的visibility变为hidden的话，没有明显地公开在__attribute__((visibility("default")))上的符号便都不能显示出来，这次的例子中隐藏了func2和Bar的主要函数。

```
% g++ -o test.so -fvisibility=hidden -shared visibility.cc
% nm --demangle -D test.so |grep func
000006ac T func1()
000006b8 T Foo::func()
```

与上面例子相反使用__attribute__((visibility("hidden")))来明显地隐藏符号的方法也有，但是用缺省来隐藏想要公开的东西的方法，更能明确地生成代码，这是由于visibility可以不经过PLT函数直接读出hidden的函数。

总结

与使用版本控制脚本的方法相比，使用了Visibility属性的方法，能生成高效代码。因为Visibility属性为hidden的函数可以不经过PLT直接调用。

介绍了在GNV的开发环境中，控制向库外公开符号的方法。这是在大规模工程中利用共享库时特别有用的技巧。

—— *Satoru Takabayashi*

在对外公开库的符号上利用版本来控制动作

对通过使用版本控制脚本便可利用的版本符号进行了介绍。

像在“[Hack #29]控制向外公开库的符号”中介绍的那样，使用版本控制脚本可控制是否将符号公开。在这里，对通过使用版本控制脚本便可利用的版本符号进行了介绍。

版本的导入

首先，作为版本生成含有在两个变量中返回最大值的函数max与含有这个函数的库Libmax。

libmax1.c

```
#include <stdio.h>
int max(int a, int b)
{
  printf("max__1\n");
  return (a > b ? a : b);
}
```

libmax1.h

```
extern int max(int a, int b);
```

vertest1.c

```
#include <stdio.h>
#include "libmax1.h"
int main(void)
{
  printf("max(1, 2)=%d\n", max(1, 2));
  return 0;
}
```

运行例

```
% gcc -fPIC -c -o libmax1.o libmax1.c
% gcc -shared -Wl,-soname,libmax.so.1 -o libmax.so.1.0 libmax1.o
% ln -s libmax.so.1.0 libmax.so.1
% gcc -L. -lmax -o vertest1 vertest1.c
% LD_LIBRARY_PATH=. ./vertest1
max__1
max(1, 2)=2
```

上述程序中，生成了带有libmax.so.1共享库文件名的库libmax.so.1.0，还生成并运行将其读出的程序vertest1。libmax.so.1在运行时会被动态加载链接，这样vertest1则会返回函数max的结果。

版本符号

至此，将上述例子中举出的取得了两个变量的max变为取得三个变量。在这里，通过使用版本符号即不改变符号名max，也可保持进行各种运作的复数版本。另外，还可控制分别在旧程序中读出双变量max、在新程序中读出了三次变量max。看看下面的例子。

libmax2.c

```
#include <stdio.h>
int max__1(int a, int b)
{
  printf("max__1\n");
  return (a > b ? a : b);
}
```

```
int max__2(int a, int b, int c)
{
  int d = a > b ? a : b;
  printf("max__2\n");
  return (d > c ? d : c);
}
__asm__ (".symver max__1,max@LIBMAX_1.0");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

libmax2.h

```
extern int max(int a, int b, int c);
```

vertest2:

```
#include <stdio.h>
#include "libmax2.h"
int main(void)
{
  printf("max(1, 2, 3)=%d\n", max(1, 2, 3));
  return 0;
}
```

在新程序vertest2中使用了3变量版max 。因此，libmax2.h的max的变量也声明为3个。但如果这样将libmax.so.1变更为3变量版后，原来生成的vertest1也会用3变量版运行。在新生成的共享库libmax2.c中，将2变量版和3变量版附上max之外的函数名来编写。最后，用symver汇编程序在针对于同符号名max的@后面附上版本名，从max开始进行alias。在这里，把旧的2变量版当作LIBMAX_1.0，新的3变量版作为LIBMAX_2.0。

在编译时，将下面的版本控制脚本传送到链接器上，将版本设定在符号上。在这个文件中，通过定义LIBMAX_1.0和LIBMAX_2.0来表示各版本分别带有不同的max符号名。另外，还表示LIBMAX_2.0是LIBMAX_1.0的下一个版本（也可以说依存）。

libmax2.def

```
LIBMAX_1.0 {
  global: max;
  local: *;
};
LIBMAX_2.0 {
  global: max;
} LIBMAX_1.0;
```

进行编译并运行

```
% gcc -fPIC -c -o libmax2.o libmax2.c
% gcc -shared -Wl,-soname,libmax.so.1 -Wl,--version-script,libmax2.def -o
  libmax.so.1.0 libmax2.o
```

```
% gcc -L. -lmax -o vertest2 vertest2.c
% LD_LIBRARY_PATH=. ./vertest2
max__2
max(1, 2, 3)=3
% LD_LIBRARY_PATH=. ./vertest1
max__1
max(1, 2)=2
```

很有趣的是，旧二进制 vertest1 读出了旧 2 变量版 max。分别将这些二进制加上 readelf，可以看出会发生什么。

```
% readelf -a vertest1 | grep max
0x00000001 (NEEDED) Shared library: [libmax.so.1]
08049700 00000307 R_386_JUMP_SLOT 00000000     max
     3: 00000000    37 FUNC    GLOBAL DEFAULT  UND max
    64: 00000000    37 FUNC    GLOBAL DEFAULT  UND max
% readelf -a vertest2 | grep max
0x00000001 (NEEDED) Shared library: [libmax.so.1]
0804968c 00000307 R_386_JUMP_SLOT   00000000  max
     3: 00000000    69 FUNC    GLOBAL DEFAULT  UND max@LIBMAX_2.0
(3)
    76: 00000000    69 FUNC    GLOBAL DEFAULT  UND max@@LIBMAX_2.0
  000000: Version: 1 File: libmax.so.1 Cnt: 1
% readelf -a libmax.so.1
...
Symbol table '.symtab' contains 67 entries:
   Num:    Value  Size  Type    Bind   Vis      Ndx  Name
...
    50: 000004f1    69  FUNC    LOCAL  DEFAULT   11  max__2
    52: 000004cc    37  FUNC    LOCAL  DEFAULT   11  max__1
    63: 000004f1    69  FUNC    GLOBAL DEFAULT   11  max@@LIBMAX_2.0
    65: 000004cc    37  FUNC    GLOBAL DEFAULT   11  max@LIBMAX_1.0
...
Version definition section '.gnu.version_d' contains 3 entries:
  Addr: 0x00000000000002e8 Offset: 0x0002e8 Link: 3 (.dynstr)
  000000: Rev: 1  Flags: BASE  Index: 1  Cnt: 1  Name: libmax.so.1
  0x001c: Rev: 1  Flags: none  Index: 2  Cnt: 1  Name: LIBMAX_1.0
  0x0038: Rev: 1  Flags: none  Index: 3  Cnt: 2  Name: LIBMAX_2.0
  0x0054: Parent 1: LIBMAX_1.0
```

如上，在旧二进制 vertest1 中，想要读出不带有版本的函数 max，却读出了 max__1(max@LIBMAX_1.0)。在新二进制 vertest2 中，max 符号带上了版本，就会被指定读出函数 LIBMAX_2.0.max。

这样一来，便可以运用同名符号将在旧二进制中运用旧版本与在新二进制中运用新版本区分开来。

版本的3种表现方法

在这里，通过在max上附带着四种不同的版本名为例，来展示怎样合理利用上述的版本。

libmax3.def

```
LIBMAX_1.0 {
  global: max;
  local: *;
};
LIBMAX_1.5 {
  global: max;
} LIBMAX_1.0;
LIBMAX_2.0 {
  global: max;
} LIBMAX_1.5;
```

libmax3.c

```
#include <stdio.h>
int max__0(int a, int b)
{
  printf("max__0\n");
  return (a > b ? a : b);
}
int max__1(int a, int b)
{
  printf("max__1\n");
  return (a > b ? a : b);
}
int max__1_5(int a, int b)
{
  printf("max__1_5\n");
  return (a > b ? a : b);
}
int max__2(int a, int b, int c)
{
int d = a > b ? a : b;
  printf("max__2\n");
  return (d > c ? d : c);
}
__asm__ (".symver max__0,max@");
__asm__ (".symver max__1,max@LIBMAX_1.0");
__asm__ (".symver max__1_5,max@LIBMAX_1.5");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

将这个文件进行和上面同样的编译，针对libmax.so.1运行readelf 将会得到以下结果。

```
% readelf -a libmax.so.1
Symbol table '.symtab' contains 74 entries:
```

```
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    51: 0000065b    85 FUNC    LOCAL  DEFAULT   11 max__2
    53: 000005f1    53 FUNC    LOCAL  DEFAULT   11 max__1
    54: 00000626    53 FUNC    LOCAL  DEFAULT   11 max__1_5
    57: 000005bc    53 FUNC    LOCAL  DEFAULT   11 max__0
    63: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LIBMAX_1.0
    64: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LIBMAX_1.5
    65: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LIBMAX_2.0
    66: 00000626    53 FUNC    GLOBAL DEFAULT   11 max@LIBMAX_1.5
    68: 000005bc    53 FUNC    GLOBAL DEFAULT   11 max@
    70: 0000065b    85 FUNC    GLOBAL DEFAULT   11 max@@LIBMAX_2.0
    72: 000005f1    53 FUNC    GLOBAL DEFAULT   11 max@LIBMAX_1.0
...
Version definition section '.gnu.version_d' contains 4 entries:
  Addr: 0x0000000000000036c  Offset: 0x00036c Link: 3 (.dynstr)
  000000: Rev: 1  Flags: BASE  Index: 1  Cnt: 1  Name: libmax.so
  0x001c: Rev: 1  Flags: none  Index: 2  Cnt: 1  Name: LIBMAX_1.0
  0x0038: Rev: 1  Flags: none  Index: 3  Cnt: 2  Name: LIBMAX_1.5
  0x0054: Parent 1: LIBMAX_1.0
  0x005c: Rev: 1  Flags: none Index: 4  Cnt: 2  Name: LIBMAX_2.0
  0x0078: Parent 1: LIBMAX_1.5
```

要注意，作为版本名只存在着3种不同的标记方法，它们是：@后面没有版本、@后面有版本和@@后面有版本。下面分别对其进行介绍。

@ 后面没有版本（max@）

将这个时候的符号称为未指定版本的基本符号。上述gnu.version_d entry中成为BASE的Index值为1。之所以要使用未指定版本的基本符号，是因为在运行vertest1，并且在libmax.so.1中只存在max__0 和max__2的库时，使用旧max__0。但是，若存在@后带有版本的entry 则不能使用这个基本符号。

libmax3.c 只包含以下的 .symver 时

```
__asm__ (".symver max__0,max@");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

使用基本符号

```
% LD_LIBRARY_PATH=. ./vertest1
max__0
max(1, 2)=2
```

@ 后有版本（max@LIBMAX_1.0..max@LIBMAX_1.5）

对于符号名max，附在@后面的版本名可设定不同的复数符号（这便是GNU扩张）。gnu.version_d entry中Index的值变为2以下。在libmax3.def

的例子中，由于LIBMAX_1.5依赖着LIBMAX_1.0所以Index的值要变为比LIBMAX_1.0更大的值。

未指定版本的基本符号是否被定义不重要，在带有版本符号的复数存在时，旧二进制ｖｅｒｔｅｓｔ１则会变为由选择函数ｍａｘ最初设定的版本max@LIBMAX_1.0。

libmax3.c 包含以下的 symver 时

```
__asm__ (".symver max__0,max@");
__asm__ (".symver max__1,max@LIBMAX_1.0");
__asm__ (".symver max__1_5,max@LIBMAX_1.5");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

使用带版本符号的最初版本

```
% LD_LIBRARY_PATH=. ./vertest1
max__1
max(1, 2)=2
```

@@ 后面有版本（max@@LIBMAX_2.0）

将@@后面有版本的称为缺省版本。缺省版本是在定义版本符号时必不可少的东西。在一个符号中存在非缺省版本但缺省版不存在，或存在复数个缺省版本，在链接时就会出错。

缺省版本通常表示最新的符号版本。只要不指定版本并编译，从二进制中被链接的版本不论何时都能被选择为缺省版本max@@LIBMAX_2.0。在上面的例子中，将 /vertest2 编译便能自动使用max__2便是这个原因。

用带版本的库编译过的二进制，将在符号上附上缺省版本予以记录。因此，如果库在升级时重新定义LIBMAX_3.0，并且使之依赖LIBMAX_2.0的情况，vertest2也可在这里读出max@LIBMAX_2.0。

版本符号的优点

使用版本符号最大的好处就是它将库的扩张简单化了。想要变更上述的max这样的函数名时，若是应用程序内部的函数，将在2参数版max的其他函数上追加3参数版的max_arg3，从而变更max自身的样子。但是，若是库中公开的函数，变更库的主要版本或将新式样的函数变更为别的名称（例如new_max.等），实施这些互换性对策就十分重要。

但是，在libmax被广泛应用的情况中，由于只变更了函数max 的式样使得主要版本的变更以及函数名的改变变得困难。不妨考虑一下libc和libpng等。若将主要版本改变了话，一直参照libmax.so.1这个名称的二进制便会由于不能进行任何动态链接而无法运行，而且在libc中公开的函数名从某种程度上讲已被决定。版本符号能很好地解决这些问题。

总结

在这里，结合例子介绍了版本符号。对库进行维护的话，会出现无论如何都想要变换式样的情况。利用版本符号，则使在保持二进制互换性的同时变更库内部式样成为可能。

—— *Masanori Goto*

HACK #31 在 main()的前面调用函数

分别介绍了通过使用GCC的扩张和C++的constructor这两种手法在 main()的前面调用函数的方法。

在C/C++的程序中，有时候会想要在main()的前面调用函数。在这里分别介绍了如何运用 GCC 的扩张和 C++ 的 constructor。

使用方法

可用__attribute__((constructor))扩张功能在GCC中定义在main()前被呼叫的函数。例如：在下面的程序中，将在main()的前面读出foo()。

```
#include <stdio.h>
__attribute__((constructor))
void foo() {
    printf("hello, before main\n");
}
int main () {
    printf("hello, world\n");
    return 0;
}
```

运行结果如下所示：

```
% ./a.out
hello, before main
hello, world
```

__attribute__((constructor))是GCC的扩张功能所以没有移植性。关于GCC其他函数的属性请参照GCC手册和“[Hack #22]GCC的GNU扩张入门”。

然而，在C++中通过使用class的constructor却能用可移植方法进行同样的操作。在下面的程序中，通过foo_caller这个class的constructor在无名的名称空间中读出foo()。若能建成foo_caller类的对象caller，则foo_caller的constructor能被读出，随之foo()也就会被读出。

```
#include <stdio.h>

void foo() {
    printf("hello, before main\n");
}
namespace { struct foo_caller { foo_caller() { foo(); } } caller; }

int main () {
    printf("hello, world\n");
    return 0;
}
```

库的情况

使用了__attribute__((constructor))的方法也好，使用了C++的constructor的方法也好，无论哪种情况

```
__attribute__((constructor))
void foo() {
    printf("hello, before main\n");
}
```

以及

```
void foo() {
    printf("hello, before main\n");
}
namespace { struct foo_caller { foo_caller() { foo(); } } caller; }
```

将这两个部分作为其他文件（foo.cpp等）用ar来建成静态链接库时，则不能在main的前面读出foo()。

```
% g++ -c foo.cpp
% g++ -c main.cpp
% ar r libfoo.a foo.o
% g++ main.o libfoo.a
% ./a.out
hello, world
```

不建成libfoo.a便直接链接foo.o和生成libfoo.so并链接的情况是一样的。

```
# 链接foo.o
% g++ main.o foo.o
% ./a.out
hello, before main
hello, world

#建成 libfoo.so并链接
% g++ -fPIC -shared -o libfoo.so foo.o
% g++ main.o ./libfoo.so
% ./a.out
hello, before main
hello, world
```

即便是在libfoo.a中，如果是通过-Wl,--whole-archive libfoo.a -Wl,--no-whole-archive这些选项强行链接包含在libfoo.a里的所有目标文件的话，便会读出foo()。

```
% g++ main.o -Wl,--whole-archive libfoo.a -Wl,--no-whole-archive
% ./a.out
hello, before main
hello, world
```

关于库中静态对象的操作，会因OS的种类和GCC、glibc、Binutils的版本的不同而变化，因而要重视其可移植性。

在main()后读出函数

应用了静态对象destructor便可在main()运行完成后读出任意函数，也可用与__attribute__((constructor))相对的__attribute__((destructor))来定义。而且，程序终止时的运行函数可用ANSI C的atexit()函数来设置。

```
#include <stdio.h>

void foo() {
    printf("hello, after main\n");
}
namespace { struct foo_caller { ~foo_caller() { foo(); } }
caller; }

int main () {
    printf("hello, world\n");
    return 0;
}

% ./a.out
```

```
hello, world
hello, after main
```

总结

虽然一般没有必要在main()之前读出函数，但这在运用了LD_PRELOAD来进行HACK的情况中可能会起作用。在“[Hack #63]开始用C表示backtrace”介绍了的libSegFault.so里用__attribute__((constructor))来设置signal handler。

—— *Satoru Takabayashi*

GCC根据生成的代码来生成运行时的代码

解释了GCC通过生成的代码来生成运行中代码的情况以及堆栈溢出这样的安全防范的相关问题。

首先，介绍了GCC通过生成的代码来生成运行时的代码的情况。

trampoline

在GCC中生成的代码可自动在运行中生成代码。但是这种情况只可能在C语言的函数中定义其他函数，然后将该内部函数的地址向其他函数传送，经过被传送的地址读出内部函数，然后访问用外部函数定义的本地函数——这种极其稀有的情况下才能发生。

来看看具体的代码。

```
void other(void (*funcp)()) {
  funcp();
}

void outer(void) {
  int a = 10;

  void inner(void) {
    printf("outer's a is %d\n", a);
  }

  other(inner);
}
```

这里考虑读出了outer()的情况。outer()将函数inner()的地址作为变量读出了other()，读出的other()经过被传送的地址读出了inner()。inner()取得了外部的outer()函数的本地变量a的值。

读出 inner()时 stack 的情况如图 3-1 所示。inner()为了得到变量 a 的值，穿越了数个函数帧才能访问 outer()的帧内部。因此 inner()必须要把握好变量 a 的位置。若从 inner()运行中的堆栈指针和基指针到变量 a的距离是一定的话，那只需GCC在生成的代码中简单的填补这个距离就可以了。但遗憾的是不能判断这个距离是否能运行，进而言之，inner()的读出也可能会发生改变。

那么，outer()在 stack 上生成代码。outer()生成的代码和 inner()一同便可以进行从 inner()到 outer()的本地变量的访问。

outer()在 stack 上生成下面的代码。

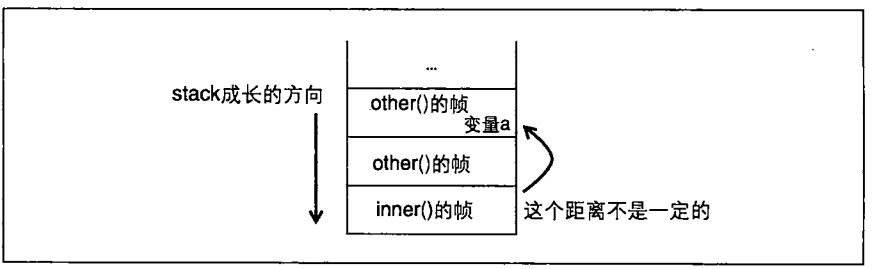


图 3-1：stack 的状态

将 outer()的本地变量的地址包含在某个寄存器中
跳转到 inner()的首地址上

作为 inner()的地址，将代替真正的地址，将生成的代码的地址传送到 other()上。other()虽然想通过函数 pointer funcp 读出 inner()，但实际上将运行这个代码。这样一来，由于 outer()的本地变量的地址进入了某个寄存器将读出 inner()，inner()便可以以此为线索来访问 outer()的本地变量。

这里在 stack 上生成的代码称为“trampoline”。一旦被调用的话，便立即将控制权转移至 inner()。

GCC 的扩展

在 C 语言的 ISO 标准即在 JIS 标准中，不允许在函数内部定义其他函数，这是 GCC 特有的功能。

和安全防范的关系

stack上的代码的生成，要以安全防范的问题来做保证。这是因为stack上常常会由于运行不正当的命令来引起缓冲区外溢这样的攻击。因此，为了安全起见，stack上的命令不能超越无法运行的设置。所以，设置在stack和heap上的命令的运行经常会被OS禁止。若想强制运行，则处理器会发生意外。在这个例子中，UNIX的OS中发生了signal。

但是，即使如此也不能运行stack上的trampoline。对于想要允许运行设置在stack上的代码的对象，GCC会附上表示该意愿的节头。关于这一点，在"**[Hack #33]**许可/禁止设置在stack上的代码的运行"中有相关介绍。

总结

在本Hack中，介绍了GCC通过已生成的代码来生成运行中代码的情况以及堆栈溢出这样的安全防范相关问题。

——*Kazuyuki Shudo*

HACK #33 允许/禁止运行放置在stack里的代码

通过设置决定可否运行ELF形式的对象中stack上代码的flag，可允许运行放置在stack里的代码。

目前，由于安全防范上的原因，一般由OS来禁止放置在stack或heap上代码的运行。但由于也有想要允许其运行的时候，GCC可以设置决定ELF形式的对象中stack上代码运行可否的flag。

运行stack的flag的变更

下面的代码中，有必要运行置放在stack里的代码（trampoline）。

```
void other(void (*funcp)()) {
  funcp();
}

void outer(void) {
  int a = 10;

  void inner(void) {
    printf("outer's a is %d\n", a);
  }
```

```
  other(inner);
}
```

将其作为trampoline.c，并进行编译。只要书写合适的main()函数并读出outer()，表示为outer's...便可以运行。

接下来，让我们来看看得到的对象trampoline.o的节头。

```
% gcc -c trampoline.c
% readelf -S trampoline.o
...
Section Headers:
  [Nr] Name              Type      Addr     Off    Size   ES Flg Lk Inf Al
...
  [ 7] .note.GNU-stack   PROGBITS 00000000 000105 000000 00   X  0   0  1
```

要注意节头note.GNU-stack的flag（Flg）变成了X。在这里X表示execute，即运行可否。

接下来，试着将这个flag复位。在这里要使用GNU binutils的objcopy命令。也可以使用prelink命令。但这样一来便不能运行stack上的代码，上述程序也会异常终止。

```
% objcopy --set-section-flags .note.GNU-stack= trampoline.o
trampoline-modified.o
(链接 )
(运行)
 Segmentation fault
```

若此时不想异常终止，则你所有的处理器和OS则没有禁止运行放置在stack里的代码的功能。

x86 处理器和 Linux 的 exec-shield 补丁

SPARC 等处理器和以前的相比拥有了禁止特定内存区域运行的功能，然而当初的x86处理器，用同一个flag表示可否读出和可否运行，所以可以读出和自动运行。因此，在2004年年初发布的迎新版 Pentium 4.Prescott 中，引入了独立表示可否运行的 flag。

但是，Prescott以前的x86处理器并不是不能禁止运行stack和heap上的代码。将面向Linux内核的补丁exec-shield公开，在其所用的内核中，若是处理器不带有该功能，在内存区域也可读出。但可能被设定为不可运行。

在 Linux distribution 中，也有内核上适用 exec-shield 补丁的情况。（例

如Fedora Core）在这种情况中，若处理器不具备该功能，会导致变更了运行可否flag的上述代码异常终止。

总结

介绍了操作ELF对象中stack运行flag的方法，放置在stack上的代码通常设定为不可运行，在linker-loader与节头note.GNU-stack相吻合时，使用该操作，便可控制stack上的代码运行可否。

—— *Kazuyuki Shudo*

运行放置在heap上的代码

在本HACK中，介绍了使用mprotect(2)来运行放置在heap上的代码的方法。

mprotect系统调用

在内存保护完备的处理器和OS中，用malloc(3)等获取的内存区域，不能运行放置在heap上的代码。

来看看下面的程序，用malloc(3)复制确定的heap上的函数func()，然后读出heap上的该复制的程序，将其运行，便会表示出Segmentation Fault并导致异常终止。若没有异常终止，那么在这个环境中便不能执行该种内存保护。

```
double func(void) {
  return 3.14;
}

int main(int argc, char **argv) {
  void *p = malloc(1000);
  memcpy(p, func, 1000);
  printf("PI equals to %g\n", ((double (*)(void))p)());
}
```

那么便轮到了mprotect(2)运用这个系统调用，可以变更内存区域所允许的访问方法，在上述程序中追加以下函数：

```
void allow_execution(const void *addr) {
  long pagesize = (int)sysconf(_SC_PAGESIZE);
  char *p = (char *)((long)addr & ~(pagesize - 1L));
  mprotect(p, pagesize * 10L, PROT_READ|PROT_WRITE|PROT_EXEC);
}
```

读出main()函数的memcpy()后立即追加上以下的部分，读出这个allow_execution()。

```
allow_execution(p);
```

这样一来，不会异常终止并可以表示PI equals。

mprotect(2)的参数

在allow_execution()中，使用了sysconf(3)便可获取页面（OS管理内存的单位）的大小。这是因为将mprotect(2)的第一参数作为了页面大小的倍数。至少在Linux 2.4..2.6中有必要将这个参数变为页面大小的倍数，也有为了从某个值的倍数中获得起始内存区域而使用posix_memalign(3)的方法。

在这里显然用malloc(3)确定的heap中读出mprotect(2)，但要注意的是，由于POSIX的规定，mprotect(2)只能在用mmap(2)取得的内存区域中使用，而且，在Mac OS X中还有必要用宏PAGE_SIZE来取得页面的大小，而不是用sysconf(3)。

总结

在本Hack中，介绍了用mprotect(2)来允许运行放置在heap上的代码的方法。

—— *Kazuyuki Shudo*

建成PIE（位置独立运行形式）

本部分介绍了PIE（位置独立运行形式）的建成方法及其特性。

PIC(位置独立代码)一般用于共享库中，但在Linux上若使用了最近的GCC、glibc以及Binutils后，可执行文件也可位置独立。在本Hack中介绍了PIE（位置独立运行形式）的建成方法及其特性。

PIE的基本情况

来看看这个例子，假设有下面这样的文件foo.c：

```
#include <stdio.h>
void foo() {
```

```
    printf("hello\n");
}

int main() {
    foo();
    return 0;
}
```

若将这个文件附上-fPIE选项进行编译，附上-pie选项进行链接的话便可建成PIE，生成的文件可以运行。

```
% gcc -c -fPIE foo.c
% gcc -o foo -pie foo.o
% ./foo
hello
```

试着用objdump -d来反汇编foo，会发现地址会变成非常小的数字。这是因为，无论在地址空间上的哪个位置进行map都可运行，和共享库一样在PIE内全部都是相对地址

```
% objdump -d foo |grep main -A3
00000862 <main>:
 862:   55                      push   %ebp
 863:   89 e5                   mov    %esp,%ebp
 865:   83 ec 08                sub    $0x8,%esp
```

在不向gcc传送任何东西而生成的可执行文件中，要像以下这样来分配绝对地址。

```
% gcc foo.c
% objdump -d a.out |grep main -A3
08048378 <main>:
 8048378:       55                      push   %ebp
 8048379:       89 e5                   mov    %esp,%ebp
 804837b:       83 ec 08                sub    $0x8,%esp
```

在PIE的运行中，和在共享库中进行的处理一样，动态链接ld.so会将相对地址在地址空间上进行map处理。此时Linux distribution中为了强化安全防范会将map后的地址随机化，这样有防止侵入特定地址等攻击的效果。

既能运行也能进行动态链接的二进制

从这之前的介绍中可以看出，PIE具有和共享库非常相似的性质。实际上，若在链接时在.gcc上加上-rdynamic选项的话，便可生成既能运行也能进行动态链接的二进制，-rdynamic选项在运行文件中也能留下动态链接用的符号。若向gcc上传送-rdynamic，则会向链接器ld上传送--export-dynamic选项。

在下面的例子中，生成将文件 foo 进行动态链接从而读出函数 foo() 的程序 call-foo.c，并运行。

```
% cat foo.c
#include <stdio.h>
void foo() {
    printf("hello\n");
}

% cat call-foo.c
void foo();
int main() {
    foo();
    return 0;
}

% gcc -c -fPIE foo.c; gcc -rdynamic -o foo -pie foo.o
% gcc -o call-foo call-foo.c ./foo
% ./call-foo
hello
```

既能运行又能进行动态链接的二进制虽然不是常用，但在非常特殊的情况下却能发挥特殊作用。下面的例子是在自动动态链接的同时计算阶乘的程序：

```
% cat factorial.c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <assert.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        void *handle = dlopen ("./factorial", RTLD_LAZY);
        assert(handle != NULL);
        int (*factorial)(int) = dlsym(handle, "factorial");
        n = n *factorial(n - 1);
        dlclose(handle);
        return n;
   }
}

int main() {
    const int n = 5;
    printf("factorial(%d) = %d\n", n, factorial(n));
    return 0;
}

% ./factorial
factorial(5) = 120
```

和共享库的区别

共享库和能动态链接的PIC虽然相似，但在读出非 static 函数这一点却有所不同，前者要经过PLT来读出共享库内的非static函数，后者则直接读出PIE内的非 static 函数而不经过 PLT。

因此，把PIE当作动态链接来运用的话，在LD_PRELOAD上符号的更新则变为共享库时的样子。因为LD_PRELOAD没有不经过PLT而直接读出函数的能力。

总结

在本 Hack 中，介绍了 PIE（位置独立运行形式）的建成及其特性。虽然现在PIE 还不是一种使用十分广泛的技术，但在强化安全防范的方面也可以使用，所以，今后的利用会扩大。

—— *Satoru Takabayashi*

用 C++ 书写同步方法（synchronized method）

介绍了2个用C++书写同步方法（synchronized method）的方法。

在用C++书写复数thread程序时，要怎样来记录“同时只想让一个thread运行的函数”呢？在本 Hack 中，介绍了安装这种函数的奇妙方法。

C 的情况

在C语言中安装这种函数时，会变成如下的样子，没有什么特别的问题，会正常运行。

```
void need_to_sync() {
  static pthread_mutex_t m = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
  pthread_mutex_lock(&m);
    // 某些处理
  pthread_mutex_unlock(&m);
}
```

C++ 的情况

那么，在 C++ 中这种函数会怎样呢？一般在 C++ 中，不会直接去使用

pthread_mutex_t型，而是使用wrap之后的Mutex类，不是使用pthread_mutex_lock函数，而是使用Lock类。

Lock类在constructor上连接到Mutex则立即将其锁住，在destructor上自动解锁，lock类使用这样的方法来安装实例。运用了被称为RAII（Resource Acquisition Is Initialization）的方法的话，即便是Mutex在锁住中抛出异常的情况，也可以将Mutex解锁。下面，来看看在C++中的3种安装实例。

不理想的方法（1）

马上想到这和C语言中的例子很相近，安装过程如下所示。

```
#include <boost/thread/recursive_mutex.hpp>
void need_to_sync() {
  static boost::recursive_mutex m;
  boost::recursive_mutex::scoped_lock lk(m);
  // ...某些处理 ...
}
```

但是，这个却不能很好的运行，由于C++的规定是在首次读出need_to_sync函数时读出对象m的constructor，而这个constructor的读出却不能保证thread安全进行。

试着将上述代码附在g++ -S上，这样一来便可以得到下面的目录。加上注释，以便于理解。

```
        # 若flag不是0则跳跃到L11
        cmpb    $0, guard variable for need_to_sync()::m
        jne     .L11
        #读出m的constructor
        pushl   need_to_sync()::m
        call    boost::recursive_mutex::recursive_mutex()
        # 将flag变为1
        movb    $1, guard variable for need_to_sync()::m
.L11:
```

"guard variable for need_to_sync()::m"是用来记载能否读出m的constructor的flag。从上面可以看出，need_to_sync函数的初次读出若被复数个线程（thread）同时执行，将会发生多次读出m的constructor这样的错误，这个安装不能正常运作。

不理想的方法（2）

那么，将m从局部的静态变数变为非局部的静态变数（global变数或类变数）又会如何呢？在GCC中，由于会经过entrypoint的_start函数，在读出

main函数之前读出m的constructor（**[Hack #31]**），所以一旦控制到达main函数，need_to_sync函数便会进行正常的排他控制。

```
#include <boost/thread/recursive_mutex.hpp>
namespace /* anonymous */ { boost::recursive_mutex m; }
void need_to_sync() {
  boost::recursive_mutex::scoped_lock lk(m);
  // ... 某些处理 ...
}
```

这个方法虽然可以正常地运行，但有一个难懂之处。

在其他的.cpp文件上使用非局部的静态对象，其constructor直接或间接地读出need_to_sync函数时，need_to_sync函数将会锁住在constructor中未完成初始化的对象m。

像m这样的变量是复数的情况下，很难明确控制其初始化的顺序。这个问题称为“static initialization order fiasco”。当然不仅是在Mutex中，在下面这样的代码中也同样出现了这种问题。

```
const std::string FOO_BAR = "foobar"; // 全局变量
```

详细信息请参照文献[1]。

巧妙的方法：将C互换构造体的Mutex静态初始化后再使用

避免“static initialization order fiasco”的方法有很多种，在这里介绍一个最黑客化的手法。“static initialization order fiasco”发生的原因，主要是对global的对象（constructor等）进行了动态初始化。因此，停止这种动态初始化，将运行后的二进制（进行mmap后）在短时期内完成初始化的话就可以了。

仔细参考C++的标准，“C互换构造体”是“集成体”的一种，可以进行动态的初始化，我们便可以利用它。C互换构造体，虽然不含有constructor、destructor、基础class、假想函数、protected member、private member等部分这个局限性，但是具有非假想函数是没有问题的。这是很重要的。Mutex/ Lock的安装如下所示。因为static_mutex类不具备基础class和假想函数，所以scoped_lock类要用模板来调用static_mutex类的lock/unlock函数。

```
// 用模板安装RAII的lock class
template<typename T>
```

```
class scoped_lock_ : private boost::noncopyable {
  T& m_;
public:
  explicit scoped_lock_(T& m) : m_(m) { m_.lock(); }
  ~scoped_lock_() throw() { m_.unlock(); }
};

// 可静态初始化的Mutex类（省略错误处理）
#define STATIC_MUTEX_INIT {PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP}
struct static_mutex {
  pthread_mutex_t m_;
  void lock() { pthread_mutex_lock(&m_); }
  void unlock() { pthread_mutex_unlock(&m_); }
  typedef scoped_lock_<static_mutex> scoped_lock;
};
```

以如下方式将其运行：

```
namespace { static_mutex m = STATIC_MUTEX_INIT; }
void need_to_sync() {
  static_mutex::scoped_lock lk(m);
  // ... 某些处理 ...
}
```

将need_to_sync函数附加到g++ -S上，来看看会输出什么样的代码。

```
(anonymous namespace)::m:
.long 0
.long 0
.long 0
.long 1
.long 0
.long 0
```

像这样直接将m的初始化内容（.long的6个数据）直接输入到对象中。当然不能全部进行动态初始化。这样一来，便可以在C++上安全地实现.synchronized method。

窍门：-fthreadsafe-statics

在最新的使用GCC的情况中，实际上实行“不理想的方法（1）”这样的编码也能安全运行。在g++上附上-fthreadsafe-statics选项，并试着将“不理想的方法（1）”中的代码进行编译、反汇编。

```
% g++ -D_REENTRANT -fthreadsafe-statics -c bad1.cpp
% objdump -Cdr bad1.o | grep -B1 __cxa_guard_acquire
118:    e8 fc ff ff ff    call 119 <need_to_sync()+0x1d>
119: R_386_PC32 __cxa_guard_acquire
```

若在分解后的目录中，像上述这样读出__cxa_guard_acquire函数的话，对象m的constructor将在GCC中用thread safe的方法来调用。只要使用了这个GCC，便可安全地用“不理想的方法(1)”那样的代码来实现同步方法。

再者，支持-fthreadsafe-statics的GCC的大部分，在缺省时这个选项则有效。反之，在安装组成的用途方面若static变量的thread safe的初始化产生的时间和空间的成本成为问题的话，则明确地选择g++-fno threadsafe-statics比较好。

总结

用C++编写同步方法非常重要。总结一下，有如下两个方法：

- 自动生成C互换构造体型的Mutex类，并进行静态初始化，然后用class模板来上锁。
- 直接使用既存的Mutex类。但是由于要让g++附上-fthreadsafe-static选项来使用，所以避免了各种问题。

前者比较复杂但需要类平台，后者虽然依赖特定版本的GCC，但很方便。

虽然C++语言和C语言相比水平明显增高，但在用C++进行这样的编码时，binutils却是相当有用的。笔者在严格书写对于C++的源代码能否被编译这个问题时，分别用了objdump命令和nm命令进行了逐个确认。

参考文献

1. 《C++ FAQ Lite [10.12]～[10.16] What's the "static initialization order fiasco"?》(*http: //www.parashift.com/c++-faq-lite/ctors.html*)
2. JIS X 3014:2003《C++程序语言》(*http://www.webstore.jsa.or.jp/*)

C++语言标准（ISO/IEC 14882:2003）的日语译文。

—— *Yusuke Sato*

用C++生成singleton

试着用四种方法来安装与复数个线程相符的singleton。

在本Hack中，介绍了用C++这种高级语言的编码来活用本书中所介绍的各

种 hack 的方法。具体而言，就是试着用四种方法来安装与复数线程相符的 singleton。

关于 singleton

singleton是一种“保证针对一个类只存在一个实例，并且提供可进行访问的 global 方法”的一种设计模型。用 C++ 来实现 singleton 时，代码如下。

```
class Singleton : private boost::noncopyable {
public:
  static Singleton *instance(); // 获得唯一的实例
private:
  Singleton(); // 禁止生成从 instance 函数以外的对象
  static Singleton *instance_; // 面向唯一实例的 point（最初为 NULL）
};
```

singleton虽然是简单易懂的模型，但可能发生复数个线程同时调用instance函数的情况，这时要详细考虑其安装过程。这是因为用 instance 函数在 thread safe 中生成 singleton 特别难。

4 种安装方法

试着用下面的方法来安装 singleton。这要使用到“**[Hack #36]**用 C++ 书写 synchronized method”中介绍了的 Hack、“静态初始化的 Mutex”，所以要对相关知识有所了解。

(1) 最为保守的 singleton

首先，准备以下这样的 class 变量 m_，并将其静态初始化。

```
private:
  static static_mutex m_;
```

运用 Mutex，以如下方式安装 instance 函数。

```
Singleton *Singleton::instance() {
  static_mutex::scoped_lock lk(m_);
  if(instance_ == NULL) instance_ = new Singleton;
  return instance_;
}
```

虽非常易懂，但在一次生成对象后，每读出 instance 函数都会将 Mutex 锁住，所以不能算是高效的方法。

(2) 运用了内存障碍和双重检查锁的 singleton

为了改良 (1) 中的缺陷，常使用的方法被称为“双重检查锁”。可用一定的条件来省略排他控制。

```
// 不理想的例子
Singleton *Singleton::instance() {
  if(instance_ == NULL) {
    static_mutex::scoped_lock lk(m_);
    if(instance_ == NULL) instance_ = new Singleton;
  }
  return instance_;
}
```

然而，在最开始的部分（最初的 if 部分），由于没有进行排他控制便在复数个线程上操作了共享变量（instance_），所以要注意到“**[Hack #94]**注意处理器的内存秩序”中介绍的问题。为了使模型在依靠 CPU 或编译器的情况下安全运行，有必要插入下面这两个内存障碍。详细信息请参照《Double-Checked Locking, Threads, Compiler Optimizations, and More》[2]。

```
// 正确的例子（资源来自于参考文献2，做了部分修改）
Singleton *Singleton::instance() {
  Singleton *tmp = instance_;
  RMB(); //内存障碍
  if(tmp == NULL) {
    static_mutex::scoped_lock lk(m_);
    if(instance_ == NULL) {
      tmp = new Singleton;
      WMB(); //内存障碍
      instance_ = tmp;
    }
  }
  return instance_;
}
```

在安装RMB()和WMB()时，例如：可以参考用 Linux 内核的linux-2.6.x/include/asm-<architecture>/system.h来定义的 smp_rmb()和 smp_wmb()。

```
//在 PowerPC上的例子
#define RMB() __asm__ __volatile__ ("lwsync" : : : "memory")
#define WMB() __asm__ __volatile__ ("eieio" : : : "memory")
```

这里所说的代替内存障碍来使用volatile修饰的方法，一般而言不是好主意。

(3) 运用了 TLS 的 Singleton

利用在“**[Hack #26]**使用 TLS (thread local storage)”中介绍了的__thread 关键字，可安装高效的 Singleton。

```
Singleton *Singleton::instance() {
   static __thread Singleton *tls_instance = 0;
   if (!tls_instance) tls_instance = do_instance();
   return tls_instance;
}

// private:
Singleton *Singleton::do_instance() { /*和(1)相同 */ }
```

在每个线程中，虽只需一次用(1)的方式（完全同步化）就可取得实例，但由于是将其保存在TLS中，所以第二次及其以后的锁定则无用。若使用的环境中可使用TLS，则是个不错的方法。

(4) 运用了GCC的-fthreadsafe-statics的Singleton

利用最新的GCC的功能，可以非常简单地生成安全线程的Singleton。

```
Singleton *Singleton::instance() {
   static Singleton instance_;
   return &instance_;
}
```

将类变量instance_移动到instance函数内部，之后便和“**[Hack #36]**用C++来书写同步方法”中介绍的Hack相似了。虽然使用最新GCC的情况有所限定，但能成功地将instance_高速地在安全线程上初始化。

bench mark

在提供参考之前，我们试着测定了四种Singleton的速度。下面的图表中，是在笔者的环境（Linux/x86_64）中读出10亿次instance函数时花的时间（图3-2）。(2)～(4)中使用的方法哪一个更快会因环境的不同而不同。

更进一步的Hack

运用了本书的Hack后，除了这里列举的(2)～(4)之外，还可以安装其他类型的Singleton。例如：只要使用了“**[Hack #80]**用自动更新的方式来更换程序的运行”中的Hack，便能实现像“若一次生成实例，会将instance函数自动更新到下面这样没有条件分歧的简单程序中”这样强烈的Singleton。

```
// 自动更新后
Singleton *Singleton::instance() {
  return instance_;
}
```

请多做尝试。

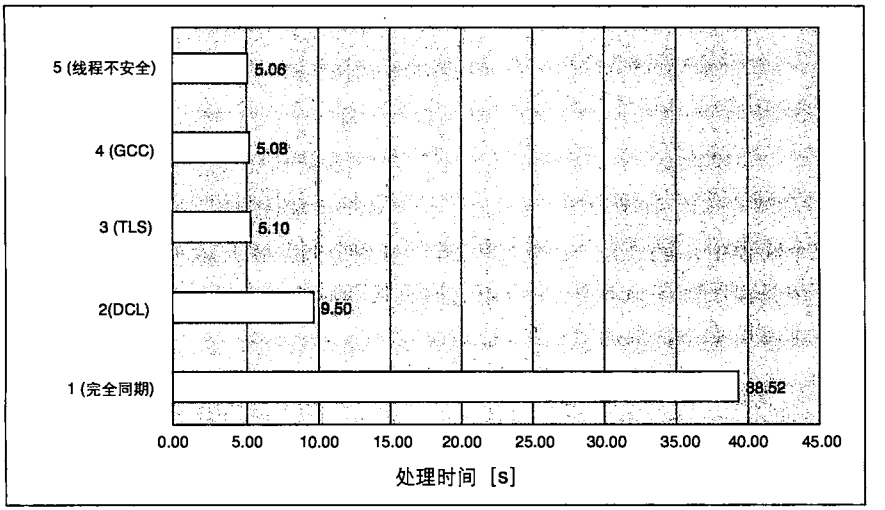


图3-2：四种 Singleton 的 bench mark text

总结

在本 Hack 中，用 4 种方法在 thread safe 安装了作为 C++ 的固定编号 idom 的 singleton。对设计模型安装有用的 Binary Hack 是很有效的。

在这里，只是采用了“对象在真正必要阶段生成对象，并初始化”这种典型的 singleton。singleton 决定怎样生成或废弃 singleton 比较困难，要根据具体情况来决定。

参考文献

1. 《在对象指向中为了实现再利用的设计模型》(本位田真一、吉田和树翻译，soft bank creative)
2. 《Double-Checked Locking, Threads, Compiler Optimizations, and More》(*http://www. nwcpp.org/Downloads/2004/DCLP_notes.pdf*)

这是探讨关于 C++ 中双重检查锁问题点的文献。

—— *Yusuke Sato*

理解 g++ 的异常处理（throw 篇）

介绍了 g++ 的 throw 是如何进行处理的。

C++ 的异常处理包含只用 ANSI C 的话写不出来的处理。

- 一边读出本地变量的析构函数，一边大范围的抛出异常
- 抛出对象的管理

在本 Hack 中介绍了 g++ 是如何进行这些处理的。

throw 处理

throw 会执行以下处理：

- 读出 __cxa_allocate_exception，分配例外对象用的内存。
- 读出 __cxa_throw，执行例外 handler 的检索和读出。

__cxa_throw和__cxa_allocate_exception都是包含在libstdc++里的库函数，安装是在GCC 的代码中的libstdc++-v3/libsupc++/eh_throw.cc、eh_alloc.cc 中。

在“[Hack #39]]理解 g++ 的例外处理（SjLj 编）”中有关于例外处理的详细介绍。

例如，下面这个函数，

```
int func ( void )
{
        throw 0xff;
}
```

不使用 throw 来改写的话，会变成下面这样。

```
#include <typeinfo>
#include <stddef.h>

extern "C" void __cxa_throw (void *thrown_exception, const
std::type_info *tinfo,
void (*dest) (void *) );
extern "C" void *__cxa_allocate_exception( size_t size );

int func( void )
{
        void *throw_obj = __cxa_allocate_exception( sizeof(int) );  // - (1)
       *(int*)throw_obj = 0xff;                                     // - (2)
```

```
        __cxa_throw( throw_obj, &typeid(int), NULL );                    // - (3)
    }
```


在(1)这一部分中，正在分配抛出对象的区域。__cxa_allocate_exception的变量是分配的大小。

在（2）这一部分中，正在设定抛出对象的值。由于投放对象是int型的值0xff，所以这样设定。

在（3）这一部分中，正在进行实际的抛出处理。__cxa_throw的变量是面向抛出对象、该对象的类型信息、整理已抛出对象函数的pointer。在g++中由于是在catch的部分进行整理，所以在此将最终变量设为NULL。

Itanium C++ ABI

这里的异常处理的接口由被称为是由“Itanium C++ ABI”（*http://www.codesourcery.com/cxx-abi/abi.html*）的ABI例外处理的部分（*http://www.codesourcery.com/cxx-abi/abieh.html*）来决定。

总结

介绍了在实际中throw是如何处理的。

实际上throw是通过读出包含在GCC的libstdc++中的库函数来实现的。

—— *Takashi Nakamura*

HACK #39 理解g++的异常处理（SjLj篇）

介绍了使用了g++的SjLj情况中安装例外的详情。

在进行C++异常处理时，大范围脱溢的组成是不可缺的。在g++中，为了实现这个大范围脱溢，要准备Unwind-SjLj、Unwind-dw2这两种大范围脱溢的组成。

在这里，将介绍Unwind-SjLj。Unwind-SjLj在Cygwin Mingw环境中运用。同时，若在生成GCC向configure传送--enable-sjlj-exceptions选项的话，在其他环境中也可以使用。

想要知道当前的GCC在使用哪个SjLj的话，可以用如下的方式来查询。

```
$ gcc -v
```

输出结果中有文字列 --enable-sjlj-exceptions 时，其编译器在使用 SjLj。

接下来的运作将会附上 --enable-sjlj-exceptions 而编译的 GCC 版本 3.4.4 在 i686 的 Linux 上驱动执行。

理解 Unwind-SjLj

Unwind-SjLj的安装大致如以下这样。(在GCC源的gcc/un wind.inc 中安装。由 _Unwind_RaiseException 探索 handler，由 _Unwind_Raise Exception_Phase2 来运行 clean up)。

- 将判定是否 catch 的函数（被称为personality）和catch之后记录在某处 longjmp 的 jmp_buf 一起合成为 SjLj_Function_Context，与目录相连。
- throw 后将寻找 SjLj_Function_Context 的目录，之后运用 personality 来判定是否为可捕捉类型。
- 若是可捕捉类型，则再次追寻SjLj_Function_Context的list，执行整理。
- 整理完成后，longjmp到包含在SjLj_Function_Context中的jmp_buf上。

和实际上的catch处理有所不同，但为了理解使用SjLj的大范围脱溢，所以在此介绍只进行 catch 的程序。

```
#include <unwind.h>
/* unwind.h是包含有关大范围脱溢的多种类型的定义和原型声明的帧头 */

#include <setjmp.h>
#include <stdio.h>

struct SjLj_Function_Context /* SjLj_Function_Context 的定义在
gcc/unwind-sjlj.c 中*/

{
    struct SjLj_Function_Context *prev;
    int call_site;
    _Unwind_Word data[4];
    _Unwind_Personality_Fn personality;
    void *lsda;
    jmp_buf jbuf __attribute__((aligned));
};

static _Unwind_Reason_Code my_personality(int code,
                        _Unwind_Action act,
                        _Unwind_Exception_Class cls,
                        struct _Unwind_Exception *e,
```

```
                        struct _Unwind_Context *ctxt ) {
    // 判定是否能catch 。在这里，一般认为可catch 。

    if ( act & _UA_CLEANUP_PHASE )
        return _URC_INSTALL_CONTEXT;
    else if ( act & _UA_SEARCH_PHASE )
        return _URC_HANDLER_FOUND;
    /* 无论如何不会至此 */
    return (_Unwind_Reason_Code)0;
}

void catch_func( void ) {
    struct SjLj_Function_Context sjlj;
    sjlj.personality = my_personality;
    sjlj.lsda = NULL;
    if ( __builtin_setjmp(sjlj.jbuf) == 1 ) {
        puts("catch");
    } else {
        _Unwind_SjLj_Register( &sjlj ); // 在此将sjlj与目录相连
        throw 0;
    }
    _Unwind_SjLj_Unregister( &sjlj );
}
int main( ){ catch_func(); }
```

读出 _Unwind_SjLj_Register 后，sjlj 则与目录相连。

进行 throw 后，personality 将附上参数 _UA_SEARCH_PHASE 被读出从而进行是否能 catch 的判定。

在这里参数为 _UA_SEARCH_PHASE 的情况，一般认定为能 catch，返回到 _URC_HANDLER_FOUND（关于 personality 后面有详细说明）。

判定为可 catch 后则进行 longjmp，然后返回到已 setjmp 过的地方。

最后，作为整理，读出 _Unwind_SjLj_Unregister，从 SjLj_Function_Context 的目录中除去 sjlj。

这个程序只执行了 catch 。

personality

g++ 使用的大范围脱溢的组成实际上并不是为 C++ 所专用。为了与各种语言相符，故分为以下两个处理。

- 大范围脱溢时的整理
- 检测是否为相符的 handler

通过变更SjLj_Function_Context的构成要素personality，可更新这个处理。

在检测是否为相符的handler时，将_UA_SEARCH_PHASE传送到第2个变量上，读出personality。 此时，若personality能返还_URC_HANDLER_FOUND，则被视为相符的handler。

整理时，将_UA_CLEANUP_PHASE传送到第2个变量上，读出personality。此时，要是personality返还了_URC_INSTALL_CONTEXT，则终止整理，执行大范围脱溢。

虽然在上面的my_personality中，没有过什么特别的操作，但在使用实际的C++的异常处理的情况下，在读出这个personality时，必须进行以下的处理。

- 检测是否为可catch型
- 读出本地变量的destructor

在g++中，作为与C++异常处理相应的personality，具备了库函数__gxx_personality_sj0（__gxx_personality_sj0的安装，在GCC源的.libstdc++-v3/ libsupc++/eh_personality.cc中）。

语言特定数据区域

能用__gxx_personality_sj0检测是否为可catch型，但上面程序的catch_func函数中，没有任何关于可catch型的信息，那么该怎样来介绍catch型信息呢？

在__gxx_personality_sj0中，关于是否为catch类的信息包含在了LSDA（语言特定数据区域）中。LSDA的构造如下所示（使用SjLj 生成g++的LSDA的情况，实际上，LSDA的构造会因使用方法的不同而不同）。

```
struct lsda {
    unsigned char lpstart_format;    // landing_pad起始地址的格式化
    unsigned char ttype_format;      // 类型信息地址的格式化

    unsigned char type_offset;       // 到类型信息为止的偏移

    unsigned char call_site_format; // call-site数据的格式化
    unsigned char call_site_length; // call-site的大小

    /* landing_pad..action_record_table..catch_type 的大小可变 */
```

```
    unsigned char call_site_table[ call_site_length*2 ];
    signed char action_record_table[N];

    const std::type_info *catch_type[N];
} __attribute__((packed)) ;
```

地址的格式化可指定为pcrel（0x10:PC的相对位置）、absptr（0x00：绝对地址）等。在没有特别指定时则变为0xff(omit)。

一般，在g++ 生成的LSDA中变为 lpstart_format=0xff 和ttype_format=0x00。type_offset，是类型信息的相对位置，记录了从type_offset的完成开始，到catch_type的完成的所有偏移。

call-site的数据格式化可进行选择：没有符号则为4字节、有符号则为2字节。在g++生成的LSDA中变为uleb128。uleb128是用于DWARF中的可变长数据的表现。将8bit(1byte)中的7bit作为数据使用，剩余的1bit则作为是否有后继数据的flag来使用。带符号时则为sleb128，不带符号时则为uleb128。

在call_site_table中，记载了发生了异常的位置和使用哪个action_record的方法。

```
try { t(); } catch ( int x ) { return x;}
try { t(); } catch ( char x ) { return x; }
```

在这种情况下，要生成int用的action_table和char用的action_table以及与这两个相对应的call_site_table。

action_record_table用于在handler方面区别哪种类型被catch了。类型信息和识别号码相对应。

以上述的为基础来定义LSDA，用__gxx_personality_sj0来更新最初的程序。首先，定义LSDA。

```
struct lsda {
    unsigned char start_format; // 0
    unsigned char type_format; // 1
    unsigned char type_offset; // 2
    unsigned char call_site_format;// 3
    unsigned char call_site_length;// 4
    unsigned char call_site_table[2]; // 5
    signed char action_table[2]; // 7
    const std::type_info *catch_type[1]; // 9
} __attribute__((packed)) ;
static struct lsda my_lsda = {
    0xff,
```

```
        0x00,
        10, // 从type_offset 的完成(3)开始到catch_type
               的完成(13)为止的offset.
        0x01,
        2,
        { 0,1 },
        { 1, 0 },
        &typeid( int ), /* 可捕捉类型 */
};
extern "C" _Unwind_Reason_Code __gxx_personality_sj0( int, _Unwind_Action,
                                _Unwind_Exception_Class, struct
_Unwind_Exception *,
                                struct _Unwind_Context * );
```

利用 Unwind-SjLj 时，要将其放到 SjLj_Function_Context 的 lsda member 中。

```
< sjlj.personality = my_personality; /* 将其更新 */
< sjlj.lsda = NULL;

> sjlj.personality = __gxx_personality_sj0;
> sjlj.lsda = (void *)&my_lsda;
> sjlj.call_site = 1;
```

然后，用 sjlj.call_site 在 call_site_table 中指定使用第几个 call_site。此时，由于只有1个 call_site，所以要设定1（call_site 的索引变为 1 origin）。

接下来，g++ 将会读出要使用的 personality，也会进行类型判定和读出途中的 destructor。

catch 后对象的处理方法

可以用以下方式取得 catch 后的对象。

```
if ( __builtin_setjmp(sjlj.jbuf) == 1 ) {
    void *thrown_obj = __cxa_begin_catch( (void*)sjlj.data[0] );
    printf("%d\n",*(int*)thrown_obj);
    __cxa_end_catch( );
}
```

首先，在回到被 setjmp 过的地方时，要将包含与异常相关的内部信息的对象放入到 sjlj 的构成 data 的第 0 位上。读出 __cxa_begin_catch，在将其内部信息处理到原来的位置上后，将已抛出的对象返回。（__cxa_begin_catch 的安装，在 libstdc++-v3/libsupc++/eh_catch.cc 中）。

这里的返回对象，就像“[Hack #39]理解 g++ 的例外处理（throw 编）”中介绍过的那样，是用 __cxa_allocate_exception 分配的对象。

最后，为了整理，要读出__cxa_end_catch。__cxa_end_catch在处理异常时，会释放已分配的对象和已抛出的对象。

执行 catch 的程序

可从 catch 的判定开始一直执行到 catch 的完成。

```
void catch_func( void ) {
    try {
        throw 100;
    } catch ( int x ) {
        printf("%d\n",x);
    }
}
```

在上面的程序中，若由于不能使用 catch 而改写程序的话，则会变为下面这样。

```
#include <unwind.h>
#include <setjmp.h>
#include <stddef.h>
#include <stdio.h>
#include <typeinfo>

struct SjLj_Function_Context
{
    struct SjLj_Function_Context *prev;
    int call_site;
    _Unwind_Word data[4];
    _Unwind_Personality_Fn personality;
    void *lsda;
    jmp_buf jbuf __attribute__((aligned));
};

struct lsda {
    unsigned char start_format; // 0
    unsigned char type_format; // 1
    unsigned char type_length; // 2
    unsigned char call_site_format;// 3
    unsigned char call_site_length;// 4
    unsigned char call_site_table[2]; // 5
    signed char action_table[2]; // 7
    const std::type_info *catch_type[1]; // 9
} __attribute__((packed)) ;
static struct lsda my_lsda = {
    0xff,
    0x00,
    10,
    0x01,
    2,
    { 0,1 },
```

```
    { 1, 0 },
    &typeid( int ), /* 可捕捉类型 */
};
extern "C" _Unwind_Reason_Code __gxx_personality_sj0( int, _Unwind_Action,
                            _Unwind_Exception_Class, struct
_Unwind_Exception *,
                            struct _Unwind_Context * );
extern "C" void *__cxa_begin_catch (void *exc_obj_in) throw();
extern "C" void __cxa_end_catch ();

void catch_func( void ) {
    struct SjLj_Function_Context sjlj;

    sjlj.personality = __gxx_personality_sj0;
    sjlj.lsda = (void *)&my_lsda;
    sjlj.call_site = 1;

    if ( __builtin_setjmp(sjlj.jbuf) == 1 ) {
        void *thrown_obj = __cxa_begin_catch( (void*)sjlj.data[0] );
        printf("%d\n",*(int*)thrown_obj);
        __cxa_end_catch( );
    } else {
        _Unwind_SjLj_Register( &sjlj );
        throw 100;
    }

    _Unwind_SjLj_Unregister( &sjlj );
}

int main( ) {
    catch_func( );
}
```

总结

介绍了关于使用 SjLj 时安装异常的一些内容。

异常处理不是对象消失那样不可思议的现象，而是可巧妙处理的普通程序。

—— *Takashi Nakamura*

HACK #40 理解 g++ 的异常处理（DWARF2 篇）

对使用了 g++ 的 DWARF2 的信息的大范围脱溢以及其例外处理作了说明。

Unwind-dw2，是运用了调试信息用的格式 DWARF2 的大范围脱溢的组成。通过使用 Unwind-dw2，在只要不发生异常时成本便可实现大约为 0 的大范围脱溢。关于成本（cost），请参照“**[Hack #41]**理解 g++ 例外处理的成本”。

在这里，对使用了DWARF2的信息的大范围脱溢以及异常处理作了相关说明。

DWARF2

DWARF2（Debug With Arbitrary Record Format Version 2）即调试任意记录格式v2，是用于调试的信息格式的规格。该规格可在（*http://dwarf.freestandards.org/*）中得到。

在DWARF2中，决定了类型、文件位置以及帧信息等的格式。Unwind-dw2将利用其中的帧。g++在生成时，会根据DWARF2的格式一同生成已生成代码的帧信息。用g++来编译程序会在`eh_frame_section`中发现`.LSFDExx`和`.LSCIExx`等label。这便成为已生成的帧信息。

这个帧信息可根据程序计数器来获取寄存器的状态和堆栈帧的状态。明白了堆栈帧的状态的话，则就知道了函数的返回地址。由于从程序计数器中获取帧的状态，如果知道了返回地址，则可以再次从那里得到读出函数前的堆栈的状态。这样反复操作的话便可以恢复函数读出。

Unwind-dw2上的大范围脱溢，可通过进行该帧信息的返回来实现。

DWARF2的帧信息

简单介绍一下DWARF2的帧信息。

DWARF2的帧信息，为字节代码程序。关于这个字节代码，在编译时附上`-s`、`-dA`选项，见到输出的assembly后附上命令便可以见到字节代码。

```
.byte    0x4    # DW_CFA_advance_loc4
.long    .LCFI0-.LFB2
.byte    0xe    # DW_CFA_def_cfa_offset
.uleb128 0x8
.byte    0x85   # DW_CFA_offset, column 0x5
.uleb128 0x2
.byte    0x4    # DW_CFA_advance_loc4
.long    .LCFI1-.LCFI0
.byte    0xd    # DW_CFA_def_cfa_register
.uleb128 0x5
```

像这样，在`DW_CFA_`上开始的名称则为命令。这个程序通过更新寄存器和帧的状态，了解了运行中帧的构造，在返回地址中得到了大范围脱溢的信息。解释器的安装在`gcc/unwind-dw2.c`中。

帧信息的分类

为了实际进行大范围脱溢的处理，必须从程序计数器中检索帧信息。在这里，由于要进行两部分的检索，所以将帧信息预先分类。

在链接时进行帧信息的分类。向ld传送--eh-frame-hdr选项后，ld会检测eh_frame选项，将帧的起始地址按顺序分类后的结果保存在.eh_frame_hdr选项中。该安装在binutils的bfd/elf-eh-frame.c中。

异常table

从以上的解说中可以知道，只要有了DWARF2的帧信息，执行堆栈的返回，便可进行大范围脱溢。接下来，将介绍使用了Unwind-dw2的情况中是如何实现异常处理的。

使用了Unwind-dw2之后，异常处理的大致流程和Unwind-SjLj相同。先读出personality，然后判定类型，读出destructor，最后安装异常handler。但Unwind-dw2中handler的检索方法和使用了Unwind-SjLj时的情况有所不同。使用了Unwind-dw2的异常处理中，要通过异常table来进行handler的检索。

异常table，是记录了异常的抛出场所和异常handler的地址相符的table，例如：在下面的程序中。

```
int func() {
    try {
        statements...;                          -+
        try {              -+                    |
            func();         |  (1)                |
            ...;           -+                    | (2)
        } catch ( Obj *p ) { return func(); }     |
                                                  |
        statements...;                          -+
    } catch ( int i ) {
        return i;
    }
}
```

在(1)中，在抛出了Obj*型的异常时运行func();在(2)中，在抛出了int型的异常时，运行return i。在异常table中，分别有相应记录。

编译下面的程序，寻找下面的hogehoge函数的读出。

```
extern "C" void hogehoge();
```

```
void func( ) {
    try { hogehoge(); } catch ( int i ){}
}

.LCFI2:
.LEHB0:
    call hogehoge
.LEHE0:
```

这里的函数调用，是在.LEHB0到.LEHE0的区间进行。异常table作为LSDA的一部分存在于选项中。找到.gcc_except_table后，便可找到下面的这些程序。

```
    .section .gcc_except_table,"a",@progbits
    .align 4
.LLSDA2:
    .byte    0xff
    .byte    0x0
    .uleb128 .LLSDATT2-.LLSDATTD2
.LLSDATTD2:
    .byte    0x1
    .uleb128 .LLSDACSE2-.LLSDACSB2
.LLSDACSB2:                    ; LFB2是函数的最初
    .uleb128 .LEHB0-.LFB2      ; 到捕获异常的范围的开始位置为止的offset
    .uleb128 .LEHE0-.LEHB0     ; 捕获异常的范围大小
    .uleb128 .L7-.LFB2         ; 异常handler的offset
    .uleb128 0x1               ; int型信息(_ZTIi)参照用
    .uleb128 .LEHB1-.LFB2      ; 这是g++生成的库函数(_Unwind_Resume读
出)用的table
    .uleb128 .LEHE1-.LEHB1
    .uleb128 0x0
    .uleb128 0x0
.LLSDACSE2:
    .byte 0x1
    .byte 0x0
    .align 4
    .long _ZTIi
.LLSDATT2:
```

很明显，这里记录了捕获异常的范围和相应的handler的地址。

总结

Unwind-dw2的异常处理，以如下方式安装：

- 从程序计数器中取得帧信息
- 从包含在帧信息中的LSDA中取得例外table
- 检索是否存在与已抛出的类型相符的handler，存在的话则运行其handler

- 若不存在与已抛出的类型相符的handler时，根据帧信息，取得读出的原地址
- 取得读出原地址的帧
- 反复操作直到能捕获为止

帧信息和异常table，能在编译时静态决定，使用了Unwinddw2的异常处理，这样的话，可实现“只要不发生异常，成本便大致为0”。

—— *Takashi Nakamura*

理解 g++ 异常处理的成本

介绍了各种情况中异常处理的必要成本。

异常处理中，究竟需要多少成本，在这里，介绍的各种情况中，究竟多少成本是必要的。

不调用函数，不使用 try-catch 的函数成本为 0

```
void func( int x ) { return x + 3; }
```

此种情况中，大小和运行时间都没有成本。

读出抛出异常的函数时要在每个函数上追加帧信息

```
extern void f();
void g() { f(); }
```

使用了DWARF2的异常处理时，为了要在返回stack上追加必要的帧信息，要增加size（大小）。运行时间不变。在使用了SjLj的异常处理时，以及大小在以下这种函数没有明示出抛出异常时，运行时间和成本均不变。

```
extern void f() throw ();
void g() { f(); }
```

在 try block 内有可能发生异常的情况下，要追加 catch 的处理

```
extern void f();
void g() { try { f() } catch ( ... ) { ... } }
```

这时，会产生由catch产生的成本，不论是使用SjLj还是使用DWARF2，由于必须要指定catch类型，所以LSDA（Language Specific Data Area）也是必需的，相应的size也变大。

使用SjLj的情况

每经过try-catch时都要进行SjLj_Function_Context的setup处理，为了确保SjLj_Function_Context在stack上则要花费掉stack region。

使用DWARF2的情况

和SjLj有所不同，由于不需要setup处理，所以在异常不发生时（只通过try节时），运行时间一般不会发生变化（只增加为了超过handler的jump命令），由于异常table必不可少，因而和SjLj相比，LSDA的size会增大。

需要和try-catch同等的成本时

必须要读出destruction的情况

```
struct C { ~C(); } ;
extern void t();
void func() { C c; t(); }
```

这个程序实际上将进行以下的处理，因而和try-catch相同的成本必不可少。

```
void func () {
    try { t();}
    catch (...) { c.~C(); throw; }
    c.~C();
}
```

用带有throw()的函数读出可能抛出异常的函数的情况

```
void t(); // 可能抛出异常

void func() throw () {
    t();
}
```

这个程序实际上将进行如下处理，因而和try-catch相同的成本必不可少。

```
void func() {
    try { t(); } catch (...) { std::__cxa_call_unexpected( ... ) }
}
```

就发生异常后的处理时间而言，SjLj 的要快

使用 Unwind-SjLj，只是要在 try 的地点顺着 SjLj_Function_Context，但是在 Unuind-dw 2 中的大范围脱溢，却必须要顺着堆栈帧逐个检查帧信息，并执行返回 stack，不管有没有过 try 处理，经过的函数要全部处理，并且使用了 Unwind-dw2 后，就必须从程序计数器中检索帧信息。

```
#include <stdio.h>
int t() { throw 0xff; }

int recursive( int i ) {
    if ( i==0 ) { t(); }
    else recursive( i-1 );
}

#define N 65535
int c( int n ) {
    int i;
    for ( i=0; i<N; i++ ) {
        try { recursive(n); }
        catch ( ... ) {}
    }
}

int main() {
    c( 3 );
    return 0;
}
```

用这样的程序来实验，在笔者所用的环境中，使用 dw2 大约要用 1.3 秒，而使用了 SjLj 则只需大约 0.04 秒的时间。

而且，在再返回程度有所变化时，使用 SjLj 后时间不会有什么变化，而使用 dw2 后，在加深再返回时就会变慢。在 c(3) -> c(30) 中，用 SjLj 大约 0.04 秒而用 dw2 则用了 5.67 秒的时间。

总结

对 DWARF2 而言，只要异常不发生，可用与异常不存在时相同的速度来运行，而 SjLj 则有若干的 overhead，在发生了异常时，SjLj 要比 DWARF2 快。

—— *Takashi Nakamura*

第4章

安全编程Hack

Hack #42～57

当前，对编程而言，如何书写安全的程序是最重要的课题。在本章中，介绍了一些用C以及C++来进行安全程序编码的实用技巧。首先介绍GCC所提供的各种安全强化功能，然后介绍在GCC编写C/C++程序时必须要注意的事项，最后将介绍有用的安全程序编写工具。

GCC提供的安全防范功能随着GCC版本的升级而不断优化，Valgrind是检测与内存有关bug的固定工具，在编写安全程序方面知道这些技巧与不知道这些技巧是有很大差别的。

GCC安全编写入门

介绍了基础中的基础——GCC的警告选项和__attribute__的活用方法。

随着连接网络以及处理个人信息的软件的增多，安全编程逐渐成为大多数程序员的必要技能。在本章中介绍了与安全编程相关的Hack。

本章的内容

软件的安全防范缺陷，有以输入认证功能为代表的要件定义水平错误和在宏上产生的错误等，当然在编码中也会产生，还有“罪魁祸首就潜藏在细小的地方（庞大源代码中的几小行）”这样的说法。

从庞大的源代码中发现缓冲区溢出、整数溢出以及竞争这些微妙的问题并且不让这些问题发生而进行安全编码的操作大都是很困难的，但对于精通

CPU、OS、二进制格式化语言处理对象等知识的黑客（Binary Hacker）而言，也是让他们充分发挥才能的地方。

前提知识

本章的一些 Hack，是以 GCC 和 glibc 的使用为前提，在此，介绍一下进行安全的编码时，GCC 和 glibc 的基本使用方法。

GCC 的警告选项

编写安全代码的第一步，就是要注意编译器的警告，在书写安全代码时，要注意下面的警告选项是有效的。

`-Wall`

显示所有被认为是“有益的”的警告信息

`-W`

表示用 `-Wall` 不能表示的某些附加警告，这个选项在 GCC4 中名称变为 `-Wextra`。

`-Wformat=2`

在 `printf` 函数的格式字符串不是字符串常量时显示警告，该函数可以用来发现 `"format-string bug"` 这个安全漏洞，要注意避免用变量来提供格式字符串。

`-Wstrict-aliasing=2`

警告在 C 语言规定上以不允许的方法进行的内存访问，和 `-O2` 一起使用。

严格别名规则（strict-aliasing rul）

`-Wstrict-aliasing=2` 是在对重叠的两个内存区域分别用 A、B 两个不同类型来读写时显示警告的选项，由于这种读写在 C 语言的规格中是禁止的，所以除非是有计划地执行，否则最好避免。在只有 `-Wall` 程度的警告时，若明确地指定了 `-Wstrict-aliasing=2`，则会进行严格的检查。

```
int main(int argc, char **argv) {
  /*用 short(=16bit)书写*/
  ((short*)&argc)[0] |= 1;
  ((short*)&argc)[1] |= 1;
  /* int(=32bit)..... */
  printf("%d\n", argc);
  return 0;
}
```

这个代码会因优化的不同而改变执行结果，这是由于在优化时，GCC要重新排列改变变量的读写，请用gcc -S 和objdumb -d来确认怎样重新排列。

```
% gcc     alias.c && ./a.out
65537
% gcc -O2 alias.c && ./a.out
1
```

适当地运用警告选项，可检测出该问题。

```
% gcc -O2 -Wstrict-aliasing=2 alias.c
alias.c:4: warning: dereferencing type-punned pointer will break
strict-aliasing
rules
```

GCC 的警告选项（用法实例）

笔者通常使用下面这样的选项，可作为参考。

```
% gcc4 -Wall -Wextra -Wformat=2 -Wstrict-aliasing=2 \
       -Wcast-qual -Wcast-align -Wwrite-strings -Wconversion \
       -Wfloat-equal -Wpointer-arith -Wswitch-enum foo.c
```

在g++中，还有在此基础上，利用-Woverloaded-virtual和-Weffc++的情况。另外，在编译中使用了很多类型转换的C/C++代码时，要在-O2的后面指定-fno-strict-aliasing，对优化进行弱化。具体用法请参照GCC的目录。

GCC 的 __attribute__((format))

自动生成取得printf函数互换的格式字符串的函数时，要活用“[Hack #22] GCC的GNU扩展入门”中介绍的称为"format"的属性。

```
__attribute__((format(printf, 1, 2))) void my_printf(const char
*my_format, ...) {
  va_list ap;
  assert(my_format != NULL);
  va_start(ap, my_format);
    vprintf(my_format, ap);
    vsyslog(LOG_ERR, my_format, ap);
  va_end(ap);
}
```

在这里，数字1表示函数变量中的格式字符串的位置，2表示可变长变量的起始位置。这样一来，由-Wformat=2产生的警告在自动生成的printf系函数中也同样适用。这样还提高了format-string bug的发现率。

在 glibc 中的堆一致性检测（Heap Consistency Checking）

glibc 在缺省情况下具备运行时检测错误的 realloc/free 函数的功能。例如：在两次用 malloc 存储的内存释放后，便可以检测出来。

```
% gcc -Wall -Wextra -Wformat=2 -Wstrict-aliasing=2 -o double_free *.c
% MALLOC_CHECK_=1 ./double_free
malloc: using debugging hooks
*** glibc detected *** free(): invalid pointer: 0x08d04008 ***
```

若能顺利进行编译是很好的，要注意在运行时可用环境变量MALLOC_CHECK_来控制该检查功能是否生效。在 MALLOC-CHECK_ 中能指定的数值如下：

MALLOC_CHECK_	检测出异常时的运作
1	在 stderr 显示警告信息并继续运行
2	立即中止

从命令行开始运行时最好选 1，在 gdb 上执行调试时最好选 2。

```
% gdb ./double_free
(gdb) set environment MALLOC_CHECK_=2
(gdb) run
...
```

内存的二重释放是错误运行和安全漏洞出现的原因，因而这个检测非常有用。

总结

书写安全的代码的确很难，但若能最大限度的活用GCC和glibc功能的话就很有趣了。在本Hack中，将GCC的警告选项和__attribute__的使用方法作为基础中的基础作了介绍。

参考文献

- “Secure Coding in C and C++”（*http://www.cert.org/books/secure-coding/*）

  是有关安全编程的最新书籍，作者是CERT（电脑紧急对应中心）的成员。

—— *Yusuke Sato*

用 -ftrapv 检测整数溢出

使用了 GCC 的 -ftrapv 选项后，便可在运行时检测出带符号的加减乘除中整数溢出。

在本 Hack 中，介绍了自动检测整数溢出的 GCC 功能和 -ftrapv 选项，这个选项可在 GCC 的 3.4 及以上的版本中使用。

整数溢出

整数间四项运算的结果在该整数型中超出了表现可能值的上限或是下限的称为“整数溢出”。例如，下面的 atoi(3) 也自动生成了函数。

```
//
// my_atoi.c
//
int my_atoi(const char *s) {
  int ret = 0;
  assert(s != NULL);
  while(*s != '\0' && isdigit(*s)) {
    const int dig = *s - '0';
    ret *= 10;
    ret += dig;
    ++s;
  }
  return ret;
}
```

如果这个函数的参数 s 是像 "123" 这样的小数字时可以正常运行。但在 Linux/x86 等 ILP32 环境中，若所给的是超过了 "2147483647" 的数字则会错误运作，若提供的是 "4294967297" 则会返回为 1。

由于可能会返回这种非预期的值，（在使用了该值的其他处理中）就会经常发生缓冲区溢出等致命的问题。例如：若在解析 HTTP 的 Content-Length 值时若使用了上述的不完整的 atoi 函数，便可能会引起安全方面的问题。

GCC 中自动检测整数溢出

使用方法

使用了 GCC 的编译选项，便可在运行时检测带符号的整数间的加、减、乘运算时发生的溢出。对于想要检测溢出的程序，请附上 -ftrapv 的 -g 选项予以编译。

```
% gcc -ftrapv -g -o my_atoi my_atoi.c
% ./my_atoi 4294967297
Aborted
```

检测出溢出后，要将程序终止。

溢出的特定场所

用 -ftrapv 选项编译后的程序，在检测到溢出后会对 SIGABRT 进行提升。因此，在 gdb 上运行程序，用 SIGABRT 在停止阶段表示追踪后，便可知道源代码在哪里发生溢出。

```
Program received signal SIGABRT, Aborted.
0x003477a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
(gdb) bt
#0 0x003477a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
#1 0x003877d5 in raise () from /lib/tls/libc.so.6
#2 0x00389149 in abort () from /lib/tls/libc.so.6
#3 0x08048673 in __addvsi3 ()
#4 0x0804841a in my_atoi (s=0x8048863 "7") at my_atoi.c:9
...
```

组成

使用了 -ftrapv 选项后，不是通过 CPU 的指令，而是用包含在 GCC 附属库 libgcc.a 里的函数来执行带符号的整数间的加、减、乘运算，也就是和 soft-float 相似。例如：

```
int sqr(int a) {
   return a * a;
}
```

这个代码，进行如下编译。

```
pushl 8(%ebp)   # a
pushl 8(%ebp)   # a
call __mulvsi3 #运行乘法
```

在函数 __mulvsi3 中，边检测有无溢出，边进行乘法运算。

要注意的问题

注意点 1：存在不能被检测的情况

能用 -ftrapv 检测的只是带符号的整数间的运算。

- 不带符号的整数间的运算

- 带符号的整数和不带符号整数的混合运算
- 从有符号的整数变换到不带符号的整数（或者反过来）
- 这几种情况不能被检测。代入到bit数较少的类型时的舍去

要注意特别是在第2个混合运算中，经常会由于整数溢出而导致安全漏洞。要严格遵守C/C++语言规格的整数变换的顺序、整数扩张、一般的算术型变换等规则。进而写出不依靠GCC而编写的完美代码。

注意点2：不能检测除法运算

-ftrapv选项只能检测加法、减法、乘法运算，不能检测除法运算。这是因为Nbit变量间的除运算结果一般不超过Nbit，虽然是看起来很安全的除法运算，但在下面2种特殊的场合中会发生问题。

- 有0的除法运算
- INT_MIN / -1（在有的环境中除法的结果会超过INT_MAX）

前者为大家所知晓，但对于后者却很容易忽视。在笔者编写的环境（Linux/x86）中试着运行下面的程序，由于SIGFPE（Floating point exception）使程序异常终止。

```
#include <limits.h>
int main() {
    volatile int a = -1;
    printf("%d\n", INT_MIN / a);
    return 0;
}
```

在可能运行这种除法的地方，（由于不能用-ftrapv来检测）请人为插入检测代码。

注意点3：在旧GCC中不能使用-ftrapv

要注意，在GCC 3.3.x以前的版本中安装libgcc.a时发现了问题，即-ftrapv不能正常运作。

总结

使用了GCC的-ftrapv选项后，便可在运行时检测出有符号整数间的加、减、乘运算中产生的整数溢出。这对早期发现软件的安全漏洞是很有帮助的。

参考文献

- "Secure Coding in C and C++"（*http://www.cert.org/books/secure-coding/*）这本书用一个章节对"Integer Security"进行了详细的说明。
- "JIS X 3010:2003 C程序语言"（*http://www.webstore.jsa.or.jp/*）ISO C99规格的日语译文，在第六章中介绍了整数溢出的规则。

—— *Yusuke Sato*

用 Mudflap 检测出缓冲区溢出

Mudflap 是与 C/C++ 语言相符的调试辅助功能。

Mudflap是在GCC4中新出现的调试辅助功能，在本Hack中介绍了Mudflap的调用方法。

Mudflap 的概要

Mudflap 是与C/C++语言相符的调试辅助功能，使用了Mudflap后，便可在程序运行时动态检测与指针有关的程序错误。

- 缓冲区溢出
- 内存泄露
- 空指针异常
- 其他、指针的误用

虽然和"**[Hack #54]**用 Valgrind 检测出内存泄露"、"**[Hack #55]**，用 Valgrind 检测不正确的内存访问"中介绍的Valgrind非常相似，但有如下的几个区别：

	运作环境	解析对象的再编译、再链接	解析速度	heap 变量的检查	stack/data/bss 变量的检查
Valgrind (Memcheck)	x86/x86_64/ ppc + Linux	不要	相对低速	可	当前不可
Mudflap	GCC4 的运作环境	必要	相对高速	可	可

Mudflap只要是在运行有GCC4平台的情况下基本都能使用，不仅是在作者尝试的范围中，在x86/ppc/alpha/sparc和Linux中都可以在GCC-4.0.2的组合中运行Mudflap，还可以检测出heap/stack/data/bss上变量的误用。

就输出的易读性和实用性而言，（作者的主观意见）在刚完成的工具中，还是Valgrind最好用，大家试着用用。

使用方法

为了使用Mudflap，要对将解析程序进行再编译、再链接。

编译和链接

在编译程序时，加入-g -fmudflap选项，在链接时加入-lmudflap选项。在多线程程序中，分别为-fmudflapth、-lmudflapth。

```
% gcc -g -fmudflap -o testflap testflap.c -lmudflap
```

如出现“不能发现mf-runtime.h”或是“不能发现libmudflap”等情况时，要立即追加包，要根据情况来追加，在RedHat Linux中，用下面的命令就可以了。

```
# /usr/bin/yum install libmudflap libmudflap-devel
```

运行

照常运行生成的二进制文件，由Mudflap来运行检测，并在stderr中输出错误信息。

```
% ./testflap
```

若要暂时禁用Mudflap，可利用环境变量MUDFLAP_OPTIONS。

```
% MUDFLAP_OPTIONS="-mode-nop" ./testflap
```

若要将因为Mudflap而引起的运行速度低下问题控制到最小，可用下面的选项。

```
% MUDFLAP_OPTIONS="-no-timestamps -backtrace=0" ./testflap
```

例

“**[Hack #55]**用Valgrind检测内存的错误访问”中，用Valgrind不能检测的话，就用Mudflap检测“stack/bss上的变量访问错误”。

```
//
// testflap.c
//
static char onbss[128];

int main() {
  char onstack[128] = {0};
  int dummy;

  dummy = onbss[128]; // off-by-one bug
  dummy = onstack[128]; // ditto.

return 0;
}
```

用 GCC4.x 来编译、链接

```
% gcc -g -fmudflap -o testflap testflap.c -lmudflap
% ./testflap
```

将输出下面的这两种错误，可用Mudflap安全的检测出用Valgrind不能检测出的bug。

```
mudflap violation 1 (check/read): time=1139169907.370531
ptr=0x80c9b00 size=129
pc=0x3c0332 location=`testflap.c:10 (main)'
      /usr/lib/libmudflap.so.0(__mf_check+0x44) [0x3c0332]
      ./testflap(main+0xc1) [0x8048785]
Nearby object 1: checked region begins 0B into and ends 1B after
mudflap object 0x86451e8: name=`testflap.c:4 onbss'
bounds=[0x80c9b00,0x80c9b7f] size=128 area=static check=3r/0w
liveness=3

mudflap violation 2 (check/read): time=1139169907.371680
ptr=0xbfeb77b0 size=129
pc=0x3c0332 location=`testflap.c:11 (main)'
      /usr/lib/libmudflap.so.0(__mf_check+0x44) [0x3c0332]
      ./testflap(main+0x15b) [0x804881f]
Nearby object 1: checked region begins 0B into and ends 1B after
mudflap object 0x8645e38: name=`testflap.c:7 (main) onstack'
bounds=[0xbfeb77b0,0xbfeb782f] size=128 area=stack check=3r/0w
liveness=3
```

组成

由于Mudflap和GCC相统一，所以在编译时便可知道程序在哪里运行了指针访问。因此，Mudflap会在所检测到的指针访问的周围插入检测访问是否正确的代码（在汇编语言中为命令）。在检测中，也利用了包含在libmudflap.so中的__mf_check等函数。

另外，Mudflap会用包含在libmudflap.so里的函数的执行置换一部分标

准库函数的读出。例如：源代码中的memmove函数的读出会自动置换为执行__mfwrap_memmove函数。

```
% nm testflap2 | grep memmove
U __mfwrap_memmove
```

找到并查看mf-runtime.h文件便可知道将置换什么样的标准库函数。例如在作者拟定的环境中有如下的内容：redefine_extname这个谜一般的pragma则牵动了Binary Hacker们的心。

```
#ifdef _MUDFLAP
  #pragma redefine_extname memcpy __mfwrap_memcpy
  #pragma redefine_extname memmove __mfwrap_memmove
  ...
  #pragma redefine_extname getprotobynumber
__mfwrap_getprotobynumber
#endif /* _MUDFLAP */
```

总结

Mudflap是用于发现与指针相关的bug的工具。只要安装了GCC4及其以后的版本的话，在任何环境中都能使用，而且还能发现用Valgrind所无法检测的bug。

参考文献

- "GCCWiki - Mudflap Pointer Debugging"（*http://gcc.gnu.org/wiki/Mudflap%20Pointer%20Debugging*）。

  Mudflap的公式页。

- "Mudflap: Pointer Use Checking for C/C++ (pdf)" *http://gcc.fyxm.net/summit/2003/mudflap.pdf*。

  有关环境变量MUDFLAP_OPTIONS的详细说明。

—— *Yusuke Sato*

HACK #45 用 -D_FORTIFY_SOURCE 检测缓冲区溢出

使用了GCC的"Automatic Fortification"功能之后，便可在编译或是运行时检测是否误用了容易发生问题的函数。

在C语言中存在着很多像gets strcpy memcpy这样容易引起缓冲区溢出

的函数，利用了在本 Hack 中介绍的“GCC 的自动强化”这个功能之后，便可在编译时或运行时检测因误用了这些函数而产生的缓冲区溢出。

基本的使用方法

为了使用自动强化，必须进行源代码的再编译。像下面这样，进行 -O1 以上的优化后，在使用 -D_FORTIFY_SOURCE=1 选项的情况下进行编译，必要操作就是这些，没有必要进行源代码的更新和库的链接。

```
% gcc -O1 -D_FORTIFY_SOURCE=1 foo.c
```

这样进行编译后，再在下面两个时间点检测缓冲区溢出。

检测时间	检测内容
编译时	编译时可检测的明显的缓冲区溢出
运行时	其他溢出

在检测危险函数的使用中，这些功能看起来和之前经常使用的“libsafe”非常相似，但和用 LD_PRELOAD 插入装过 strcpy 等危险函数的共享库方式的 libsafe 有所不同，自动强化通过 GCC 和 glibc 的合作来实现。

让我们按顺序来看一下自动强化的两种检测方法。

编译时的溢出检测

确保 stack 上有 6 个字节，试着编译进行 7 个字节 strcpy 后的代码。

```
char buf[6];
strcpy(buf, "hello!");// 复制终端含'\0'的7字节
```

便会出示如下的溢出警告。

```
% gcc -O1 -D_FORTIFY_SOURCE=1 foo.c
foo.c:5: warning: call to __builtin___strcpy_chk will always
overflow destination
buffer
```

运行时溢出检测

接着，将6字节作为全局变量保存，然后试着用复制命令行参数（argv[1]）的值的代码来运行。

```
static char buf[6];
int main(int argc, char **argv) {
```

```
    strcpy(buf, argv[1]);
    return 0;
}
```

在编译时不知道argv[1]的值，所以编译时若没有出现警告则为成功。但和一般的二进制不同，如果在bar运行时给予了6字符以上的参数的话，bar将给出错误信息并中止。

```
% gcc -O1 -D_FORTIFY_SOURCE=1 -o bar bar.c
% ./bar 12345 (5个或5个以内没有问题)
% ./bar 123456 (若有6字符的话…)
*** buffer overflow detected ***: ./bar terminated
Aborted
```

在这里，将buf作为全局变量没有特别的意思，将buf保存在栈上也可执行检测。一般而言，GCC如果可以在编译时掌握strcpy、memcpy等的检测对象函数的复制地（dest）内存的大小的话，运行时检测非常有效。因此反过来，在如下的代码中完全不能进行检测。

```
// 在运行时决定复制处的内存大小的例子
char *buf = malloc(atoi(argv[1]));
strcpy(buf, "hello!");
```

另外要注意的是，在使用gcc而不是使用g++进行编译的情况下，不能执行任何检测。

组成

与自动强化兼容的GCC会提供__builtin函数_object_size。该函数是在编译时检测变量的大小（长度）等的函数。在编译时，若不知道大小的话，该builtin函数会返回-1。

与之相应的glibc会将-D_FORTIFY_SOURCE情况中的gets的定义变为如下的样子，认定__bos为__builtin_object_size。

```
(从/usr/include/bits/stdio2.h中摘录)
#define gets(__str) \
((__bos (__str) == (size_t) -1)                          \
? (gets) (__str) : __gets_chk (__str, __bos (__str)))
```

在编译时知道了传递给gets函数的__str变量的大小的情况中，将读出函数__gets_chk而不是本来的gets函数。__gets_chk函数包含在/lib/libc.so.6中。由于会将在编译时确定的__str的大小传递给__gets_chk的第2个变量中，所以__gets_chk可在其内部判断是否发生了缓冲区溢出。

检测的强化

将编译时的 -D_FORTIFY_SOURCE=1 增加到 -D_FORTIFY_SOURCE=2，检测则会变得更加严格。格式字符串 bug 会在运行时被检测出来。printf、vfprintf、syslog 等函数将包含有 "%n" 的格式字符串当作变量读出后，便会像如下这样将进程中止。

```
% cat baz.c
#include <stdio.h>
int main(int argc, char **argv) {
  int a;
  printf(argv[1]); // 若argv[1]中包含 %n则中止
  printf("%n", &a); //特例：字符串literal包含 %n则没有问题
  return 0;
}
% gcc -O1 -D_FORTIFY_SOURCE=2 -o baz baz.c
% ./baz %n
*** %n in writable segment detected ***
Aborted
```

由于格式控制字符 %n，多被用于攻击性的目的，因而，这个检测提高了安全性。

被检测函数的一览表

将被检测的函数一一过目，例如用下面的命令就可得到。

```
%grep -r "_chk " /usr/include | sed 's/.*\(__.*_chk\).*/\1/' | sort
| uniq
__confstr_chk
__fgets_chk
..
__wmemset_chk
__wprintf_chk
```

在笔者拟定的环境中，有 65 个函数。

和 Mudflap 的使用区别

自动强化可能会被认为和[Hack #44]中介绍了的 GCC 的 Mudflap 功能相似。在 GCC4.x 可用于检测缓冲区溢出这点上两者是相似，但 Mudflap 是调试用的功能，而自动强化则可在发行版本中使用。

发挥这个功能的效果后，则基本上不会出现运行时开销。在一部分 Linux 发行版中，在其包含的所有的二进制文件中，都附上了 -D_FORTIFY_SOURCE 进行编译，大家在发布自制的软件时，不妨使用这个功能。

总结

使用了GCC/glibc..automatic fortification功能后，可以在编译或运行时检测gets、strcpy、memcpy或printf、vfprintf、syslog这样的容易产生问题的危险函数的误使用。该功能通过GCC和glibc的合作而实现，所以检测的成本很小，而且，也可用于发行版本

—— *Yusuke Sato*

HACK #46 用 -fstack-protector 保护堆栈

利用GCC的stack-smashing protector选项（SSP），可以检测出用C/C++所编写的程序缓冲区溢出。

stack-smashing protector（SSP，别名ProPolice）是IBM的Hiroaki Etoh开发的GCC补丁，使用SSP，便可检测出用C/C++所编写的程序缓冲区溢出。

在IBM的网页上面向GCC（~3.4.4）的补丁被公开以后，就得到了广泛的使用。最近由于通过RedHat可执行面向GCC4的移植操作，所以在GCC 4.1及其以后的版本中没有安装补丁也可使用SSP。

使用方法

在程序上附加-fstack-protector选项并进行编译，SSP就会生效，可用SSP检测保存在堆栈上的char、signed char、unsigned char多余的排列。试着用下面的代码。

```
int main(int argc, char **argv) {
  char buf[8];
  if (argc >= 1) {
    char *s = argv[1], *d = buf;
    while(*s != '\0') *d++ = *s++;
  }
  return 0;
}
```

有效使用SSP并进行编译，将9个字符以上的字符串当作变量并运行程序的话，便可由SSP检测出堆栈多余，然后程序将中止。

```
% gcc -v
Target: x86_64-redhat-linux
gcc version 4.1.0 20060214 (Red Hat 4.1.0-0.27)
% gcc -fstack-protector -o overf overf.c
% ./overf 012345678
```

```
*** stack smashing detected ***: ./overf terminated
Aborted
```

与网络进行连接的应用程序，最好加上 -fstack-protector选项编译。另外，还可将SSP和“**[Hack #45]**用 -D_FORTIFY_SOURCE检测缓冲区溢出”中介绍了的 _FORTIFY_SOURCE 功能一起使用。

组成

SSP 会防止或检测以下的情况

1. 由缓冲区溢出引起的对局部变量的篡改
2. 由缓冲区溢出引起的对返回地址和保存的 ebp 的篡改

在 1、2 中无论是哪个篡改成功都会导致像“运行攻击者传来的代码”这样的致命安全问题。

堆栈对齐的调整

SSP 会将函数的局部变量在堆栈上的位置变成不同于一般 GCC 所生成的结果（注 1）。具体的就像图 4-1 这样，将 char 排列配置在最前面的地址上。由图可知，由于一般的 GCC 中会使排列缓冲区溢出冗余，导致篡改函数指针 fn 的值（箭头指向处），但用 -fstack-protector 编译时则不能篡改 fn。

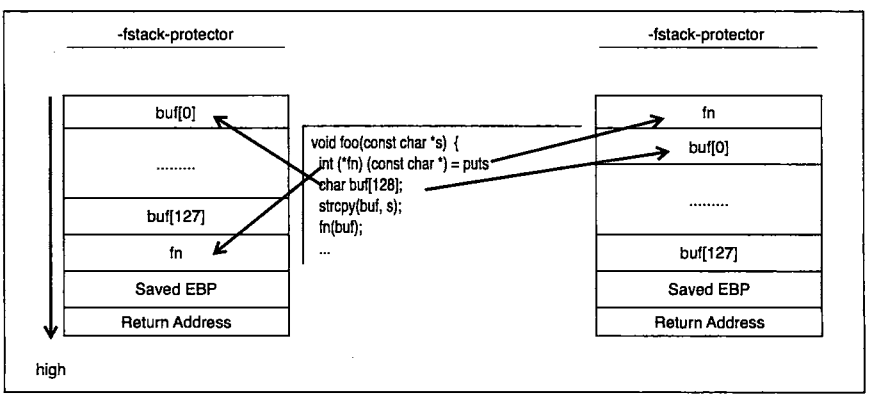


图 4-1：在有无 SSP 时 stack 排列的变化

注 1：实际上 SSP 的函数变量也是排版的调整对象，但这里由于页面空间不够所以省略了。

通过这样的堆栈排版的调整，可防止由缓冲区溢出引起的对局部变量的篡改。

Guard-canary 的配置与检测

让 SSP 在函数的局部变量和保存的 ebp 之间插入 "guard" 值（图 4-2）。

由动态loader随机决定的该值，每次运行程序时都会发生变化，所以不能实现预测。

附上 -fstack-protector选项来编译，则函数 foo的开头部分会变成如下这样。

```
foo:
        ...(面向函数 foo的入口处理)...
           movl    %gs:20, %eax        ←将guard值加载到eax上
           movl    %eax, -8(%ebp)      ←在stack上配置guard值
           xorl    %eax, %eax          ←消去eax的guard值
        ...(之后，函数foo主体的运行)...
```

被插入的 guard 值会在函数 foo 的末尾检测是否被缓冲区溢出问题所覆盖了。

```
           movl    -8(%ebp), %edx      ←将stack上的guard值加载到edx上
           xorl    %gs:20, %edx        ←和original的guard值相比较
           je      .L5                 ←若两者一致则到.L5
           call    __stack_chk_fail    ← stack上的guard值被覆盖了!
                                         读出__stack_chk_fail函数
.L5:
           ...(从函数foo 中的出口处理)...
```

__stack_chk_fail是包含在glibc(以及libssp)里的函数，表示*** stack smashing detected ***信息，并将程序中止。能检测出由缓冲区溢出引起的 "return address" 和 "saved ebp" 被篡改。

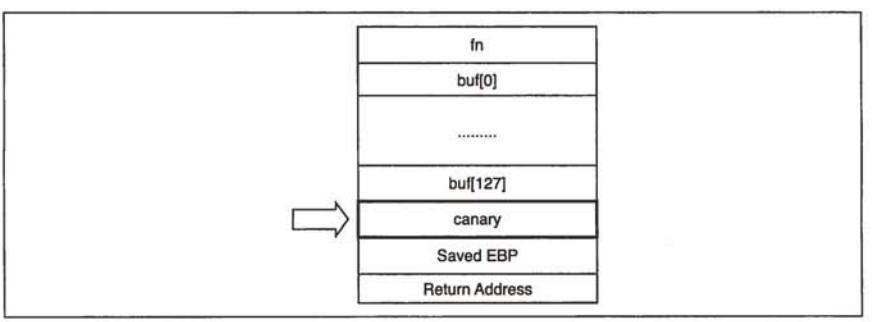


图 4-2：插入 guard 值

运行的调整

SSP 在缺省中只会保护堆栈上有 8 字节以上的 char、signed char、unsigned char 等排列的函数和调用 allocal 的函数。想要将这里的“8 字节以上”的运行调整为“N 字节以上”时，要使用 GCC 的 --param 选项。有和运行速度的 trade-off，会随着 N 的值的变小而增加保护对象的函数。

```
% gcc -fstack-protector --param ssp-buffer-size=N foo.c
```

想要无条件地保护所有的函数时，只要进行如下操作便可以了。

```
% gcc -fstack-protector-all foo.c
```

总结

利用 GCC 的 -fstack-protector 选项（SSP）后，便可在运行时检测出堆栈上的排列受到了溢出攻击，在程序发生由攻击引起的不正常运行之前，将程序中止。

在 GCC4.1 中，sh 也可利用 SSP，在利用 GCC 来安装时，也可以试试 SSP。

参考文献

- “GCC extension for protecting applications from stack-smashing attacks”（*http://www. trl.ibm.com/projects/security/ssp/*）。

  公开的 SSP 补丁（面向 GCC～ 3.4.4 ）

——*Yusuke Sato*

将进行位遮蔽的常量无符号化

在本 Hack 中，介绍了在执行位操作运算时必须注意的几点问题。

位操作运算的例子

看到以下代码，你可以马上发现哪里存在问题吗？

```
unsigned long n = 0;
unsigned char *str;
....
n |= (*str & 0xff) << 24;
```

在最后一行中，由于 *str 是 unsigned char，n 也是 unsigned long，

所以可以安全地将*str的8位值设定在第31-24位中，与此相同的代码包含在libzlibruby（Ruby的zlib binding）中，而它却得到了不可思议的运行结果，即使对于在本来没有问题的被gzip过的文件，也发生了“不正确的CRC错误”（Debian Bug#255442）。

在x86等的ILP32的环境中（int、long和指针都为32位的环境中）会这样运行。但像在amd64（x86_64）这样的环境LP64（long和指针为64bit）中会得到预想外的结果。在生成了*str的MSB的情况中。例如进行0xff时n虽规定要变为0xff000000，但实际运行后n会变为0xffffffffff000000，也就是说得到了意料之外的63-32位这样令人吃惊的结果。

运算的类型提升

为什么会得到这样的结果呢？在C中，数据类型不同的操作数间执行运算时会变换为与这些类型有互换性的某种类型。在此情况中，&运算的操作数为*str和0xff。由于*str为unsigned char，0xff为int，则根据C的整数提升规则会提升为int。这里的问题是*str的unsigned char即便是没有符号，但运算后的结果int会带符号。*str在0xff时(*str & 0xff)的结果变为了带符号0x000000ff的int。将其像(*str & 0xff)<< 24这样，向左移24位的话，结果则变为0xff000000这样带符号的int。

像这样，*str即便是没有符号，其计算后的结果(*str & 0xff) << 24却带有符号。

像i386这样是ILP32的情况中long也为32位则不会产生问题。但在像amd64那样的LP64的环境中long为64位，将其代入到unsigned long的n中时，便会从带符号的32位int向64位带符号的long发生符号扩展为0xffffffffff000000。这将会进行无符号64位的long变量n与每个位的OR运算。

怎样来评价这些类型呢？按顺序观察，如下所示。

```
*src => unsigned char
*src & 0xff => unsigned char & int
            => int & int
            => int
(*src & 0xff) << 24 => int << int
                    => int
n |= (*src & 0xff) << 24; => unsigned long |= int
                          => unsigned long |= long
                          => unsigned long |= unsigned long
```

总结

那么，为了得到预期的结果要编写怎样的代码呢？这里的问题是用于位遮蔽上的常量0xff为有符号的int常量。也就是说将其变为无符号常量的话则不会发生问题。

```
unsigned long n = 0;
unsigned char *str;
....
n |= (*str & 0xffUL) << 24;
```

必须将位遮蔽上使用的常量表现为像0xffU和0xffUL这样的无符号的数值常量。

—— *Fumitoshi Ukai*

HACK #48 注意避免移位过大

移位过大的话，有可能导致意想不到的结果。

像1<<32这样，必须要注意避免移位过大。

下面的程序表示的是将1左移32位的结果。会变成什么样子呢？

```
#include <stdio.h>

int main()
{
  unsigned int w = 32;
  printf("%x\n", 1U << w);
  return 0;
}
```

试着运行，在是否进行优化这一问题上出现了不同结果，在不进行优化时变为1，在进行了优化后则变为0（在x86环境中，编译器为Debian的gcc 4.0.3）。

```
% gcc tst.c
% ./a.out
1
% gcc -O2 tst.c
% ./a.out
0
```

将1左移32位后，虽然很希望清除所有的位变为0，但遗憾的是并不能得到这样的结果。

理由

问题的原因在于x86的位移命令。在上述例子不执行优化的编译结果中，实际上是使用了sall命令。但是sall命令只能在位移宽度低于5bit的情况中可见，低于5bit也就是只能表示从0到31，32则会被视为与0等价。这样一来，将1 << 32看作为1 << 0，结果便得到了1。

然而，在进行了最优化后会出现结果为0。这是由于位移运算在gcc中进行编译时的常量折叠。此时的位移运算与进程的位移命令的运行无关，由于是处理位移宽幅为6 bit以上的问题，所以结果变为0。

而且，用gdb技术时也有与gcc的常量折叠同样的效果。这和编译过的代码的运行有所不同，要引起注意。

```
(gdb) p 1 << w
$1 = 0
```

C语法中这些举动都被许可。这是由于指定将左操作数的偏移量以上的位移偏移量作为右操作数指定时结果未定义。

用途

也许有人会这样想：想要的结果为0的话就不必进行了32位移了。但是，有时会有进行的必要。以建成低于n位的遮蔽为例。这种mask(1 << n)可以作为- 1来构建，也很容易被考虑到。但考虑到1<<32可能会变为0时，就必须变为32 <= n ? 0xffffffff : (1 << n) - 1。

总结

C的移位操作不能将左操作数的宽幅以上的值指定为移位宽幅，由于其结果未定义，所以会招致意想不到的结果。

——*Akira Tanaka*

HACK #49 注意64位环境中0和NULL的不同之处

在本Hack中，介绍了0和NULL表现出来的不同状况。

0和NULL

在C中，将带有0值的指针作为NULL，NULL通常被定义为0或((void

*)0)，甚至存在必须要使用NULL时便写为0的程序。通常，用这样的编写方法也不会发生问题。

```
char *p = 0;
if (p != 0) ...
```

此时，p为char *，进行运算（=或者!=）前，会将操作数的类型转换为与之有互换性的某个类型上，所以int的0会变换为char *型的NULL，这样一来下面的书写方法也没有问题。

```
if (p) ...
```

在此时，if会用布尔值来判断，用表达式的布尔值是否为0来判断。如下所示

```
if (p != 0) ....
```

相反的条件

```
if (!p) ..
```

也和

```
if (p == 0)
```

等价

像这样在多数情况中，将0和NULL采用相同的处理方式也可以。但是，也存在不通用的情况。

0和NULL不同的情况

如前面所说的那样，NULL并不是0这个int的值，带有0这个值的指针为NULL。也就是说编译器必须要明白是用指针类型来解释的。

在i386那样的ILP32环境中由于int、long、指针都是32位，所以int的0通常等同于指针的NULL。但在IA-64那样的LP64环境中int虽是32位，但long和指针都为64位，因此，int的0有时便不等同于NULL。使用可变长参数时这个问题很明显。

例如：以下的代码为例来试着解释。

```
struct s *foo(const char *name, ...)
{
    va_list va;
    struct s *sp;
```

```
    char *p;
    va_start(va, fmt);
    sp = (struct s *) malloc(sizeof(struct s));
    if (!sp) {
       return 0;
    }
    memset(sp, '\0', sizeof(struct s));
    set_name(sp, name);
    while ((p = va_arg(va, char *))) {
        set_item(sp, p);
    }
    return sp;
}
...
    sp = foo("foo", "bar", 0);
...
```

这里的foo()是返回取得可变长参数后利用name这个名称和剩下的参数来设定Item的结构体的函数。这个代码在LP64环境中不能正常运行。

像foo("foo", "bar", 0)那样读出foo()时，参数会像下面这样堆积在栈上。

```
为0的int
面向"bar"的const char *
面向"foo"的const char *
```

在foo()的运行中，首先要利用const char *name来使用第1变量。此时，指向"foo"的指针会变成name，剩余的变量经过va_list来访问，在foo()中va_list的内容会像下面这样取出。

```
p = va_arg(va, char *)
```

也就是参数作为char *而取出，第1个也就是第2个参数由于是指向"bar"的指针所以没有问题。但第2个即第3个参数中只能传送int却想要取得char *。ILP32中由于每个都为32位（4字节），所以没有问题，但在LP64中int为32位（4字节），但由于char *是指针，所以是64位（8字节）便会出现问题。也就是说在读出的这一方中只能将32位（4字节）的0推进到栈上，但在被读出的那一方foo()中，却取出了64位（8字节）的值。4字节为0，于是将剩余的4字节作为栈上的无用物而读出。运气好的话把它们当作0也不会有什么问题，但却将0之外的没有想到的值设定为item。这样继续往下读栈的话应该会发生节失败吧。

总结

这是将0和NULL混淆后产生的bug，必须像下面这样读出foo()。

```
sp = foo("foo", "bar", NULL);
```

这样一来，在读出 NULL 指针时即便是由于堆积在栈上所以在 foo() 的 va_arg(va, char *) 上读取，也可以准确地得出 NULL 指针，故可以在当时终止 while()。

也有自己编写这种函数的情况，例如：利用与此相同的 execl(3) 时，必须要用 NULL 来中止后才能读出。

```
int execl(const char *path, const char *arg, ...);
```

—— *Fumitoshi Ukai*

POSIX 的线程安全函数

有一些不适合在多线程程序中调用的函数，那就准备几个安全的代用函数吧。

在本 Hack 中，介绍了在 UNIX 上编写多线程程序的方法，以及与不遵循此原则的强行安全运行程序的方法。

非安全线程函数（POSIX thread-unsafe functions）

UNIX 规则中，认为下面的 85 个函数不编写为线程安全也没关系（*http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_09.html*）。其理由将在后面说明，但首先请将“在编写多线程程序时，不可以使用这些函数”作为编写的方法来记住，特别是 getenv、gethostby name、gmtime、localtime、rand、readdir、strerror、strtok 是很容易使用的函数，要特别注意。

asctime()	dbm_store()	getchar_unlocked()	getopt()
basename()	dirname()	getdate()	getprotobyname()
catgets()	dlerror()	getenv()	getprotobynumber()
crypt()	drand48()	getgrent()	getprotoent()
ctime()	ecvt()	getgrgid()	getpwent()
dbm_clearerr()	encrypt()	getgrnam()	getpwnam()
dbm_close()	endgrent()	gethostbyaddr()	getpwuid()
dbm_delete()	endpwent()	gethostbyname()	getservbyname()
dbm_error()	endutxent()	gethostent()	getservbyport()
dbm_fetch()	fcvt()	getlogin()	getservent()
dbm_firstkey()	ftw()	getnetbyaddr()	getutxent()
dbm_nextkey()	gcvt()	getnetbyname()	getutxid()
dbm_open()	getc_unlocked()	getnetent()	getutxline()

gmtime()	localtime()	rand()	unsetenv()
hcreate()	lrand48()	readdir()	wcstombs()
hdestroy()	mrand48()	setenv()	wctomb()
hsearch()	nftw()	setgrent()	
inet_ntoa()	nl_langinfo()	setkey()	
l64a()	ptsname()	setpwent()	
lgamma()	putc_unlocked()	setutxent()	
lgammaf()	putchar_unlocked()	strerror()	
lgammal()	putenv()	strtok()	
localeconv()	pututxline()	ttyname()	

(除此之外，.ctermid..tmpnam..wcrtomb..wcsrtombs 在输出的参数中传送 Null 时也为非线程安全的)

除外面列举的89个函数外，希望用规格标准化的函数全部为线程安全的。关于未用规格标准化后的函数（例如zlib 和 libpng的函数）的线程安全性，请确认每个库的源代码和文档。

违背该编写方法的实际损害

在glibc等现实世界的libc中，一般不会将上表中的函数安装到线程安全中，例如，多会将 localtime 函数进行如下安装。

```
struct tm _tmbuf;
struct tm *localtime(const time_t *timer) {
  /* 从 timer 参数开始计算年月日等 */

  _tmbuf.tm_year = XXX;
  /* 将计算结果存放到构造体中 */
  _tmbuf.tm_hour = XXX;
  _tmbuf.tm_min  = XXX;
  _tmbuf.tm_sec  = XXX;
  return &_tmbuf; /* 归还面向全局变量的 pointer!! */
}
```

注意构造体 tmbuf 为静态全局变量，由于静态变量在线程间会被共享，所以线程1和线程2以如下顺序同时读出 localtime 函数时，就会产生错误。线程1不打印出现在（now）的时刻，而是先前（long_time_ago）的时刻。

	线程 1 at foo()	线程 2 at bar()
时刻 t	tm = localtime(now);	
时刻 t+1		tm2 = localtime(long_time_ago);
时刻 t+2	printf("今天是 %d 月 %d 日 \n", tm->tm_mon + 1, ...	

这个例子中的误用虽是“printf出来的值变得奇怪”这种轻微的错误，但在不同函数中可能会成为“线程产生Null指针引用”等致命错误的根源，虽然这种bug的再现性很小，但通常要检测或解决却很难。

安全的代用函数

在UNIX的规范中，定义了一些代替非线程安全的函数供使用，笔者所知道的有如下函数。

```
asctime_r       ctime_r         getgrgid_r      getgrnam_r
getpwnam_r      getpwuid_r      gmtime_r        localtime_r
rand_r          readdir_r       strerror_r      strtok_r
```

这些函数在多线程中使用也不会有问题。例如：想使用localtime函数时，用localtime_r函数代替它就很好。但由于locatime函数和localtime_r函数的签名不同，所以要改写源代码和再编译。未用规格标准化后的函数有时会当作libc的独自扩展被提供，例如glibc中包含着gethostbyname_r等的函数。

一部分的非线程安全函数各线程明确的执行了lock操作可安全的使用。例如：putc_unlocked函数在和flockfile函数一起使用时便可安全使用。关于这一点在本Hack中没有详细的说明，详细信息请确认man page等。

LD_PRELOAD 和 TLS 的 Hack

若不能再编译的二进制分发的软件读出了localtime等危险函数后，该怎么办呢？

```
% nm -D /opt/path/to/broken_proprietary_software | grep localtime
   U localtime （正在读出危险函数的感觉!!）
```

在这种情况下，将《**[Hack #60]**用LD_PRELOAD更换共享库》中介绍了的Hack和《**[Hack #26]**使用TLS（thread，local，starage）》中介绍了的TLS并用的话，会将Libc的localtime函数替换为自动生成的安全函数，从而避免危险的发生。

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

struct tm *localtime(const time_t *timer) {
  static __thread struct tm tmbuf;
  return localtime_r(timer, &tmbuf);
}
```

准备这样的源代码并进行共享库化，然后插入到运行中的二进制分发软件中，在自动生成这种封装器时，可参考《[Hack #61]用LD_PRELOAD来lap既存函数》介绍的使用了RTLD_NEXT的Hack。

```
% gcc -D_REENTRANT -O2 -fPIC -shared -o safe_localtime.so -c
safe_localtime.c
% LD_PRELOAD=./safe_localtime.so /opt/path/to/
broken_proprietary_software
```

这样，在不变更二进制分发软件的条件下，可除去依赖于thing的不稳定的操作。

Windows的情况

用beginthreadex生成线程后，localtime等函数便可自动运行TLS，实在是太棒了。

总结

在多线程程序中一定不能使用的函数有89个，这些函数的使用，不仅仅是违反了标准的编码，还会引起依赖于timing的不稳定操作，成为棘手bug的原因。因而可以使用备用的安全代用函数。

另外，他人编写的函数读出了线程安全函数时，可通过运用了LD_PRELOAD和TLS的Hack来将危险函数替换为安全的函数。

参考文献

- "The Single UNIX Specification, Version 3, 2004 Edition" (*http://www.opengroup.org/onlinepubs/009695399/*) 可在线阅览的UNIX规格

—— *Yusuke Sato*

#51 安全编写信号处理的方法

任何情况下都不会错误运行的信号处理的编写至关重要，本Hack介绍了3个陷阱。

"用signal handler处理信号"可以说是UNIX编程的入门，但任何情况下都不会错误运作的信号处理的编写却非常困难，在Hack中介绍了容易出现的3个陷阱。

同步信号和异步信号

在UNIX规格中，将自反应的处理由于某种原因传到了自访问上的信号称为同步信号，例如下面的信号为同步信号。

- 执行与0的除法的SIGFPE
- 执行空指针引用时的SIGSEGV
- 调用abort函数而产生的SIGABRT
- 读出raise函数而产生的（任意的）single

另外，根据kill命令，kill系统调用，pthread_kill等函数将由其他进程或其他线程上传送过来的信号称为异步信号，在本Hack中所列举的是这里的异步信号。

陷阱1：若从single handler中读出不是非同期single safe的函数的话，则比较糟糕

比如如下，试着编写睡眠20秒之后运行结束的毫无意义的一个程序。重点在于分别从main函数和signal handler函数中读出了问题函数unsafe_func。

```
//
// async_signal_unsafe.c
//
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// 问题函数
void unsafe_func() {
  struct timeval tv = {20, 0};
  pthread_mutex_lock(&lock);
  select(0, NULL, NULL, NULL, &tv); // sleep 20sec.
  pthread_mutex_unlock(&lock);
}

// single handler
void handler(int signo) {
  unsafe_func();
  _exit(1);
}

int main() {
  // single handler的登录处理
  struct sigaction sa = {
    .sa_handler = handler,
    .sa_flags = 0
  };
```

```
    sigemptyset(&sa.sa_mask);
    sigaction(SIGHUP, &sa, NULL);

    //问题函数的调用
    unsafe_func();
    return 0;
}
```

运行这个程序，在开始运行10秒后，若将SIGHUP传送到进程上会发生什么呢？

```
% gcc -D_REENTRANT -o async_signal_unsafe async_signal_unsafe.c
-lpthread
% ./async_signal_unsafe &  (xxxx)
% killall -HUP async_signal_unsafe
```

用ltrace命令观察便可明白，这个程序发生了死锁，这是由于从main函数中调用unsafe_func函数后，在实行select的系统调用时（运行中）会收到信号，再从作为Single handler的handler函数中读出的unsafe_func函数中，对于已完成lock的mutex再次进行lock。

```
% ltrace ./async_signal_unsafe
...
pthread_mutex_lock(0x8049918, 0x485ff4, 0xbffffa88, 0x804864f,
1) = 0
select(0, 0, 0, 0, 0xbffff9c0 <unfinished ...>
--- SIGHUP (Hangup) ---
pthread_mutex_lock(0x8049918, 0xb7ff6440, 0x7ab9ab2, 0x4cb515,
0x35cfd4 (xxxx)
```

async-signal-safe 函数

将能从异步信号处理器安全的读出的函数称为“异步信号安全函数”。下面为规格上认为是异步信号安全函数的一览表。另外，下表中没有的函数的读出的结果视为未规定（只要使用的环境的man page和源代码没有特别记载时）。

从异步信号的信号处理器中读出非异步信号安全函数后，不仅仅是必须在规格上读出，还会发生棘手的问题，在先前的例子中，由于读出了表中没有的pthread_mutex_lock/unlock函数，所以发生了死锁。

_Exit()	fpathconf()	read()	sigset()
_exit()	fstat()	readlink()	sigsuspend()
abort()	fsync()	recv()	sockatmark()
accept()	ftruncate()	recvfrom()	socket()
access()	getegid()	recvmsg()	socketpair()
aio_error()	geteuid()	rename()	stat()
aio_return()	getgid()	rmdir()	symlink()

aio_suspend()	getgroups()	select()	sysconf()
alarm()	getpeername()	sem_post()	tcdrain()
bind()	getpgrp()	send()	tcflow()
cfgetispeed()	getpid()	sendmsg()	tcflush()
cfgetospeed()	getppid()	sendto()	tcgetattr()
cfsetispeed()	getsockname()	setgid()	tcgetpgrp()
cfsetospeed()	getsockopt()	setpgid()	tcsendbreak()
chdir()	getuid()	setsid()	tcsetattr()
chmod()	kill()	setsockopt()	tcsetpgrp()
chown()	link()	setuid()	time()
clock_gettime()	listen()	shutdown()	timer_getoverrun()
close()	lseek()	sigaction()	timer_gettime()
connect()	lstat()	sigaddset()	timer_settime()
creat()	mkdir()	sigdelset()	times()
dup()	mkfifo()	sigemptyset()	umask()
dup2()	open()	sigfillset()	uname()
execle()	pathconf()	sigismember()	unlink()
execve()	pause()	sleep()	utime()
fchmod()	pipe()	signal()	wait()
fchown()	poll()	sigpause()	waitpid()
fcntl()	posix_trace_event()	sigpending()	write()
fdatasync()	pselect()	sigprocmask()	
fork()	raise()	sigqueue()	

经常会想要调用pthread的函数，printf系的函数以及malloc函数，但要忍耐不调用为妙（注2）。

陷阱2：注意来自信号处理器的非volatile变量的操作

在运行从异步信号的信号处理器中而来的全局变量时，必须用volatile修饰该变量，否则运行该变量的结果记为未定义。来看看有实际危害的简单例子。

```
//
// non_volatile.c
//
static int sig = 0;
void handler(int signo) {
  sig = 1;
}

int main() {
  // 省略Signal handler的登录处理
```

注2：这只是面向对异步信号的处理器没有了解的读者作出的介绍，希望引起他们的注意。存在着由于glibl的构造没有问题，而有计划地从信号处理器中调用非信号处理函数的情况，另外“用信号处理器挑战安全运行处理的极限”也是Binary Hack的一种乐趣，[Hack #53]的运用了信号安全的Hack便是其中的一种。

```
    while(sig == 0);
    return 0;
}
```

登录到信号处理器上后，无限循环等待SIGHUP的到达，虽然是到达之后便终止的程序，但将其在笔者的环境中进行优化并编译后则不能正常运作，即便是使用kill -HUP进程也不终止。用gcc -O2 -S来浏览汇编语言的目录便可明白其原因。

```
movl sig, %ebx              将sig的值复制到EBX寄存器上
.L8:
testl      %ebx, %ebx       # EBX若为0 则向.L8
jne        .L8              #
```

从内存上将变量sig的值一次复制到寄存器上，然后只监视存储器的值并循环，这样便不能检测出在信号处理器上的变量sig的更新，用volatile来修饰变量sig的话，则可解决该问题。

陷阱3：注意来自信号处理器中的非sig_atomic_t型的变量的操作.

在x86中使用了uint64_t等的64位变量后，便会对变量的代入处理分割成2条机器语句。或者反过来，32/64bit变数的操作即便是原子的，也存在将代入到8/16位变量的多个命令作为必要的CPU。在这些情况中，如果在代入的途中转移到信号处理器上，处理器的变量的值则会变为预想外的值。因此在处理器中操作的全局变量是用一个机器语言命令。就可以处理的类型的话比较安全。为了给予保证，在signal.h中使用被typedef过的类型sig_atomic_t。

可代入到sig_atomic_t型的值的范围可用SIG_ATOMIC_MIN SIG_ATOMIC_MAX来检测。另外，想要编写可移植的代码时，面向变量sig_atomic_t型的代入可选择0～127的值。

总结

在想要编写出不发生死锁、资源泄露、崩溃等问题的安全代码，以及移植性强的代码时，要牢记以下规则。

- 在异步信号的信号处理器中，只调用异步信号安全的函数
- 在异步信号的信号处理器中，不操作volatile sig_atomic_t类型以外的全局变量

认为“在这种严格条件下，不能编写信号处理器”的你可以试试“[Hack #52]用 sigwait 对异步信号进行同步处理” 和“[Hack #53]用 sigsafe 将信号处理安全化 ”中介绍了的 Hack ，便可以非常简单地处理异步信号。另外，关于同步信号的处理器并没有这样的限制。

参考文献

- “The Single UNIX Specification, Version 3, 2004 Edition, 2.4 Signal Concepts? ”（*http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html*）

—— *Yusuke Sato*

用 sigwait 将异步信号进行同步处理

在本 Hack 中，介绍了使用 sigwait，“不使用信号处理器来处理异步信号”的方法。

“[Hack #51]安全编写信号处理器的方法”这个 Hack 中介绍的是安全编写异步信号的信号处理器的方法，难度比较大。在本 Hack 中，介绍了简单运行安全的信号处理方法。用 sigwait. 这个 UNIX 的规格标准化过的函数的使用方法与“[Hack #53]用 sigsafe. 处理安全化”的方法有所不同。

idea

异步信号的处理器论述上有很多制约，是因为以下两点不能预测的问题。

- 何时在什么样的时间点来发送信号。
- 还有其结果将在何时什么时间点，信息处理会与信号处理器岔开？

这样，不使用信号处理器，如果像下面这样来处理信号会怎样呢？在这里再次以 SIGHUP 为例。

1. 完全不使用 sigaction 的函数
2. 用所有的线程来遮蔽 SIGHUP
3. 仅执行 SIGHUP 的处理，生成专用的线程
4. 该线程监视信号的到达（只在检测到达的时刻解除 SIGHUP 的遮蔽）

这样一来，由于只能在运行特定线程的特定位置的代码时才能将SIGHUP发送到进程上，便不存在异步信号的信号处理器论述中有的那些制约。

sigwait函数

有好几种像上述4中那样监视信号到达的方法，使用sigwait函数的方法是最好的，虽然pause函数和sigsuspend函数也可以进行相似的论述，但很难安全地使用它们（注3）。sigwait函数，根据附属在glibc上的man page有如下功能。

格式

```
int sigwait(const sigset_t *set, int *sig);
```

说明

sigwait会在把由set指定的signal中的1个传送到读出线程之前停止读出线程的运行，然后将收到信号编号存放到用sig指定的区域中。

在仅用这些说明不能进行ping时，就可以看看POSIX的规格。在一些Linux中，通过

```
% man 3p sigwait
```

可阅览POSIX中的sigwait函数的定义。这个选项3P的说明书非常详细，值得参考。

signal处理线程的安装实例

实际编写一下代码。首先，将处理SIGHUP的线程的入口变为如下这样。

```
void *wait_for_sighup(void *dmy) {
  int sig;

  // sigwait 函数等待 SIGHUP
  sigset_t ss;
  sigemptyset(&ss);
  sigaddset(&ss, SIGHUP);
  while(1) {
    // 等待 SIGHUP 的到达
    if (sigwait(&ss, &sig) == 0) {
      // << signal handler 适当处理 >>
      printf("Hello async-signal-safe world!\n");
    }
  }
```

注3：会将“接收到信号的数”误认为“接收到信号的号码”。

```
  return NULL;
}
```

在写着 << signal handler 适当处理 >> 的行上执行非异步也完全没有问题。这是本 Hack 最大的优点。和异步信号处理器不同，无论是 printf 函数，还是 malloc 函数，还是 pthread_cond_signal 函数都可以自由调用。虽然要注意其线程安全性，但执行非 volatile sig_atomic_t 型变量的操作也没有问题。

main 函数的安装实例

生成该线程处理如下所示。包括 wait_for_sighup 函数，为了将所有线程中的 SIGHUP 被遮蔽（mask），从而在 main 函数的开始处读出 sigprocmask 函数。

```
int main() {
  sigset_t ss;
  pthread_t pt;
  pthread_attr_t atr;

  // SIGHUP mask.(发生了 SIGHUP 也选择保留状态)
  sigemptyset(&ss);
  sigaddset(&ss, SIGHUP);
  sigprocmask(SIG_BLOCK, &ss, NULL);

  // 生成处理 SIGHUP 专用的线程
  pthread_attr_init(&atr);
  pthread_attr_setdetachstate(&atr, PTHREAD_CREATE_DETACHED);
  pthread_create(&pt, &atr, wait_for_sighup, NULL);

  // 之后执行一般的处理

  return 0;
}
```

编译后将会像下面这样安全正常地运行。

```
% gcc -D_REENTRANT -O2 -o sigwait sigwait.c -lpthread
% ./sigwait &  (在 back ground 上运行)
[2] 1337
% kill -HUP $!  (将 SIGHUP 发送到这个进程上)
Hello async-signal-safe world!
%
```

sigwait 函数的同类

在 sigwait 函数中，有 2 个 variant。

```
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

```
int sigtimedwait(const sigset_t *set, siginfo_t *info, const
struct timespec timeout);
```

详细了解sigaction函数的人很容易明白，sigwaitinfo函数为了准确知道发送了怎样的信号而扩展得到了构造体siginfo_t。另外sigtimedwait函数还附带着指定信号的等待到达时间的功能。任意函数都与sigwait函数不同，通过 errno 来提示错误。

参考文献

《UNIX 编程详解》（W · 理查德 · 史蒂芬著，大木敦雄译）。

总结

在本Hack中，通过运用sigwait函数介绍了“不使用信号处理器来处理异步信号”的方法。该方法可自由使用不是异步信号安全的函数来论述将信号发送到进程上时的处理。因此，以安全而多样的方法来处理信号的方法非常简单。

一般而言，检测以及调试并用了符号和线程的程序是非常困难的，但这个sigwait的例子却比较特别。不会引起棘手的问题，可安全使用。

——*Yusuke Sato*

HACK #53 用 sigsafe 将信号处理安全化

在本 Hack 中介绍了使用 sigsafe 库来简单安全的处理信号的方法。

sigsafe 的安装

sigsafe 在（*http://www.slamb.org/projects/sigsafe/*）中有提供。这个库不是用make而是用scons这个工具来记录构建进程。例如会像如下这样运行构建和安装：

```
% tar xfz sigsafe-0.1.3.tar.gz
% cd sigsafe-0.1.3
% vi SConstruct        # 将文件开头的debug = 1变为debug = 0。
% scons
...
% sudo cp src/sigsafe.h /usr/local/include
% sudo cp build-i386-linux-st/libsigsafe.a /usr/local/lib
```

这里将SConstruct变为debug = 0是为了在接到信号时不输出调试信息。

信号的难度

来看以下的程序。

- 从标准输入复制到标准输出（必须要输出读出的数据）。
- 接到 SIGINT 后，在标准输出上表示出复制的字节数并终止。
- 接到 SIGINT 后，这之后在标准输入的读取中不可以阻塞。

下面介绍该程序的组成，在这里，由于必须输出已读出的数据，所以在写入已读出的数据时要尽量不用 SIGINT 来遮罩。但在用 read 来锁定时有必要获得 SIGINT，所以 read 本身在 SIGINT 未被遮蔽的状态下读出。

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

#define BUFFERSIZE 4096
volatile long counter = 0;

void handler(int signum)
{
  /* 将在 signal handler 做什么 */
}

int main(int argc, char **argv)
{
  ssize_t ret;
  size_t wsize;
  char buf[BUFFERSIZE], *p;
  struct sigaction act;
  sigset_t defaultmask, intmask;

  sigemptyset(&intmask);
  sigaddset(&intmask, SIGINT);
  sigprocmask(SIG_SETMASK, &intmask, &defaultmask);

  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, NULL);

  counter = 0;
  for (;;) {
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    /* 若符号在即将 read 前到达的话将如何呢？ */
    ret = read(0, buf, BUFFERSIZE);
    /* 若符号在刚 read 之后到达的话将如何？ */
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == 0)
      return 0;
```

```
    if (ret == -1) { perror("read"); exit(1); }
    counter += ret;
    wsize = ret;
    p = buf;
    while (wsize) {
      ret = write(1, p, wsize);
      if (ret == -1) { perror("write"); exit(1); }
      wsize -= ret;
      p += ret;
    }
  }
}
```

在这里问题是SIGINT的信号处理器上将运行什么，以及若在即将read前，read后SIGINT到达的话又会怎样？

用信号处理器输出字节数并退出

首先要考虑的是从信号处理器中输出counter并终止的问题，即像下面这样定义信号处理器。

```
void handler(int signum)
{
  printf("%ld\n", counter);
  exit(0);
}
```

但这样一来，在read返回后，设定信号处理器之前获得SIGINT是会发生问题，此时，若不在read中写出已读出的数据的话，则无法知道其长度。

用信号处理器来设置标志

那么，在信号处理器中要如何只将标志设置到全局变量上呢？在主循环上检测该标志，如果被设置了，输出计数器的值并终止。另外，在read中锁定时获得信号的情况中，read失败后则变为EINTR。但一般可忽视这错误。

```
...
volatile int signal_caught = 0;
void handler(int signum)
{
  signal_caught = 1;
}
...
      return 0;
    if (ret == -1 && errno == EINTR) ret = 0;
    if (ret == -1) { perror("read"); exit(1); }
...
    }
    if (signal_caught) {
```

```
        printf("%ld\n", counter);
        exit(0);
      }
    }
  ...
```

但这样一来，在 signal（信号）在即将 read 之前到达时便会产生问题。由于只在信号处理器设置标志，所以在从信号处理器返回后读出 read 系统调用，可能会将 read 阻塞。这样的话在该阻塞完成之前不能终止处理。在这里将标准输入作为终端，在输入之前不能终止。

用 siglongjmp 跳出信号处理器

一般从信号处理器中跳出可能会导致阻塞，若用 siglongjmp 强制跳出信号处理的话，会怎么样呢？这样的话，即便信号在 read 系统调用之前到达，也可不阻塞就搞定。

```
...
sigjmp_buf env;
void handler(int signum)
{
  siglongjmp(env, 1);
}
...
  sigset_t defaultmask, intmask;
  if (sigsetjmp(env, 1) != 0) {
    printf("%ld\n", counter);
    exit(0);
  }
  sigemptyset(&intmask);
...
```

但是，若在刚 read 之后，信号在设定 signal 遮罩之前到达的话会怎样呢？此时将会丢失 read 的返回值。read 的返回值被代入到 ret 上，但在发生代入前可能会取得信号，此时，若用 siglongjmp 脱离后，由于不发生代入，故返回值不会被记录在任何地方而消失，若不能得到 read 的返回值，则不能计出字节数。当然，也不能读出数据，在这一点和从信号处理中退出相同。

在 select 附近用 siglongjmp 从信号处理中跳出

在 read 之前进行 select，然后在 select 的附近接到信号，用 siglongjmp 从信号处理中跳出的话会怎样呢？select 将检测是否可输出或输入，这个由于不影响外部所以丢失结果也无所谓。因此用 siglongjmp 即使脱离也没关系，在明白用 select 可输入时即使读出 read 也不会阻塞，在 read 附近即便不接到信号也没有问题。

```
...
sigjmp_buf env;
void handler(int signum)
{
siglongjmp(env, 1);
}
...
  sigset_t defaultmask, intmask;
  if (sigsetjmp(env, 1) != 0) {
    printf("%ld\n", counter);
    exit(0);
  }
  sigemptyset(&intmask);
...
  for (;;) {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(0, &readfds);
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    ret = select(1, &readfds, NULL, NULL, NULL);
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == -1) { perror("select"); exit(1); }
    ret = read(0, buf, BUFFERSIZE);
    if (ret == 0)
...
```

但使用了select后会发生这样的问题，若其他程序在select和read之间读出数据的话则会阻塞。在标准输入与终端相关时，由于很多程序中不能共用终端，所以也是有可能发生的。

另外，从select附近启动的信号处理中脱离的方法除了sigsetjmp之外还有the pipe trick，这可以在事前生成管道并用select监视读出的那一方。在信号处理中书写管道的一方中编写出1字节。这样一来，即使是从信号处理器返回后再启动select，由于可以从其管道中读出，故可以立即终止select而不发生阻塞。该方法不能解决其他程序在select和read间阅读出数据的问题。

使用非阻塞I/O

也许会有人认为即使是使用了非阻塞I/O也不一定好。确实，read用了非阻塞I/O的话不会发生阻塞但会产生EAGAIN错误，此时从select开始重来，搭配siglongjmp则能符合其规格。

```
sigjmp_buf env;
void handler(int signum)
{
  siglongjmp(env, 1);
```

```
}
...
  sigset_t defaultmask, intmask;
  if (sigsetjmp(env, 1) != 0) {
    printf("%ld\n", counter);
    exit(0);
  }
  sigemptyset(&intmask);
...
  for (;;) {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(0, &readfds);
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    ret = select(1, &readfds, NULL, NULL, NULL);
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == -1) { perror("select"); exit(1); }
    ret = fcntl(0, F_GETFL);
    if (ret == -1) { perror("F_GETFL"); exit(1); }
    ret = fcntl(0, F_SETFL, ret|O_NONBLOCK);
    if (ret == -1) { perror("F_SETFL"); exit(1); }
    ret = read(0, buf, BUFFERSIZE);
    if (ret == 0)
      return 0;
    if (ret == -1 && errno == EAGAIN) continue;
    if (ret == -1) { perror("read"); exit(1); }
...
```

但若将标准输入当作终端的话，将其设定为非阻塞I/O就比较难。终端能被许多程序共享，将其设定为非阻塞I/O后，则该设定也可被其他程序共享。

实际上，非阻塞I/O会对其他程序产生不好的影响。例如：有名的stdio库与非阻塞I/O的相融性较差，搭配后有时数据会消失。(Richard Stevens说将stdio通过非阻塞来使用的方法描述为“破灭的处方”)。非阻塞I/O的设定可在多个程序中共享，该程序在设定非阻塞I/O之后在读出read时，其他的程序也许会回到阻塞中。Read时究竟有没有阻塞也不知道，在即将read前接到信号，信号处理设置了标志之后，SIGINT到达了则会与标准输入中读出的阻塞的第二个方案相同。

sigsafe

signal的难度，在于进行阻塞的系统调用之前或刚结束的处理中。虽然信号在系统调用之前到达的情况，用sigsetjmp脱离是一个不错的方法，但若在后面到达时这样操作的话就会丢失系统调用的返回值。在系统调用之后到达时，用return来脱离也很好，但若在其之前到达时这样操作的话则会在系统调用中阻塞。

另外，信号很有可能在系统调用正在阻塞时到达。内核在这种情况下将处理权限转到用户级，并启动信号处理，所以无论是在系统调用的前后到达都是相同的。由系统调用是否再启动来决定和前后的哪种情况等价，再启动时则和之前到达的等价，不进行再启动时则和之后到达的等价。

因此，从信号处理中区别了是系统调用前还是系统调用后的话便可以解决问题了。若是前，则siglongjmp，若是后则直接返回就可以了。但遗憾的是没有做到这一点的简便方法。

但是，存在不很简便却能实现的方法，sigsafe便可实现。它会从信号处理中检测返回地址附近的代码，并确认是执行系统调用分配的命令之前返回还是在之后返回。该处理会因每个进程的不同而不同，也会因OS的不同而不同，当前的sigsafe会支持以下的环境。

- Darwin/ppc (a.k.a OS X)
- FreeBSD/i386
- Linux/alpha
- Linux/i386
- Linux/ia64
- Linux/x86_64
- NetBSD/i386
- Solaris/sparc
- Tru64/alpha

sigsafe安装

使用了sigsafe后，便可像以下这样安装例中的程序。在这里，将sigsafe read函数代替read来使用。该函数在运行中以及运行前接到信号时，会返回到-EINTR。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sigsafe.h>

#define BUFFERSIZE 4096

long counter = 0;
```

```
void handler(int signum, siginfo_t *si, ucontext_t *ctx, intptr_t user_data)
{
  /* 在想用 SIGINT 不让进程立马结束的情况中，信号处理虽然很重要，
     但它不做任何操作 */
}
int main(int argc, char **argv)
{
  ssize_t ret;
  size_t wsize;
  char buf[BUFFERSIZE], *p;
  sigset_t defaultmask, intmask;

  sigemptyset(&intmask);
  sigaddset(&intmask, SIGINT);
  sigprocmask(SIG_SETMASK, &intmask, &defaultmask);

  sigsafe_install_handler(SIGINT, handler);
  sigsafe_install_tsd(0, NULL);

  counter = 0;
  for (;;) {
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    ret = sigsafe_read(0, buf, BUFFERSIZE);
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == -EINTR) {
      printf("%d\n", counter);
      exit(0);
    }
    if (ret == 0)
      return 0;
    if (ret == -1) { perror("read"); exit(1); }
    counter += ret;
    wsize = ret;
    p = buf;
    while (wsize) {
      ret = write(1, p, wsize);
      if (ret == -1) { perror("write"); exit(1); }
      wsize -= ret;
      p += ret;
    }
  }
}
```

像如下这样来编译：

```
% gcc -I/usr/local/include test.c -L/usr/local/lib -lsigsafe
```

sigsafe_read在函数启动前接到信号，此时不调用read系统调用，并立即返回-EINTR。为了实现此操作，将记录sigsafe接到了哪个信号（该记录可用sigsafe_clear_received来清除）。在这个程序中sigsafe_read在起动之前接到SIGINT的情况下，sigsafe_read会立即返回_EINTR，不阻塞并表示出字节数就结束。另外，在sigsafe_read结束后

接到SIGINT的情况下，sigsafe_read的返回值即为实际read的返回值，故根据该结果可写出已读取的数据。但sigsafe为了记录接到的SIGINT，循环之后的sigsafe_read的反复读出会返回-EINTR并终止。其结果表示为当时的字节数然后终止。

也就是说，无论是在sigsafe_read之前或是之后接到信号，都能正常运行处理。

总结

在进行阻塞的系统调用之前或之后接到信号时，不推迟信号的话处理比较困难，但由于使用sigsafe，能安全地处理。

——*Akira Tanaka*

用Valgrind检测出内存泄漏

Valgrind是与Linux/x86、Linux/amd64、Linux/ppc32兼容的用于动态解析程序运行的工具。

Valgrind（*http://valgrind.org/*）是通过在虚拟的CPU上运行程序来动态解析程序运作的工具，使用了Valgrind便可像下面那样高效地发现bug：

- 内存泄漏、内存的不正常释放、二次释放。
- 各种不正确的内存访问。
- 多线程程序中的内存访问冲突。

安装

Valgrind在Linux/x86、Linux/amd64、Linux/ppc32上操作。到目前为止最新版为v3.1.0。遗憾的是，就执行CPU模拟实验这个属性而言，其他的OS和进程（官方的）不会支持。

Debian GNU/Linux、FedoraCore、CentOS等的distribution中有标准的Valgrind版本（v2.x系列），想要使用最新版3.1.0时，要从源代码编译并安装。一般地，只要运行/configure && make && sudo make install便OK了。

使用方法

Valgrind 的使用方法很简单

```
% valgrind --leak-check=full --leak-resolution=high --show-reachable=yes
<target_program>
```

像以上这样，在 Valgrind 上运行 <target_program>，将会输出 bug 报告，试着编写代码。

```
//
// valgrind_test1.cpp
//
#include <cstdlib>
int* leaky_foo(void) {
  int *a = new int;
  int *b = new int; // (bug 1)内存泄露return a;
}

static int *c;
int main(int argc, char **argv) {
  c = leaky_foo(); // (bug2) 内存泄露(在程序终止之前不删除c )
  char *d = (char *)std::malloc(sizeof(char) * 10U);
  delete[] d; // (bug3) 删除已malloc过的东西。(free正确)
}
```

在编译调试对象的程序时，要使用 -g 以及 -ggdb 选项，将优化最大到 -O1。另外，要避免清除链接，这是因为 valgrind 不能改写 malloc 等函数。

```
% g++ -g -O0 -o valgrind_test1 valgrind_test1.cpp
% valgrind --leak-check=full --leak-resolution=high --show-
reachable=yes ./
valgrind_test1
```

将会有如下的报告：

```
==31927== Mismatched free() / delete / delete []
==31927== at 0x400550E: operator delete[](void*) (vg_replace_malloc.c:256)
==31927== by 0x80486A3: main (valgrind_test1.cpp:16)
==31927== Address 0x4012098 is 0 bytes inside a block of size 10 alloc'd
==31927== at 0x400446D: malloc (vg_replace_malloc.c:149)
==31927== by 0x804868C: main (valgrind_test1.cpp:15)

==31927== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==31927== at 0x40047F8: operator new(unsigned) (vg_replace_malloc.c:164)
==31927== by 0x80485EB: leaky_foo() (valgrind_test1.cpp:8)
==31927== by 0x8048639: main (valgrind_test1.cpp:13)

==31927== 4 bytes in 1 blocks are still reachable in loss record 2 of 2
==31927== at 0x40047F8: operator new(unsigned) (vg_replace_malloc.c:164)
==31927== by 0x80485DB: leaky_foo() (valgrind_test1.cpp:7)
==31927== by 0x8048639: main (valgrind_test1.cpp:13)
```

完美地检测出了所有的问题（'31927' 为进程 ID）

命令行选项（command line option）

Valgrind 的命令行选项有很多，常用的有以下 6 个。

- 指定 --show-reachable=yes 后，会报告（bug2）那样的可能无害的内存泄漏。
- 指定 --trace-children=yes 后，解析中的程序将用 fork 生成的子进程也变为解析对象。
- 指定 --track-fds=yes 后，会出现忘记关闭文件描述符的报告。
- 指定 --error-limit=no 后，即便是错误值超过了极限值之后 Valgrind 也会继续进行解析，在调试初期该选项十分有用。
- 可用 --num-callers=<number> 来指定到错误位置为止表示出多少回溯（默认为 12），在调试 C++ 程序时可能需要要指明。
- 可用 --xml=yes 将输出变为 XML 形式，在想要自动处理输出时非常方便。

总结

Valgrind 是与 Linux/x86、Linux/amd64、Linux/ppc32 相符的动态解析程序运作的工具。使用 Valgrind 可高效地发现“内存泄露”、“不正确的内存释放”、“二次释放”等麻烦的 bug。

—— *Yusuke Sato*

使用 Valgrind 检测出错误的内存访问

使用了 Valgrind 后，不仅可以检测出内存泄漏，还可以检测出各式各样错误的内存访问。

用了“**[Hack #54]**用 Valgrind 检测出内存泄漏”介绍了的工具 "Valgrind" 之后，不仅可以检测出内存泄漏，还可以检测出各式各样错误的内存访问。在本 Hack 中首先介绍：

- 未初始化变量的使用
- 范围外的内存访问

- 访问已释放的内存
- 复制的源地址和目的地址的重叠

介绍了 Valgrind 检测出以上这些的样子，最后介绍 Valgrind 不能检测出的 bug 种类。

错误的程序的例子

试着编写下面的程序。将这个程序进行一般的运行，在很多环境中没有什么问题，main 函数也会终止，但实际上会执行很多比较复杂的处理。在 Valgrind 上运行这个程序会出现怎样的结果呢？

```
//
// horrible.c
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static char onbss[128];
int main() {
  char onstack[128];
  int uninitialized, dummy;
  char *onheap = (char*)malloc(128);

  dummy = onbss[128];
  dummy = onstack[150];

  if (uninitialized == 0) {
    printf("hello world!\n");
  }
  close(uninitialized);

  dummy = onheap[128];

  free(onheap);
  dummy = onheap[0];

  strcpy(onstack, "build one to throw away; you will anyway.");
  strcpy(onstack, onstack + 1);

  return 0;
}
```

Valgrind 上的运行

在 Valgrind 上运行 horrible 命令。在不进行内存泄漏的检测时，没有命令行也可以运行 Valgrind。

```
% gcc -g -o horrible horrible.c
% valgrind ./horrible
```

这样一来，便可以检测出5个错误。来看看这些错误。

未初始化变量的使用

将保存在栈上的变量uninitialized在没有初始化的情况下使用。

```
if (uninitialized == 0) {    // 使用位置1
  printf("hello world!\n");
}
close(uninitialized);        // 使用位置2
```

Valgrind会检测出这个问题，表示为以下的2个错误。

```
==24754== Conditional jump or move depends on uninitialised value(s)
==24754==    at 0x804848F: main (horrible.c:16)

==24754== Syscall param close(fd) contains uninitialised byte(s)
==24754==    at 0x4187BC: __close_nocancel (in /lib/tls/libc-2.3.4.so)
==24754==    by 0x374E22: __libc_start_main (in /lib/tls/libc-2.3.4.so)
```

范围外的内存访问

本来堆上只保存了128字节，却引用了第129字节。

```
dummy = onheap[128];
```

将表示出下面的错误。

```
==24754== Invalid read of size 1
==24754==    at 0x80484BB: main (horrible.c:21)
==24754== Address 0x40120A8 is 0 bytes after a block of size 128
alloc'd
==24754==    at 0x400446D: malloc (vg_replace_malloc.c:149)
==24754==    by 0x8048467: main (horrible.c:11)
```

访问已释放的内存

用free释放内存后引用这块内存。

```
free(onheap);
dummy = onheap[0];
```

表示出下面的错误

```
==24754== Invalid read of size 1
==24754==    at 0x80484DB: main (horrible.c:24)
==24754== Address 0x4012028 is 0 bytes inside a block of size 128 free'd
==24754==    at 0x4004F62: free (vg_replace_malloc.c:235)
==24754==    by 0x80484D1: main (horrible.c:22)
```

复制的源地址和目的地址的重叠

strcpy、memcpy等的函数，若将复制的源地址和目的地址重叠会引起错误运行。

```
strcpy(onstack, "build one to throw away; you will anyway.");
strcpy(onstack, onstack + 1);                    // 重叠过的复制
```

表示出下面的错误。

```
==24754== Source and destination overlap in strcpy(0xBEEBB980,
0xBEEBB981)
==24754==    at 0x4006047: strcpy (mac_replace_strmem.c:269)
==24754==    by 0x8048511: main (horrible.c:26)
```

用 Valgrind 不能检测的错误

Valgrind不能检测保存在栈上的内存和data/bss区域的内存的错误访问（参照*http://valgrind.org/docs/manual/faq.html#faq.overruns*）。因此，以下的两个错误会被忽视。

```
dummy = onbss[128]; // 范围外的访问
dummy = onstack[150]; // 范围外的访问
```

为了检测出这些错误，必须要使用“**[Hack #44]**用 Mudflap 检测出缓冲区溢出”中介绍的 Hack。

组成

valgrind命令，会在开始运行后立即生成虚拟CPU。由于这个虚拟CPU将运行所有要解析的对象程序，故Valgrind可完全控制程序的运行。不用root权限也可以使用Valgrind，因为不用改写解析对象程序和再连接，root权限也没有必要。

为了使程序在虚拟CPU上高速运作，要在Valgrind内部使用x86-to-x86的JIT技术。另外，解析对象程序在调用系统调用的瞬间，会将虚拟CPU的广泛使用的寄存器和栈指针的内容复制到真正的CPU上，然后进行软件中断。这些操作十分有趣。

详细的组成，在Valgrind官方网页的《Valgrind Technical Documentation》（*http://valgrind.org/docs/manual/mc-tech-docs.html*）有说明。另外，关于系统调用的组成在“**[Hack #59]**怎样调用系统调用”中有介绍，请一同参照。

总结

使用了Valgrind后，不仅仅是内存溢出，“范围外的内存访问”、“访问已释放内存”这些与内存、指针相关的bug都可以被检测出。

这在提高程序安全性以及加强安全防范上非常有用。

—— *Yusuke Sato*


用Helgrind检测出多线程程序的bug

Valgrind可以根据被称为tool的插件扩展。

Valgrind根据tool（一种插件那样的）组成，可以进行多种多样的扩展。用标准形式准备下面的tool。

Memcheck	检测内存泄漏和错误的内存访问（默认）
Cachegrind	测定CPU的L1/L2cash的命中率
Massif	测定堆的使用情况
Helgrind	检测出多线程程序的bug

自动生成全新的tool是不可能的。实际上，执行比Memcheck更严密的内存访问的检测的工具（Annelid）和检测是否在信号处理中进行了复杂的处理的工具（Crocus）等实验的tool公开在了（*http:// valgrind.org/downloads/variants.html*）中。

尝试Helgrind

在这里，试着用Helgrind来检测多线程程序的bug。试着编写test程序。

```
// 省略了错误检测

// test1:没有lock的变量访问
static int count = 1;
void *incr_count(void *p) {
  ++count;
  return 0;
}

// test 2: 没有被统一的lock顺序
static pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
void *lock_m1_then_m2(void *p) {
  pthread_mutex_lock(&m1); pthread_mutex_lock(&m2);
```

```
  pthread_mutex_unlock(&m2); pthread_mutex_unlock(&m1);
  return 0;
}
void *lock_m2_then_m1(void *p) {
  pthread_mutex_lock(&m2); pthread_mutex_lock(&m1);
  pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
  return 0;
}

int main() {
  pthread_t t1, t2, t3, t4;
  pthread_create(&t1, NULL, incr_count, NULL);
  pthread_create(&t2, NULL, incr_count, NULL);
  pthread_create(&t3, NULL, lock_m1_then_m2, NULL);
  pthread_create(&t4, NULL, lock_m2_then_m1, NULL);
  pthread_join(t4, NULL); pthread_join(t3, NULL);
  pthread_join(t2, NULL); pthread_join(t1, NULL);
  return count;
}
```

这个程序中至少潜伏着 2 个问题。

问题 1：没有 lock 的变量操作

不进行 lock，来更新变量 count。

```
void *incr_count(void *p) {
  ++count;
```

这个 C 语言代码的第 2 行，用汇编语言分解为以下的多个命令。

```
movl   count, %eax
incl   %eax
movl   %eax, count
```

多个线程在同时运行 incr_count 函数时，变量 count 会不正常的增加。

问题 2：lock 的顺序不统一

将 Mutex 的 m1、m2 像以下这样 lock。

- 在 lock_m1_then_m2 中按 m1 → m2 的顺序 lock
- 在 lock_m2_then_m1 中按 m2 → m1 的顺序 lock

lock 的顺序不统一。在 lock 某个 Mutex 时，进行 lock，在程序的整个区域中统一其 lock 的顺序是必须的。忽视了它（省略了详细介绍）会发生程序死锁的情况。

使用方法

试着在Helgrind上运行上面的程序。除了用命令行选项指定-tool=helgrind之外，其他的使用方法和Memcheck相同。

```
% gcc -ggdb -o data_race data_race.c -lpthread
% valgrind --tool=helgrind ./data_race
```

没有lock的变量操作会像以下那样被检测出。

```
==4278== Possible data race writing variable at 0x80497D8 (count)
==4278== at 0x804847F: incr_count (data_race.c:6)
==4278==   by 0xB000F14F: do__quit (vg_scheduler.c:1872)
==4278== Address 0x80497D8 is in data section of /tmp/data_race
==4278== Previous state: shared RO, no locks
```

lock顺序不统一，也能像以下那样被检测出。

```
==4278== Mutex 0x80497E0(m1) locked in inconsistent order
==4278==    at 0x1D4B0AA3: pthread_mutex_lock
(vg_libpthread.c:1324)
==4278==    by 0x80484FB: lock_m2_then_m1 (data_race.c:22)
==4278==    by 0x1D4AF8D1: thread_wrapper (vg_libpthread.c:867)
==4278==    by 0xB000F14F: do__quit (vg_scheduler.c:1872)
==4278== while holding locks 0x80497F8(m2)
==4278== 0x80497F8(m2) last locked at
==4278==    at 0x1D4B0AA3: pthread_mutex_lock
(vg_libpthread.c:1324)
==4278==    by 0x80484EB: lock_m2_then_m1 (data_race.c:21)
==4278==    by 0x1D4AF8D1: thread_wrapper (vg_libpthread.c:867)
==4278==    by 0xB000F14F: do__quit (vg_scheduler.c:1872)
==4278== while depending on locks 0x80497E0(m1)
```

完成。

注意

在写这个Hack时使用了Valgrind-2.2.0。遗憾的是，在Valgrind-2.4.0以后的版本中不能使用Helgrind。Helgrind不能随着Valgrind本身的进化而进化，但官方决定将在下个版本支持。

Massif和Cachegrind在3.1.0中可以使用。请务必使用。

总结

"**[Hack #54]**用Valgrind检测内存溢出"的Hack介绍了Valgrind根据被称为tool的插件的组成可进行多样的扩展。这期间用标准形式包含在Valgrind

的发布套件中的Helgrind这个工具非常有用，可以检测出以下这样的多线程程序的状态。

- 多线程在没有 lock 同一变量时操作的固定场所。
- 互斥量的 lock 顺序的不统一。

这些可以有效地发现很难再现的错误运行以及死锁等棘手的 bug。

—— *Yusuke Sato*

用 fakeroot 在相似的 root 权限中运行进程

在本 Hack 中将介绍与 root 有类似权限的 fakeroot 实现进程运行的环境。

fakeroot

在编译软件后，在当前的环境中不管要进行 `make install`，还是像 `DESTDIR=$(pwd)/debian/tmp` 那样，把正在编译的目录当作以下目录 `make install`，然后安装文件。通过生成将其目录选择到 `$(pwd)/debian/tmp` /上的文件存档来打包（实际上包中包含了文件存档的其他控制信息和主要脚本等）。

此时，想要将存档文件的所有者变为 root，在 root 权限中运行。但是为了生成包将 root 权限视为必要，在安全性来看并不是不好。

因此开发了 fakeroot。在 fakeroot 环境内运行，在其中可像 root 操作文件那样看到其进程。但在实际的文件系统上可用原来的 user 权限来操作文件。

```
% ls -l
合计 0
-rw-r--r-- 1 ukai ukai 0 2006-02-18 01:04 foo
% fakeroot
# id
uid=0(root) gid=0(root)所属组 =40(src),1000(ukai)
# ls -l
合计 0
-rw-r--r-- 1 root root 0 2006-02-18 01:04 foo
# touch bar
合计 0
-rw-r--r-- 1 root root 0 2006-02-18 01:05 bar
-rw-r--r-- 1 root root 0 2006-02-18 01:04 foo
# chown www-data:www-data foo
```

```
# ls -l
合计0
-rw-r--r-- 1 root root 0 2006-02-18 01:05 bar
-rw-r--r-- 1 www-data www-data 0 2006-02-18 01:04 foo
# exit
% ls -l
合计 0
-rw-r--r-- 1 ukai ukai 0 2006-02-18 01:05 bar
-rw-r--r-- 1 ukai ukai 0 2006-02-18 01:04 foo
```

这样运行fakeroot后将启动fakeroot环境的命令行。其中，可以发现相似的root在运行。运行了id(1)后将被报告为root，文件的所有者也变为root。新建文件的所有者也会变成root，现存的文件会变成为其他的所有者。

但实际上这些并不会变更，只不过在fakeroot环境中才会这样。会明白用exit脱离出fakeroot环境后实际文件会如何变化。

在Debian中，在生成包时root权限是必要的，这样的操作也是有的。也可指定为了得到root权限的命令，使用了sudo等可得到真正的root权限并运行。一般用fakeroot来得到相似的root权限并运行。

fakeroot的组成

fakeroot是单纯的shell脚本，以下为运用fakeroot命令进行的基本处理。

- 启动faked，得到FAKEROOTKEY。
- 设定FAKEROOTKEY、LD_LIBRARY_PATH、LD_PRELOAD，启动命令（默认为shell）。

faked是为管理文件的inode和与之相应的相似的owner等信息的Demon。

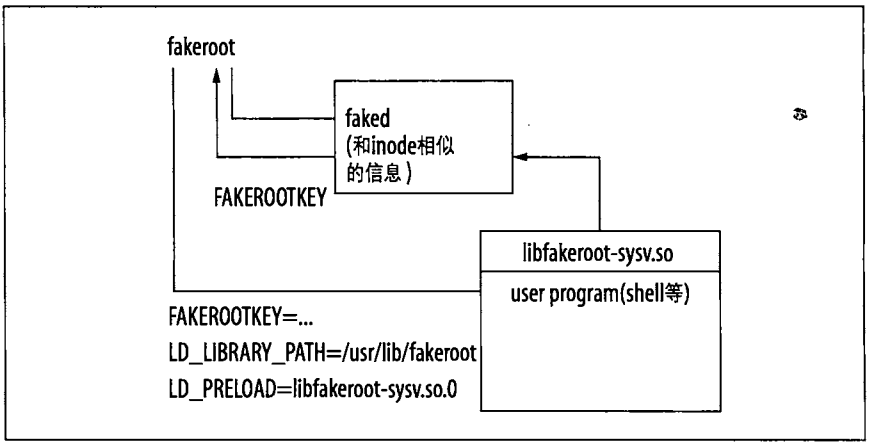
从fakeroot中运行的命令通过LD_LIBRARY_PATH以及LD_PRELOAD指定的共享对象（*/usr/lib/libfakeroot/libfakeroot-sysv.so.0*）中有些API会被lap。

```
LD_LIBRARY_PATH=/usr/lib/libfakeroot LD_PRELOAD=libfakeroot-sysv.so.0
```

libfakeroot-sysv.so.0 lap的是以下的API。

```
getuid()    geteuid()     getgid()              getegid()
mknod()
chown()     fchown()      lchown()
chmod()     fchmod()
mkdir()
lstat()     fstat()       stat()实际上 __xlstat
unlink()    remove()      rmdir()               rename()
```

这些被 lap 的 API，使用 FAKEROOTKEY 信息和 faked 交流，得到相似的 owner 等信息后返回给 user 程序。



/usr/lib/libfakeroot-sysv.so.0

为了联系 faked 拥有 rapper 的共享库为 /usr/lib/libfakeroot/libfakeroot-sysv.so.0，但也存在/usr/lib/libfakeroot-sysv.so.0。这些是在 faker 环境中安全运行 suid 程序的必需的东西。

fakeroot 有必要设定 /usr/lib/libfakeroot-sysv.so.0。在设定 LD_LIBRARY_PATH时LD_PRELOAD不能使用绝对路径名。但 LD_PRELOAD 在不使用绝对路径名时，由于二进制即便被 suid，即便二进制被 suid，会无视 LD_LIBRARY 并从 /lib 以及 /user/lib 去访问 LD_PRELOAD 指定的共享库。这时，若不存在用 LD_PRELOAD 指定的共享库，其 suid 二进制的启动则会失败。因此在 fakeroot 中要提供模型被 suid 过的共享库。在 fakeroot 环境中，想要启动 suid 二进制时，其运行将与用这个模型进行 /usr/lib/libfakeroot-sysv.so.0 来 preload 后的 faked 无关。

普通的二进制

通过 LD_LIBRARY_PATH=/usr/lib/libfakeroot LD_PRELOAD=libfakeroot-sysv.so.0将/usr/lib/libfakeroot/libfakeroot-sysv.so.0 进行 preload 之后可和 faked 通信。

suid 二进制

忽视 LD_LIBRARY_PATH，根据 LD_PRELOAD=libfakeroot-

sysv.so.0，被suid的/usr/lib/libfakeroot-sysv.so.0会被preload，由于它是模型，所以和不被preload时一样运作。

总结

在Debian中生成packsge时，在root权限成为必要时为了运行带有相似root权限的进程，要使用fakeroot。fakeroot由将lap的几个API的libfakeroot-sysv.so.0和管理与其相关联的相似root权限信息的faked这些demon组成。

—— *Fumitoshi Ukai*

第5章 运行时 Hack

Hack #58～86

在本 Hack 中，介绍了用以 C 语言为首的编译语言来实现脚本语言所带有的运行时功能的各种技巧。通常，C 及 C++ 程序没有映射功能。通过利用运行留下的符号表和调试信息，由 OS 提供的进程和共享库的相关信息，能够实现脚本语言所拥有的近似反射的功能。运行时 Hack 可以认为是实现“普通”的程序所无法实现的 Hack。本章所介绍的技巧可实际运用于脚本语言等的处理系统和调试器、Java 的 JIT 编译器等高级程序中。

HACK #58 程序转变成 main()

在本 Hack 中，叙述了在 Linux 中一般的程序在到达 main 之前是如何处理。参照的是 Linux kernel 2.4.27、glibc 2.3.6。

程序的开始运行

生成 hello world 那样的程序，一般会变成如下的 ELF 二进制。

```
% file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux
2.2.0, dynamically linked (uses shared libs), not stripped
```

使用 ldd 的话，就可以看到使用了 libc.so.6 和 /lib/ld-linux.so.2。

```
% ldd hello
        libc.so.6 => /lib/libc.so.6 (0x4001c000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

在从shell（例如bash）开始启动时就追踪该程序。首先，用命令行指定"./hello"后shell会在完成fork(2)以及标准输出的重定向处理之后，通过读出exec(2)，其进程会将fork(2)过的bash更换为指定的程序（在这里为/hello），然后运行程序。

读出execve(2)

在bash情况下，在execute_cmd.c的shell_execve()中要使用execve(2)，execve(2)是glibc源的sysdeps/unix/sysv/linux/execve.c来定义的，会像下面这样使用INLINE_SYSCALL读出系统的调用。

```
return INLINE_SYSCALL (execve, 3, file, argv, envp);
```

这个宏会在sysdeps/unix/sysv/linux/i386/sysdep.h中用#define，通过运行下面的汇编程序代码来展开。

```
movl <envp>,%edx
movl <argv>,%ecx
movl <file>,%ebx
movl $11, %eax              # execve
int $0x80
```

像这样分别将系统调用号码设置在%eax上，变量设置在%ebx、%ecx、%edx上，从而在int $0x80中读出系统调用。在运行中断命令时会发生软件中断，从用户态转换到内核态。

另外，也有在2.6的内核中使用的sysenter/syscall命令的情况，请参照**[Hack #59]**。

转入内核态

内核在初始化阶段时会像下面这样用arch/i386/kernel/traps.c的trap_init()将系统向量初始化。

```
set_system_gate(SYSCALL_VECTOR,&system_call); // SYSCALL_VECTOR = 0x80
```

因此，在运行中断$0x80并发生软件插入后，控制会转移到这里设定的system_call上。该system_call是存在于arch/i386/kernel/entry.S中的ENTRY(system_call)。控制在转移到这里后，会首先将寄存器保存在栈上，然后根据以下的命令按%eax的内容调用处在sys_call_table上的地址。

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

sys_call_table也同样在entry.S中，由于在读出时设定的是%eax = 11，所以见到第11号时变为sys_execve。

```
ENTRY(sys_call_table)
     (略)
        .long SYMBOL_NAME(sys_unlink) /* 10 */
        .long SYMBOL_NAME(sys_execve)
        .long SYMBOL_NAME(sys_chdir)
     (略)
```

sys_execve寄存于arch/i386/kernel/process.c的asmlinkage int sys_execve(struct pt_regs regs)中。

```
asmlinkage int sys_execve(struct pt_regs regs)
{
       int error;
       char * filename;

       filename = getname((char *) regs.ebx);
       error = PTR_ERR(filename);
       if (IS_ERR(filename))
              goto out;
       error = do_execve(filename, (char **) regs.ecx, (char **)
regs.edx, &regs);
       if (error == 0)
              current->ptrace &= ~PT_DTRACE;
       putname(filename);
out:
       return error;
}
```

完成设定在%ebx上的文件名的检查后，运行do_execve()。在读出系统调用时将argv设置到%edx上，将envp设置在%edx里，想到了这点，便能明白以下的调用。

```
do_execve(filename, argv, envp, ...)
```

到此为止的是架构依存的代码，这之后将变为非架构依存的代码。

execve(2)的安装 do_execve()

do_execve()被定义为fs/exec.c。在do_execve()中，首先使用open_exec()从文件系统中读取filename指定的文件。在此进行是否存在等检查。

成功打开的话将进行struct linux_binprm信息的设定。struct linux_binprm里有保存argv和envp、uid、gid等信息的结构体。

然后读取将exec化的文件最初的BINPRM_BUF_SIZE（128字节）。根据该内容选择适当的二进制处理器来进行处理。用search_binary_handler()在包含在这个已读取的文件前面的部分内，进行魔数的检测，从struct linux_binfmt *formats中搜索已匹配后的二进制处理器。

ELF二进制的读取和运行

关于ELF，fs/binfmt_elf.c的elf_format是二进制处理器，在二进制处理器中通过设定的load_elf_binary来进行ELF头的检测。若ELF头正确的话，就使用kernel_read()读取ELF的程序的头表。

在程序的头表中发现了带有PT_INTERP的段的程序头后，读取其内容。一般PT_INTERP会变为interp节，内容为"/lib/ld-linux.so.2"，由此变为elf_interpreter = "/lib/ld-linux.so.2"。

ELF解释器的解读

再次用open_exec()只解读BINPRM_BUF_SIZE部分的elf_interpreter。elf_interpreter = "/lib/ld-linux.so.2"寻址到符号链接，因为是"/lib/ld-2.3.6.so"这一共享对象，所以将会运行它。

用flush_old_exec()消除当前的程序（fork后的bash）的信息，把current的信息更新为新的程序的信息，然后将进程的内存映射初始化，并在程序头表中将成为LOAD段的部分映射在内存上。如Memsz比段的filesz大，此时存储器将分配用0隐藏的内存而不是映射文件。

最后将作为elf_interpreter的/lib/ld-2.3.6.so将被load_elf_interp()映射到存储器上，然后再次设定current的信息并开始运行。

运行开始

设置current的信息后用start_thread()开始运行。

```
start_thread(regs, elf_entry, bprm->p);
```

elf_entry为运行开始的地址。在这里elf_entry会变成作为elf_interpreter的/lib/ld- 2.3.6.so的入口地址。

将start_thread()用include/asm-i386/processor.h像下面这样#define 。

```
#define start_thread(regs, new_eip, new_esp) do {                    \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));            \
        set_fs(USER_DS);                                             \
        regs->xds = __USER_DS;                                       \
        regs->xes = __USER_DS;                                       \
        regs->xss = __USER_DS;                                       \
        regs->xcs = __USER_CS;                                       \
        regs->eip = new_eip;                                         \
        regs->esp = new_esp;                                         \
} while (0)
```

elf_entry 将设定成 %eip。其他的寄存器也应该设定为初始值。

以上search_binary_handler()、do_execve()、sys_execve()完成了。

返回到用户状态

调用 sys_execve()的地方是 arch/i386/kernel/entry.s。

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

回到这里后开始执行以下的代码。

```
        movl %eax,EAX(%esp)      # save the return value
ENTRY(ret_from_sys_call)
        cli                      # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
        restore_all:
RESTORE_ALL
```

因为reschedule()的调用，调用了kernel/sched.c.的asmlinkage void schedule()，在其中调用include/asm-i386/system.h..switch_to()，调用了 arch/i386/kernel/process.c 的—— swith_to()。这期间 %esp、%fs、%gs 寄存器被替换并执行了用户进程的环境转换。

环境转换结果，将控制交给调用 execve(2)的进程上后，由于该进程的 %eip 会像 start_thread()中被设定的那样变成 elf_interpreter 的入口地址，所以将从那里开始运行。

ELF 解释器的运行

ELF Interperter /lib/ld-2.3.6.so的入口地址是_start，这是glibc 的 sysdeps/i386/dl-machine.h的 #define RTLD_START 代码。

在这之中调用_dl_start()。把_dl_start()定义为elf/rtld.c。这里首先把bootstrap_map.l_info初始化，使用定义为sysdeps/i386/dl_machine.h的elf_machine_load_address()查询这个ELF解释器自身的存在地址。

接下来从sysdeps/generic/dl-sysdep.c的_dl_sysdep_start()之中调用ele/rtld.c的dl_main(),用dl_main()先调用process_envvars(),进行环境变量的检查。

/lib/ld-2.3.6.so 环境变量的处理

像以下的环境变量被 /lib/ld-2.3.6.so 使用。

LD_WARN	警告级别。输出与否
LD_DEBUG	动态链接的调试
LD_VERBOSE	输出版本信息
LD_PRELOAD	指定预加载的共享库
LD_PROFILE	指定取得概要信息的共享库
LD_BIND_NOW	立即绑定
LD_BIND_NOT	动态链接，不绑定
LD_SHOW_AUXV	表示从内核传递来的辅助信息
LD_HWCAP_MASK	设定硬件特性
LD_ORIGIN_PATH	设定寻找二进制的路径
LD_LIBRARY_PATH	设定搜索共享库的路径
LD_DEBUG_OUTPUT	设定调试输出的文件名
LD_DYNAMIC_WEAK	使用弱符号
LD_PROFILE_OUTPUT	设定概要文件的输出文件名
LD_TRACE_PRELINKING	跟踪预定链接
LD_TRACE_LOADED_OBJECTS	跟踪已加载的共享对象

使用LD_DEBUG，可以看到各种各样的动态链接运行。可以从LD_DEBUG=help里看到LD_DEBUG可以做什么。

```
% LD_DEBUG=help /lib/ld-2.3.1.so
Valid options for the LD_DEBUG environment variable are:
libs        display library search paths
reloc       display relocation processing
```

```
files       display progress for input file
symbols     display symbol table processing
bindings    display information about symbol binding
versions    display version dependencies
all         all previous options combined
statistics  display relocation statistics
help        display this help message and exit

To direct the debugging output into a file instead of standard
output a filename can be specified using the LD_DEBUG_OUTPUT
environment variable.
```

如果是LD_DEBUG=all的话可以全部输出。而且对LD_DEBUG_OUTPUT指定使用输出文件来代替标准错误。使用LD_SHOW_AUXV可以看到从内核传来的信息。设定LD_TRACE_PRELINKING的话可以输出连接信息。如果设定LD_TRACE_ LOADED_OBJECTS的话，可以输出加载了什么共享对象。也就是说和ldd是相同的作用。不如说ldd是这种功能实现的。

/lib/ld-2.3.6.so 的直接运行

可以直接运行 /lib/ld-2.3.6.so。

```
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked `ld.so', the helper program for shared library
executables. This program usually lives in the file `/lib/
ld.so', and special directives in executable files using ELF
shared libraries tell the system's program loader to load the
helper program from this file. This helper program loads
the shared libraries needed by the program executable, prepares
the program to run, and runs it. You may invoke this helper
program directly from the command line to load and run an ELF
executable file; this is like executing that file itself, but
always uses this helper program from the file you specified,
instead of the helper program file specified in the executable
file you run. This is mostly of use for maintainers to test new
versions of this helper program; chances are you did not intend
to run this program.

  --list                list all dependencies and how they are
                        resolved
  --verify              verify that given object really is a
                        dynamically linked object we can handle
  --library-path PATH   use given PATH instead of content of the
                        environment variable LD_LIBRARY_PATH
  --inhibit-rpath LIST  ignore RUNPATH and RPATH information in
                        object names
in LIST
```

举例： --list选项按照ldd来操作。

没有指定选项的ELF二进制赋予参数，同那个ELF二进制直接执行一样执行。

共享对象的加载

首先，预先加载的共享对象由LD_PRELOAD来指定的话，加载共享对象。接着看到 /etc/ld.so.preload同样处理预先载入。

预装载的处理结束后，用执行的ELF二进制的NEEDED来处理被指定的共享对象。

_GLOBAL_OFFSET_TABLE_的再配置

```
ELF_DYNAMIC_RELOCATE (&bootstrap_map, 0, 0);
```

以上部分，对于_GLOBAL_OFFSET_TABLE_进行重定位。ELF_DYNAMIC_RELOCATE()是用elf/dynamic-link.h来#define的。实际处理是用glibc、sysdeps/i386/dl-machine.h和elf_machine_ runtime_setup()来执行的。

elf_machine_runtime_setup()是设定_GLOBAL_OFFSET_TABLE_的头部两个入口。

```
got[1] = (Elf32_Addr) 1; /* Identify this shared object. */
got[2] = (Elf32_Addr) &_dl_runtime_resolve;
```

这个dl_runtime_resolve是解决符号的代码。调出用共享对象定义的函数时，使用这个_dl_runtime_resolve，便能解决符号的值。程序方面调用函数时变为下面的符号。

```
.text
      ....
  call    PLT[n]的相对地址
      ....
.plt:

   PLT[n]:
      jmp *GOT[n]
   PLT_resolv[n]:
      push  entry index
      jmp   resolver

resolver
      pushl GOT[1]
   jmp *GOT[2]
```

```
.got:  ...
       GOT[1] 用于识别这个共享对象自身的信息
       GOT[2] dl_runtime_resolve
       GOT[3] PLT_resolv[n]
```

开始，从符号中调出PLT[n]，会跳转到GOT[n]中的地址。延迟resolve的情况下，GOT[n]的初始值通常为PLT_resolv[n]该跳转之后的值。在PLT_resolv[n]中，把入口索引进栈，然后跳向resolver的代码。Resolver将GOT[1]压栈后跳到GOT[2]的地址上。在GOT[2]上的地址是动态生成dl_runtime_resolve,因此会读出_dl_runtime_resolve。在-dl-runtime-resolve中，引用stack上的GOT[1]和入门索引，以解决symbol的值替换GOT[n]；然后跳转到那里进行实际处理。

接下来从代码中调出PLT[n]是也要跳转到GOT[n]的地址，因为一开始调出时GOT[n]的地址已经变成symbol解决过的地址了，所以可以直接跳转到共享对象的代码。

还有LD_BIND_NOW=1的时候，动态链接那个wind处理要先进行。

进入到 EFL 二进制实体中

动态链接的处理完成后，原来的用ELF二进制从入口地址设定的地址开始处理。通常用_start符号跟在代码后。

最终调用glibc的sysdeps/generic/libc-start.c的__libc_start_main()进行初始化，最后调用main()。

```
result = main (argc, argv, __environ);
```

总结

对执行ELF二进制程序时如何开始进行处理进行解释。一部分基于核心的ELF句柄，一部分基于ELF解释器输入的动态链接，设定程序的运行环境。另外，对在ELF解释器中能利用的环境变量也进行了说明。

—— *Fumitoshi Ukai*

怎样调用系统调用

在本Hack中，介绍了在x86架构上的Linux中，将怎样调用系统调用。

在现代的UNIX系的操作系统中，系统的操作模式（也称为环境等）主要分为用户态和内核态两种类型（实际上还有插入环境）。

用户环境可以说是通常的程序运行模式。与此相反，运行硬件和系统的控制操作的模式是内核环境。系统调用是从用户环境到内核环境处理的转换，此时将指出OS执行了怎样的处理。举例说明，在末尾字符的输出系统路径是writes，知道自己的项目进程ID的系统路径是getpid。

这些系统调用用户程序开始表示的状态是用strace指令来确认的。详细请参照**[Hack #82]**。用strace来追踪系统调用。另外，请参照作为应用例子的**"[Hack #25]不用glibc书写Hello World。"**

以下是从x86上的Linux开始运行系统调用的方法。

使用syscall函数

作为系统调用的函数，有syscall()函数。例如，请看下面的程序。

```
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
  int ret;
  ret = syscall(__NR_getpid);
  printf("ret = %d pid = %d\n", ret, getpid());
  return 0;
}
```

在这里，使用__NR_getpid这个名字，但编译时变换成"20"的数字。名字和号码的对应关系是用/usr/include/bits/syscall.h和/usr/include/asm-i486/unistd.h等文件来定义。执行时如下所示。

```
% ./syscall-func
ret = 4846 pid = 4846
```

很明显正常运行。

使用 int 0x80

接下来，上述的syscall()函数，实际上是glibc中调出系统调用的操作重复使用的东西。那么，实际上发生了什么呢？

从前使用的方法是user程序发行x86处理器的软件中断(trip)用的命令int。发出了int 0x80的命令后，控制将从user程序上转移到Linux内核上，在内核这方面确认发生了系统调用。来看看下面这个例子。

```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
int main(void)
{
  int ret;
  asm volatile ("int $0x80" : "=a" (ret) : "0" (__NR_getpid));
  printf("ret = %d pid = %d\n", ret, getpid());
  return 0;
}
```

第7行中gcc出现了in line。简单地解释一下，发出int 0x80命令时，作为输入，在eax上设置20，作为输出将值从eax寄存器中代入到ret上。

发出int0x80命令时，在eax寄存器中放入系统调用的编号，在这里放入20，运行getpid。而在取得参数的情况下，参数的顺序向ebx、ecx、edx、eci、edi、ebp寄存器代入值。

系统调用完成后，结果的值将被代入到eax寄存器中。例如，将当前的进程ID作为结果代入到了getpid系统调用中。在x86中，错误发生时，错误值变为负数放入eax寄存器中。例如，在-EINVAL返回时，glibc将设定EINVAL为errno，并将返回值改为-1放入eax寄存器。若负数超过了某范围的话则当作正常值来对待，不改写errno和返回值。在不同的架构中，errno值可能会有别的用途。

使用 sysenter

从Linux内核2.6开始，上述的int 0x80之外也支持别的调出系统指令的方法-sysenter。而且能变更为被称为支持复数系统调用的vsyscall。支持这样变更的最大理由是在Pentium 4以后，int 0x80命令花费的时间比Pentium III更多。在这里用户不用在意任何东西，在内核内部有可变更系统调用机制的支持。**[Hack #66]**没有提及。Linux内核2.6开始进程的内存空间的最后要追加特别邻域。以/proc/<pid>/maps为例来说明以下的0xffffe000～0xfffff000区域：

```
% cat /proc/self/maps | grep vdso
ffffe000-fffff000 ---p 00000000 00:00 0 [vdso]
```

另外，这个地址是根据Linux内核版本的不同而变化的。在下面的例子中，假定范围从0xffffe000开始。由于随着程序的启动会变更地址的运行空间，从而提高安全防范性。在这里假定这个设定无效，这里实际上在内核中随意为进程分配适当的空间，成为调出vsyscall的页。关于vdso在*http://www.trilithium/johan/2005/08/linux-gate/*有详细的说明。读出这一页，执行readelf -S的结果如下。这很明显成为了一个ELF对象。

```
% dd if=/proc/self/mem of=vdso bs=1 skip=0xffffe000 count=4096
% readelf -S vdso
Section Headers:
  [Nr] Name              Type      Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL      00000000 000000 000000 00      0   0  0
  [ 1] .hash             HASH      ffffe0b4 0000b4 000038 04   A  2   0  4
  [ 2] .dynsym           DYNSYM    ffffe0ec 0000ec 000090 10   A  3   5  4
  [ 3] .dynstr           STRTAB    ffffe17c 00017c 000056 00   A  0   0  1
  [ 4] .gnu.version      VERSYM    ffffe1d2 0001d2 000012 02   A  2   0  2
  [ 5] .gnu.version_d    VERDEF    ffffe1e4 0001e4 000038 00   A  3   2  4
  [ 6] .text             PROGBITS  ffffe400 000400 000060 00  AX  0   0 32
...
```

从0xffffe400开始的区域，实际执行调出系统指令。

```
% objdump -d vdso --start-address=0xffffe400
...
ffffe400 <__kernel_vsyscall>:
ffffe400:       51              push   %ecx
ffffe401:       52              push   %edx
ffffe402:       55              push   %ebp
ffffe403:       89 e5           mov    %esp,%ebp
ffffe405:       0f 34           sysenter
ffffe407:       90              nop
...
```

试着用实例程序来确认。

```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
int main(void)
{
  int pid;
  asm volatile ("call *%2 \n" : "=a" (pid) :
                "0" (__NR_getpid), "S"(0xffffe400));
  printf("ret = %d pid = %d\n", pid, getpid());
  return 0;
}
```

结果如下：

```
ret = 29261 pid = 29261
```

而且，这个地址0xffffe400是在[Hack #27] glibc中说明过的AT_SYSINFO。而且，glibc内部上面标记的地址，不用直接指定就可以调出。

总结

在x86架构上的Linux，说明了关于如何调用系统指令的方法。而且也说明了在内部根据int 0x80命令的方法和根据sysenter的方法。

—— *Masanori Goto*

用 LD_PRELOAD 更换共享库

在本Hack中介绍了为了更换共享库运作而使用LD_PRELOAD的方法。

共享库的更换

例如在hostname(1)中，作了将host名返回为与实际host名不同的host名的变更。在hostname(1)为了得到host名使用了gethostname(3)，试着用ltrace后便能看到读出了gethostname(3)。

```
% hostname
akira.fsij.org
% ltrace hostname
(..)
gethostname("akira.fsij.org", 128)                    = 0
(..)
%
```

在hostname(1)中，gethostname(3)读出了共享库libc.so.6中的代码。

```
% nm -D /bin/hostname | grep gethostname
         U gethostname
% ldd /bin/hostname
         libresolv.so.2 => /lib/tls/i686/cmov/libresolv.so.2
(0xb7fc7000)
         libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e91000)
         /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb7fea000)
% nm -D /lib/libc.so.6| grep gethostname
000d0e90 W gethostname
```

通过将共享对象指定为环境变量LD_PRELOAD并运行，先链接LD_PRELOAD的共享对象。因此，将同名函数定义在共享对象中，与共享库相比，用LD_PRELOAD指定的共享对象中的代码更容易链接并能被读出。

为了更改gethostname(3)，要生成如下的共享对象。

```
% cat -n gethostname.c
     1 #include <stdlib.h>
     2 #include <string.h>
     3
     4 int
     5 gethostname(char *name, size_t len)
     6 {
     7           char *p = getenv("FAKE_HOSTNAME");
     8           if (p == NULL) {
     9                   p = "localhost";
    10           }
    11           strncpy(name, p, len-1);
    12           name[len-1] = '\0';
    13           return 0;
    14  }
% cc -shared -fPIC -o gethostname.so gethostname.c
%
```

gethostname(3)是非常简单的代码，若设定了环境变量FAKE_HOSTNAME，则把它当作host名，没有被设定的话，则将localhost作为主名返回。为了使用这个共享对象，在环境变量LD_PRELOAD中要指定其路径名。

```
% LD_PRELOAD=./gethostname.so hostname
localhost

% FAKE_HOSTNAME=sai.fsij.org LD_PRELOAD=./gethostname.so hostname
sai.fsij.org
```

current directory 时，必须像[./]这样指定。在不含[./]时，用环境变量LD_LIBRARY_PATH指定为directory，必须将这个共有对象放置在标准的库path上。

```
% LD_PRELOAD=gethostname.so hostname
hostname: error while loading shared libraries: gethostname.so:
cannot open shared
object file: No such file or directory

% LD_PRELOAD=gethostname.so LD_LIBRARY_PATH=/tmp hostname
localhost
```

即使在用LD_PRELOAD指定的共享对象中有未定义的符号，只要共享对象依存的库提供了这些，便可在加载时执行适当的符号值的解决。

```
% nm -D gethostname.so
00001820 A _DYNAMIC
000018f4 A _GLOBAL_OFFSET_TABLE_
w _Jv_RegisterClasses
0000191c A __bss_start
         w __cxa_finalize
         w __gmon_start__
0000191c A _edata
00001920 A _end
000007e0 T _fini
000005d4 T _init
         U getenv
00000734 T gethostname
         U strncpy
% ldd ./gethostname.so
       libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eb8000)
       /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

这种情况下，getenv(3)和strncpy(3)使用了libc.so.6．中的东西。

总结

使用LD_PRELOAD ，说明变更共享库路径的方法。

——*Fumitoshi Ukai*

用 LD_PRELOAD 来 lap 既存的函数

在本 Hack 中，介绍了用 LD_PRELOAD 来 lap 既存函数的方法。

Lap 既存的函数

在**[Hack #60]**用 LD_PRELOAD 更换共享库中介绍了用 LD_PRELOAD 将既存的函数置换为其他代码的方法。考虑一下下面的运用。

在有的服务器端二进制中，套接字被bind(2)时不指定地址，可以选择INADDR_ANY。一般使用情况没有问题，复制接口的话，想要安装特定的接口连接，有源代码的如果改做成二进制的话，一定要尽可能正确地设定bind地址。但是在没有源代码情况下怎么办呢？也有用iptables等切断内核级指定的接口的地址以外的连接的方法。但这种方法和同样在接口启动别的服务器就比较麻烦了。因此，使用LD_PRELOAD，考虑替换bind(2)，例如下面的指令。

```
% cat -n bindwrap0.c
```

```
 1 #include <string.h>
 2 #include <sys/types.h>
 3 #include <sys/socket.h>
 4 #include <netinet/in.h>
 5
 6 int
 7 bind(int sockfd, const struct sockaddr *my_addr,
socklen_t addrlen)
 8 {
 9     struct sockaddr_in saddr;
10     if (my_addr->sa_family == AF_INET
11         && ((struct sockaddr_in *)my_addr)->sin_addr.s_addr
== INADDR_ANY) {
12         struct in_addr sin_addr;
13         inet_aton(getenv("BIND_ADDR"), &sin_addr);
14         memset(&saddr, 0, sizeof(saddr));
15         saddr.sin_family = AF_INET;
16         saddr.sin_port = ((struct sockaddr_in *)my_addr)-
>sin_port;
17         saddr.sin_addr = sin_addr;
18         return bind(sockfd,
19                     (const struct sockaddr *)&saddr,
sizeof(saddr));
20     }
21     return bind(sockfd, my_addr, addrlen);
22 }
```

但是，即使充分考虑了也不能运转自如。因为在bind中还要调出bind，形成再循环圈。并且因为没有终结条件不断地再循环，无法流通，便堵死了。那么应该怎么做呢？

指向既存函数的指针

要点是，从这个替换的bind调出的bind ，不是这个新的bind，而是原来的bind。为了得到原来的bind有几个方法，最简单的方法是使用handle RTLD_NEXT，用dlsym(3)调出"bind"。handle**[Hack #62]**是RTLD_NEXT扩展的特殊代名，在共享对象的下一共享对象以后取得寻找符号值。这种情况下因为替换的bind本身是这个共享对象的符号，使用RTLD_NEXT可以查询下面，可以取得在libc.so.6上的bind符号的值。RTLD_NEXT是GNU.的扩展，在包含dlfcn.h之前有必要先定义GNU_SOURCE。

```
#define _GNU_SOURCE
#include <dlfcn.h>

static int (*bind0)(int sockfd, const struct sockaddr *myaddr,
socklen_t addrlen);
..
```

```
bind0 = dlsym(RTLD_NEXT, "bind");
```

作为其他的方法，有把既存函数含有的共享数据库 dlopen(3)，然后进行 dlsym(3)的方法。即使 dlopen(3)想要多次加载同样共享的数据库，也只能进行一次映射，所以没有说明问题。如果映射的是系统调用，也可以使用 syscall(2)调出系统调用。

用于 lap 的共享对象

使用 RTLD_NEXT 的话，映射共享对象代码如下。

```
% cat -n bindwrap.c
    1 #define _GNU_SOURCE
    2 #include <dlfcn.h>
    3 #include <string.h>
    4 #include <sys/types.h>
    5 #include <sys/socket.h>
    6 #include <netinet/in.h>
    7
    8 static int (*bind0)(int sockfd, const struct sockaddr
*myaddr, socklen_t addrlen);
    9 static struct in_addr sin_addr;
   10
   11 void __attribute__((constructor))
   12 init_bind0()
   13 {
   14     bind0 = dlsym(RTLD_NEXT, "bind");
   15     inet_aton(getenv("BIND_ADDR"), &sin_addr);
   16 }
   17
   18 int
   19 bind(int sockfd, const struct sockaddr *my_addr, socklen_t
addrlen)
   20 {
   21    struct sockaddr_in saddr;
   22    if (my_addr->sa_family == AF_INET
   23        && ((struct sockaddr_in *)my_addr)->sin_addr.s_addr ==
INADDR_ANY) {
   24        memset(&saddr, 0, sizeof(saddr));
   25        saddr.sin_family = AF_INET;
   26        saddr.sin_port = ((struct sockaddr_in *)my_addr)-
>sin_port;
   27        saddr.sin_addr = sin_addr;
   28        return (*bind0)(sockfd,
   29        (const struct sockaddr *)&saddr, sizeof(saddr));
   30    }
   31    return (*bind0)(sockfd, my_addr, addrlen);
   32 }
  % cc -shared -fPIC -o bindwrap.so bindwrap.c -ldl
```

在这个例子中，把init_bind0()列入了constructor的属性中。在加载这个共享对象时只运行一次bind0和sin_addr的初始化。这个共享对象可以像如下所说的那样使用。

```
% LD_PRELOAD=./bindwrap.so BIND_ADDR=127.0.0.1 daemon-program
```

像这样如果事先下载bindwrap.so的话，就可以把BIND_ADDR环境变量中指定的地址代替INADDR_ANY来使用。从netstat -a来看的话，0.0.0.0就成了Local Address。在这个例子中127.0.0.1就成了Local Address。

总结

从使用RTLD_NEXT取得的lap函数读出存在函数的point。像这样指定共享对象，就可以做到LD_PRELOAD后lap存在的函数。

—— *Fumitoshi Ukai*

HACK #62 用dlopen进行运行时的动态链接

在此Hack中，对于运行时的动态链接dlopen(3)的使用方法进行说明。

动态链接

有这样的情况，根据程序的不同，决定了动态运行时要使用多少共享对象。比如说浏览器的plug-in，在建造浏览器时有什么plug-in，一开始是不知道的。Plug-in是后来完成的。运行时才动态链接plug-in的。这是怎样实现的呢？

使用dlopen(3)的程序的例子

使用dlopen(3)、dlsym(3)、dlclose(3)的程序如下：

```
% cat -n dlsay.c
     1 #include <stdio.h>
     2 #include <dlfcn.h>
     3
     4 int
     5 main(int argc, char *argv[])
     6 {
     7         void *handle;
```

```
 8          char *(*msg)();
 9          char *error;
10
11          handle = dlopen(argv[1], RTLD_LAZY);
12          if (handle == NULL) {
13          fprintf(stderr, "load error %s: %s\n", argv[1], dlerror());
14          exit(1);
15          }
16          dlerror(); /* clear error */
17          msg = (char *(*)()) dlsym(handle, argv[2]);
18          if ((error = dlerror()) != NULL) {
19                  fprintf(stderr, "dlsym error %s: %s\n",
argv[2], error);
20                  exit(1);
21          }
22          printf("%s\n", (*msg)(argc > 3 ? argv[3] : NULL));
23          dlclose(handle);
24          exit(0);
25 }
26
% cc -o dlsay dlsay.c -ldl
```

dlsay是对最初的变量指定共享库名。把字符串作为循环函数，把下一个命令行变量作为那个函数的变量。读出并表示下一个变量指定的符号的程序。这可以像下面一样做出共享库。

```
% cat -n hello.c
    1 #include <stdio.h>
    2 char *hello(char *arg) {
    3         static char buf[4096];
    4
    5         snprintf(buf, sizeof buf, "hello, %s", arg);
    6         return buf;
    7 }
% cc -shared -fPIC -o hello.o hello.c
```

实际操作如下：

```
% ./dlsay ./hello.so hello world
hello, world
```

除hello.so外，若是其中包含将char *作为变量返回char *的函数的共享对象的话，则可任意使用。例如，dlsay可以像下面一样的运行。

```
% ./dlsay libc.so.6 tmpnam
/tmp/fileCZKwXZ
```

dlopen(3)..dlsym(3)..dlclose(3)

简单地介绍一下是如何实现dlsay的。

首先，用dlopen(3)加载共享库。dlopen(3)指定共享库后，就会加载并改变句柄。如果共享库中含有“/”，一定要解释成路径名或者是相对路径名，./hello.so的情况下加载当前目录中的hello.so。有当前目录的情况下明确的标出“./”有必要指定路径名。像以下执行时出错的情况也有(取决于hello.so存在于哪里)。

```
% ./dlsay hello.so hello foo
load error hello.so: hello.so: cannot open shared object file:
No such file or
directory
```

那么，不含有“/”的情况下使用哪种共享对象呢？不含“/”的情况下，按二进制设定的.RPATH.、.RUNPATH、环境变量LD_LIBRARY_PATH等指定的方向寻找。否则的话寻找/etc/ld.so.cache和/lib、/usr/lib。如libc.so.6这样指定的情况，不管在哪里有当前目录，都加载/lib/libc.so.6。

用dlopen(3)指定的RTLD_LAZY是加载时，逃避解决符号的值。代替RTLD_LAZY指定RTLD_NOW，加载时解决符号的值。如果解决符号的值失败的话，dlopen(3)本身就失败了。RTLD_LAZY的场合有无法解决的符号时，用dlsym(3)也无法解决，dlopen(3)出现错误，而且程序变成RTLD_LAZY的话，对环境变量LD_BIND_NOW设置为字符串，可以和RTLD_NOW进行相同的运行。

对于RTLD_LAZY或者RTLD_NOW来说，可以设定RTLD_LOCAL或是RTLD_GLOBAL。如果是RTLD_GLOBAL的话，用dlopen(3)可以自动使用读入共享对象上的符号解决其他共享对象上的符号。使用RTLD_LOCAL、dlopen(3)来读入共享对象的符号，没有其他任何影响。

关于用dlopen(3)返回的句柄，可以得到那个含有共享对象的符号的值。为此要使用dlsym(3)。赋予dlsym共享对象的代名和符号名，在那个共享对象中返回被定义的符号值。找不到符号的场合下，返回NULL。错误的内容用dlerror()查询。例如下面的执行后，“hi”、“hello.so.”，没有存在，故出错。

```
% ./dlsay ./hello.so hi world
dlsym error hi: ./dlsay: undefined symbol: hi
```

因为可以用dlsym(3)取得的只是符号的值，这个符号是怎样的变量，函数是由调用一方作适当的选择。

调用dlclose(3)时，那个句柄相对应的共享对象的坐标显示出来。同样由复制共享对象dlopen(3)时，被dlopen(3)的次数在被dlclose(3)之前，实际上坐标留在那里就行了。

GNU 扩展

用glibc提供的dlopen(3)有几个GNU扩展。GNU扩展是#define_GNU_SOURCE做#include <dlfcn.h>使用。

特殊的句柄（RTLD_NEXT..RTLD_DEFAULT）

RTLD_NEXT是“下一个”共享对象以后开始寻找符号时使用。用共享对象对一些函数lap时，获得对lap以前的函数的指针的情况下使用RTLD_NEXT。在**[Hack #61]** LD_PRELOAD中运用了这种机能。RTLD_DEFAULT是搜寻全局符号时使用的。

```
void *dlvsym(void *handle, char *name, char *version);
```

虽然dlvsym(3)与dlsym(3)相像，但是指定符号版本寻找符号的地方是不同的。

```
int dladdr(void *addr, Dl_info *info);
```

dladdr(3)返回包含叫做addr的指针的对象的符号的信息。

```
typedef struct {
  const char *dli_fname; /* 共享对象的文件名 */
  void *dli_fbase;       /* 共享对象的登录地址 */
  const char *dli_sname; /*符号名 */
  void *dli_saddr;       /* 符号的值 */
} Dl_info;
```

```
int dladdr1(void *addr, Dl_info *info, void **exra_info,
int flags);
```

通过对flag指定RTLD_DL_SYMENT..RTLD_DL_LINKMAP，在extra_info中可取得ElfNN_Sym *或struct linkmap *。

```
int dlinfo(void *handle, int request, void *arg);
```

可以取得共享对象的各种信息，用request指定取得的信息。

request	arg ..	说明
RTLD_DI_LINKMAP	struct linkmap **	link map信息
RTLD_DI_SERINFO	Dl_serinfo *	library search bus信息
RTLD_DI_SERINFOSIZE	Dl_serinfo * 1	ibrary search bus数目
RTLD_DI_ORIGIN	char *	$ORIGIN的目录名

其他注意事项

如果载入的共享对象有_init、_fini符号，那么_init()会在dlopen(3)，_fini()会在dlclose(3)时被自动调用。

应该使用构造、析构属性。构造上__attribute__ ((constructor))属性，析构加上__attribute__((destructor))属性来定义函数。

总结

关于为动态载入共享对象的API、dlopen(3)进行了说明。使用dlopen(3)的话，可以变更运行时必须加载的共享对象。只要不要用RTLD_GLOBAL、dlopen(3)，采用dlsym(3)加载共享对象的符号，其他没有影响，能同时载入几个拥有同样符号的共享对象。

—— *Fumitoshi Ukai*

用C表示回溯

本Hack介绍了使用glibc函数用C表示回溯的方法。

什么是回溯

大致上，回溯就是到达当前函数的路径。比如说，如果运行下面的程序Ruby，那么1/0行发生例外。表示back trace的同时，程序异常终止。

```
def foo
  1 / 0
end

def main
  foo
end

main
```

这个例子叫做从main到foo，在foo中1 / 0的部分发生了例外。

```
% ruby divide-by-zero.rb
divide-by-zero.rb:2:in `/': divided by 0 (ZeroDivisionError)
        from divide-by-zero.rb:2:in `foo'
        from divide-by-zero.rb:6:in `main'
        from divide-by-zero.rb:9
```

回溯可以从一串叫做堆栈帧的数据中复原。stack framede是每次调用函数时积在堆栈中的返回地址和变量数据的集合。

同样的程序这次用 C 写出来并试着运行。

```
int foo() {
    return 1 / 0;
}

int main() {
    foo();
    return 0;
}
```

这样一来，虽可以标出错误信息，却不能标出 back trace。

```
% ./a.out
zsh: 6392 floating point exception (core dumped) ./a.out
```

在 C 程序中，使用了 gdb 的话则可以标出 back trace。

```
% gdb a.out core
(gdb) bt
#0 0x08048369 in foo ()
#1 0x08048389 in main ()
```

在编译时 gcc 附上了 -g 选项的情况中，可表示出文件名和行号码。这是由于运用了隐藏在可执行文件中的调试信息。

```
(gdb) bt
#0 0x08048369 in foo () at divide-by-zero.c:2
#1 0x08048389 in main () at divide-by-zero.c:6
```

在 C 中表示回溯

使用了包含在 glibc 中的 back trace()和backtrace_symbols_fd()后，便可表示出运行中 C 程序的 back trace 。关于这两个函数的介绍在 glibc 的手册中有记载。下面介绍了简单的使用实例。

```
#include <execinfo.h>

void foo() {
    void *trace[128];
    int n = backtrace(trace, sizeof(trace) / sizeof(trace[0]));
    backtrace_symbols_fd(trace, n, 1); //STDOUT ......
}

int main() {
    foo();
```

```
    return 0;
}
```

用gcc -g -rdynamic将这一程序编译并运行后会标示出下面这样的back trace。由于在i386上backtrace_symbols_fd()要利用dynsym节内的信息（内部化使用那个dladdr），因而-rdynamic很必要。

```
% ./a.out
./a.out(foo+0x1f)[0x8048693]
./a.out(main+0x15)[0x80486d0]
/lib/libc.so.6(__libc_start_main+0xc6)[0x40032e36]
./a.out[0x80485d1]
```

虽然不容易看懂，但正在从main中调用foo这一点也很清楚。另外，使用了GNU binutils的addr2line之后，通过使用ELF二进制中的调试信息，还可表示源代码的文件名和行号码。

```
% ./a.out | egrep -o '0x[0-9a-f]{7}' | addr2line -f
foo
/home/tmp/c/backtrace.c:5
main
/home/tmp/c/backtrace.c:11
??
??:0
_start
../sysdeps/i386/elf/start.S:105
```

原理的概述和在x86中的安装

本Hack介绍的backtrace函数，通过stack frame来实现。在读出函数时，不保存函数的返回位置则不能返回到原来的地方，所以要在stack上保存其返回处，但该方法因构造的不同而不同。例如在x86种，便可自动取得back trace。

```
#include <stdio.h>

typedef struct layout {
    struct layout *ebp;
    void *ret;
} layout;

void print_backtrace() {
    layout *ebp = __builtin_frame_address(0);
    while (ebp) {
        printf("0x%08x\n", ebp->ret);
        ebp = ebp->ebp;
    }
}
```

```
void foo() {
    print_backtrace();
}

int main() {
    foo();
    return 0;
}
```

在x86中，一般ebp寄存器会变成为指向堆栈帧的指针，可通过GCC扩展的__builtin_frame_address和in line assemble来取得该指针。在这个样本中虽只标出地址，但和“**[Hack #67]**用.libbfd取得符号的一览表”组合之后，便可表示函数名等信息。

关于x86的stack frame，用glibc的backtrace.c的说明中的下图表示简单易懂。

```
        +-----------------+        +-----------------+
%ebp -> | %ebp last frame--------> | %ebp last frame--->...
        |                 |        |                 |
        | return address  |        | return address  |
        +-----------------+        +-----------------+
```

异常终止时的back trace

在环境量LD_PRELOAD中设置了/lib/libSegFault.so后，在程序异常终止时可标出back trace。

作为尝试，将最初的最单纯的C的程序设置为LD_PRELOAD=/lib/libSegFault.so SEGFAULT_SIGNALS =all 后并运行的话，则会标示出下面的信息。

```
% export LD_PRELOAD=/lib/libSegFault.so
% export SEGFAULT_SIGNALS=all
% ./a.out
*** Floating point exception
Register dump:

EAX: 00000001  EBX: 40150880  ECX: 00000001  EDX: 00000000
ESI: 40016540  EDI: bfffe894  EBP: bfffe828  ESP: bfffe824

EIP: 080485f9  EFLAGS: 00010286

CS: 0023   DS: 002b   ES: 002b   FS: 0000   GS: 0000   SS: 002b

Trap: 00000000 Error: 00000000 OldMask: 00000000
ESP/signal: bfffe824 CR2: 00000000
```

```
Backtrace:
./a.out(foo+0x15)[0x80485f9]
./a.out(main+0x15)[0x8048619]
/lib/libc.so.6(__libc_start_main+0xc6)[0x40036e36]
./a.out[0x8048541]
Memory map:

08048000-08049000 r-xp 00000000 09:00 5538441 /home/satoru/tmp/a.out
08049000-0804a000 rw-p 00000000 09:00 5538441 /home/satoru/tmp/a.out
40000000-40016000 r-xp 00000000 08:01 700392  /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 08:01 700392  /lib/ld-2.3.2.so
40017000-40018000 rw-p 00000000 00:00 0
40018000-4001b000 r-xp 00000000 08:01 700650  /lib/libSegFault.so
4001b000-4001c000 rw-p 00002000 08:01 700650  /lib/libSegFault.so
40021000-40149000 r-xp 00000000 08:01 700628  /lib/libc-2.3.2.so
40149000-40151000 rw-p 00127000 08:01 700628  /lib/libc-2.3.2.so
40151000-40154000 rw-p 00000000 00:00 0
bfffd000-c0000000 rwxp ffffe000 00:00 0
zsh: 11875 floating point exception (core dumped) ./a.out
```

在这个回溯中，标示出了包含正在进行0的除法运算的`foo()`的回溯。

作为`/lib/libSegFault.so`的看门人，有段错误专用的`catchsegv`命令，`catchsegv`只是shell的脚本。使用`catchsegv`时像以下这样运行：

```
% catchsegv ./a.out
```

总结

使用了glibc的函数后，在C中也可简单地标出回溯。当作调试用的信息来使用则很方便。本Hack有以下3个要点。

- glibc中包含了`backtrace()`、`backtrace_symbols()`、`backtrace_symbols_fd()`。
- 可标示出仅由于将`./lib/libSegFault.so`指定到环境变量LD_PRELOAD上而产生异常终止时的back trace。
- 只是段错误可使用`catchsegv`命令。

—— *Satoru Takabayashi*

HACK #64 检测运行中进程的路径名

本Hack中，介绍了检测运行中进程的路径名。要将其与OS以及针对不同用途的各种方法区别使用。

在想检测只能在运行过的进程中取得的信息时，需要通过解析可执行文件而从中获取信息。本Hack中介绍了取得可执行文件的路径的方法。虽然这个被认为很简单，但由于不存在只能在标准C中确实可行的方法，所以要和OS以及其他针对各种用途的方法区别开来。

从argv[0]获取

在main的第2变量argv中，存放了已运行的命令行字符串的排列。在见到其最初的要素argv[0]后，在很多时候便可知道运行文件的路径。但是，仅仅这样的话，shell在搜索过环境变量$PATH后，无法找到文件的路径。例如，对shell来说，运行ls命令时的argv[0]变成了“ls”，但一般可执行文件的全路径为/bin/ls等。

作为取得path的方法中的一种，若能从argv[0]中得到path则用这个方法，若不能得到则有通过模拟shell的运行来检查path的方法。此时，按顺序搜索用环境变量$PATH表示的目录，发现了文件后则将其当作path。这个方法可在大多数OS中应用（要注意在Unix和Windows中环境变量$PATH的分隔符不尽相同），但安装会比较麻烦，并且运行时会有一定开销。

另外，在Unix等OS中用shebang（在文件的第一行#!来指定自身的解释器的声明）来启动程序时，argv[0]是OS依存的。在用shebang指定全路径时，在大多数的OS（SysVR4、SunOS、Solaris、IRIX、BSDI、BSD-OS、OpenBSD、DU、Unixware、Linux 2.4、FreeBSD）中，该全路径将作为argv[0]被传送，但是在一些OS（Tru64 4.0、AIX 4.3、5.1、Linux 2.2.）中会变为可执行文件名而不是路径名，在HP-UX中（非运行的文件）则变成script的全路径。详细信息在《#! - the Unix truth as far as I know it》（*http:// homepages.cwi.nl/~aeb/std/hashexclam.html*）中有记载。

利用UNIX OS的procfs

在提供了OS取得可执行文件的路径的方法时，利用这些方法非常便利。

若能利用UNIX OS的procfs的话，则会变得更加便利。procfs是OS为

提供运行中的进程信息而mount到一般的/proc上的虚拟文件系统。/proc的下面有进程ID的解释器，在它的下面有带有各个进程信息的虚拟文件。

想要查询自身的进程信息时，在Linux中由于/proc/self会变为面向自身进程ID的解释器的符号链接，查询就很简单。另外，在各程序ID的索引中有叫做exe的符号链接，由于这个符号链接的前端会变成正在运行的文件，故可以通过查询/proc/self/exe符号链接的值来取得路径。

用readlink(2)来查询symbolic链接的值。下面表示的是在带有procfs的Linux环境中标示出运行文件的全路径的程序。

```
#include <unistd.h>
#include <stdio.h>
#include <assert.h>

int main() {
    int ret;
    char fullpath[4096]; // 要注意path的最大长度会因系统的不同而不同。
    ret = readlink("/proc/self/exe", fullpath, 4096);
    assert(ret != -1);
    printf("%s\n", fullpath);
    return 0;
}
```

procfs是OS依存的，"/proc/self/exe"因OS种类和版本的不同而不同。例如在FreeBSD中运行了"/proc/curproc/file"这个方法后，便可根据查询"/proc/<process_id>/exe"取得知道进程的ID的其他进程的全路径。另外，即使是在OS没有提供"/proc/self"等符号链接的环境中，也可以用getpid(2)来查询自身的进程ID从而得到全路径。下面介绍的是用getpid(2)来取得全路径的代码。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>

int main() {
    int ret;
    char buf[256];
    char fullpath[1024];
    sprintf(buf, "/proc/%d/exe", getpid());
    ret = readlink(buf, fullpath, 1024)
    assert(ret != -1);
    printf("%s\n", fullpath);
    return 0;
}
```

利用 Win32 API

Windows 环境中虽然没有 procfs，但可用 Win32 API 来轻松查询自身的名称。所使用的 API 是取得可执行文件的用户名的 GetModule 句柄和从句柄中取得文件名的GetModuleFileName。下面是利用了 Win32 API 来取得全路径的程序。

```
#include <windows.h>
#include <stdio.h>
#include <assert.h>

int main() {
    int ret;
    HMODULE h;
    char fullpath[1024];

    h = GetModuleHandle(NULL);
    assert(h);
    ret = GetModuleFileName(h, fullpath, 1024);
    assert(ret != 0);

    printf("%s\n", fullpath);

    return 0;
}
```

利用 getexecname(3C)

在 Solaris 中存在着为了得到运行进程的 path 的函数getexecname(3C)。这样在 OS 取得了 path 的环境中可以轻松地查询 path。

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

int main() {
    const char *fullpath = getexecname();
    printf("%s\n", fullpath);

    return 0;
}
```

利用动态加载信息

有很多在动态加载信息中包含了运行进程自身信息的环境，也可以从它们中取得信息。关于取得动态加载信息的方法在下一个 Hack 中有介绍。该方法在动态加载信息时很重要，并且在其他不能轻松地取得路径的环境中非常有效。

总结

介绍了许多查询运行进程的路径的方法。这个方法没有什么秘诀，要根据程序的要求选择方法。这些方法有以下几个要点。

- 从argv[0]和 $PATH 中查询的方法，虽然在安装和运行时有一定开销，但在很多情况中却确实可行。
- 使用了 procfs 的方法，虽然是很容易，但 OS 依存性却很高。
- 使用了 Win32 API 和 getexecname(3C)的方法，在有 OS 协助的环境中可将 API 的运用简单化。
- 通过下一个 Hack 中介绍的取得动态加载信息的方法，能更多地获得路径。

—— *Shinichiro Hamaji*

检测正在加载的共享库

在本 Hack 中介绍了检测运行中正在加载的共享库的方法。这个 Hack 会根据目标的不同而进行适当的选择。

在“**[Hack #64]**检测运行中进程的路径名”中介绍了检测运行文件的路径的方法，但带有运行时的信息并不是只有可执行文件。在很多情况中，有必要知道共享库的路径。在本 Hack 中介绍了发现运行中被加载的共享库的路径和加载地址的方法。关于静态检查共享库的方法请参照“**[Hack #7]**用 ldd 检测共享库的依存关系”。

加载地址

像“**[Hack #6]**静态库和共享库”中所介绍的那样，共享库在运行前将会被 loader（加载器）用 memory map 来加载，这之前的符号的地址不能确定。为了指定被加载的共享库的符号地址，要用被加载后的位置来调整，这个被加载后的位置叫做加载地址。

在Linux中，可简单地检测查询用`procfs`加载的共享库和加载地址。例如：将下面的命令运行的话，

```
% cat /proc/self/maps
```

将会得到下面的输出。

```
08048000-0804d000 r-xp 00000000 03:06 193783   /bin/cat
0804d000-0804e000 rw-p 00004000 03:06 193783   /bin/cat
0804e000-0806f000 rw-p 0804e000 00:00 0 [heap]
41000000-4101a000 r-xp 00000000 03:06 345981   /lib/ld-2.3.5.so
4101a000-4101b000 r--p 00019000 03:06 345981   /lib/ld-2.3.5.so
4101b000-4101c000 rw-p 0001a000 03:06 345981   /lib/ld-2.3.5.so
4101e000-41141000 r-xp 00000000 03:06 345983   /lib/libc-2.3.5.so
41141000-41143000 r--p 00123000 03:06 345983   /lib/libc-2.3.5.so
41143000-41145000 rw-p 00125000 03:06 345983   /lib/libc-2.3.5.so
41145000-41147000 rw-p 41145000 00:00 0
b7cb8000-b7d92000 r--p 0205b000 03:06 261739   /usr/lib/locale/
locale-archive
b7d92000-b7f92000 r--p 00000000 03:06 261739   /usr/lib/locale/
locale-archive
b7f92000-b7f93000 rw-p b7f92000 00:00 0
b7fb0000-b7fb1000 rw-p b7fb0000 00:00 0
bfd9a000-bfdb1000 rw-p bfd9a000 00:00 0        [stack]
ffffe000-fffff000 ---p 00000000 00:00 0        [vdso]
```

虽在procfs中可通过检测/proc/<process_id>/*来查询进程信息。但像在“[Hack #64]检测运行中进程的路径名”中所介绍了的那样，在Linux环境中/proc/self将会变成面向运行进程ID的symoblic link。也就是说此时将会输出为输出而运行cat命令自身的信息。

从这个表的左边开始，分别为：memory map的开始位置——终止位置、内存保护属性、文件内的offset、文件设备的major号码：minor号码、文件的i节点、文件名。本Hack的目的是想动态取得memory map的开始位置和文件名。

在OS用procfs来提供这个信息时，perspective文件也是实现Hack的方法的一种。像Linux那样只要有文本形式便可运行perspective文件，sys/procfs.h这样的头文件也会很有用，但要注意由prelinker产生的加载地址有时可能会无效。

在Linux上利用dl_iterate_phdr(3)

在Linux的glibc内，为了实现这个Hack目标，存在有函数dl_iterate_phdr(3)。这个函数为Linux所固有，是Linux环境中方便的解决方法之一，和“[Hack #64]检索运行中进程的路径名”一样，在目标的OS提供了对Hack有用的方法时，使用这个方法最简单。

dl_iterate_phdr通过callback来检索库，下面介绍的是sample代码。

```
#define _GNU_SOURCE
#include <stdio.h>
```

```
#include <link.h>

static int print_callback(struct dl_phdr_info *info, size_t size,
void *data) {
    printf("%08x %s\n", info->dlpi_addr, info->dlpi_name);
    return 0;
}

int main() {
  dl_iterate_phdr(print_callback, NULL);
  return 0;
}
```

这个sample只在标准输出上输出加载地址和库文件名。callback函数的data变量将传送dl_iterate_phdr的第2变量，在想要将context的影响施加到callback的处理上时可利用。另外，若callback函数中返还到0以外的话，则会在途中终止callback。下面表示的为Fedora Core 4中的sample代码的运行结果。

```
% ./a.out
00000000
00000000
00000000 /lib/libc.so.6
00000000 /lib/ld-linux.so.2
```

虽然表示出了被动态加载的库的名称，但有2个疑虑。

第一，最初的2行的文件名不能被表示出的部分究竟是什么？第一行是该进程的自身信息。在dl_iterate_phdr中，无论如何也不能进行像“**[Hack #64]** 检索运行中进程的路径名那样的操作。第2行为内核用linux-gate.so.1虚拟共享库提供的东西，所以不能表示出文件名。刚刚/proc/self/maps的内容中的[vdso]这一空间对应于此。关于这个文件的解说请参照**[Hack #59]**。

第二，为什么正被加载的共享库的加载地址两个都会变为0？参照先前的/proc/self/maps，若不为0x41000000附近的值则会很奇怪。这是因为在此环境中运行了prelink，并在prelink符号的值上附加了加载地址部分的offset。关于prelink的详细信息，请参照“**[Hack #85]** 用prelink将程序的启动高速化”。

下面表示的是在非prelink使用环境的Debian GNU/Linux中的运行结果的例子。

```
% ./a.out
00000000
```

```
00000000
4001d000 /lib/tls/libc.so.6
40000000 /lib/ld-linux.so.2
```

利用 dlinfo(3)

在 FreeBSD 和 Solaris 中，可通过综合使用 dlopen 函数和 RTLD_SELF RTLD_DI_LINKMAPflag来实现这个 Hack。dlinfo(3)是用dlopen打开的用户名和通过指定已取得的信息来获取信息的函数，可通过使用 RTLD_SELF来指定自身的用户名。另外，要求通过指定 RTLD_DI_LINKMAP 来返回共享库的 list。下面为样本代码。

```
#include <stdio.h>
#include <dlfcn.h>
#include <link.h>
#include <assert.h>

int main() {
    struct link_map *lmap;
    int ret = dlinfo(RTLD_SELF, RTLD_DI_LINKMAP, &lmap);
    assert(ret == 0);
    while (lmap) {
        printf("%08x %s\n", lmap->l_addr, lmap->l_name);
        lmap = lmap->l_next;
    }
    return 0;
}
```

以下为在 FreeBSD 中的运行范例。

```
% ./a.out
08048000 ./a.out
28067000 /usr/lib/libc.so.4
28049000 /usr/libexec/ld-elf.so.1
```

以下为在 Solaris 中的运行范例。

```
% ./a.out
00010000 a.out
ff3fa000 /usr/lib/libdl.so.1
ff280000 /usr/lib/libc.so.1
ff3a0000 /usr/platform/SUNW,Sun-Fire-V210/lib/libc_psr.so.1
```

在dlfcn.h中,还有可借助实现该Hack的函数,记录在NetBSD和OpenBSD的头内的dlctl(3)可以实现这个 Hack。但遗憾的是笔者所确认过的版本还未安装，故记录在了头上。

另外，只要明白了共享库名和函数名其中一个之后，便可以通过 dlopen、dlsym、dladdr 的合成来查询加载地址。

利用 Win32 API

Windows 也可以通过 Win32 API 来提供实现这个 Hack 的方法，通过在 tlhelp32.h 里的 CreateToolhelp32Snapshot 取得进程的截图，然后用 Module32First 和 Module32Next 返回 module 用户名。下面表示的为样本代码。

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <tlhelp32.h>

void print_module(MODULEENTRY32 *me) {
    char buf[1024];
    int ret = GetModuleFileName(me->hModule, buf, 1024);
    assert(ret);
    printf("%08x %s\n", (int)me->modBaseAddr, buf);
}

int main() {
    HANDLE ss;
    MODULEENTRY32 me;
    int ret;

    ss = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, 0);
    assert(ss);
    me.dwSize = 1024;
    ret = Module32First(ss, &me);
    assert(ret);

    print_module(&me);
    for (;;) {
        me.dwSize = 1024;
        if (!Module32Next(ss, &me)) {
            break;
        }
    print_module(&me);
}

    return 0;
}
```

以下表示的为运行结果的例子。

```
% ./a.exe
00400000 D:\wrk\binhack\a.exe
7f7b0000 C:\windows\system\msvcrt.dll
7fd20000 c:\windows\system\kernel32.dll
7ffa0000 c:\windows\system\ntdll.dll
```

由于在第一行中可得到自身的全路径，所以可作为代替“**[Hack #64]**检索运行中进程的路径名”的方法来使用。

在 Mac OS X 上利用 dyld(3)

dyld(3)在Mac OS X中的利用很简单。实际上并不存在函数dyld(3)，为了在_dyld preface上利用开始函数群，在man 3 dyld上发现这个目录，故标记为这样。

下面所示的为样本代码，通过_dyld_image_count取得共享库的数量，数量表示多少则多少次读出_dyld_get_image_name和_dyld_get_image_vmaddr_slide。

```
#include <mach-o/dyld.h>

int main() {
    int num, i;
    num = _dyld_image_count();
    for (i = 0; i < num; i++) {
        printf("%08x %s\n",
                _dyld_get_image_vmaddr_slide(i),
_dyld_get_image_name(i));
    }
    return 0;
}
```

以下表示的是运行结果。

```
% ./a.out
00000000 ./a.out
00000000 /usr/lib/libSystem.B.dylib
00000000 /usr/lib/system/libmathCommon.A.dylib
```

Mac OS X根据prebinding的结构，执行与prelink相同的运行。因此，所有的加载地址都变为0。

另外，在第一行中表示出来的./a.out和argv[0]等值。因此，不能作为“**[Hack #64]**检查运行中进程的路径名”这个方法来发挥作用。

总结

介绍了检索正在运行中进程的加载共享库的方法。这个Hack的实现方法会因环境的不同而不同，切勿混淆。要根据目标来适当选择。

—— *Shinichiro Hamaji*

HACK #66 掌握 process 和动态库 map memory

使用pmap命令和/proc/<pid>/maps文件，可确认如何使用各程序的memory cheap.stack等。

启动程序后，它的运行二进制文件和由动态加载器加载的共享库文件，将作为该进程的虚拟内存空间的一部分被映射。在[Hack #65]检索正在加载的共享库中介绍了将映射什么样的文件。在本Hack中，对进程使用的虚拟内存的范围将如何变化进行了说明。

使用 pmap 命令

Linux的procps包和Solaris的标准命令中都含有pmap命令。通过使用这个命令，进程可表示为使用中的虚拟memory map的状态，其使用方法会在pmap命令的变量上指定进程ID。另外，在Linux中，pmap命令实际上会从/proc/<pid>/maps文件中取得mapping信息，然后将其变换为用户易懂的形式表示出来，因此，直接表示出/proc/<pid>/maps文件也可获得同样的信息。

作为例子，将在运行通过指定停止时间终止/bin/sleep命令过程中，在i386架构的Linux 2.6.15内核（Debian/GNU/Linux sarge/bin/sleep）上进行了pmap的状态表示出来。

```
% /bin/sleep 10000 &
[1] 24039
% ps auxw | grep /bin/sleep
gotom 24039 0.0 0.0 3892 596 pts/13 S 10:48 0:00 /bin/sleep 10000
% pmap 24039
24039:   /bin/sleep 10000
08048000     16K r-x--  /bin/sleep
0804c000      4K rw---  /bin/sleep
0804d000    132K rw---    [ anon ]
b7b69000   2048K r----  /usr/lib/locale/locale-archive
b7d69000      8K rw---  [ anon ]
b7d6b000     60K r-x--  /lib/tls/i686/cmov/libpthread-2.3.5.so
b7d7a000      8K rw---  /lib/tls/i686/cmov/libpthread-2.3.5.so
b7d7c000      8K rw---    [ anon ]
b7d7e000   1220K r-x--  /lib/tls/i686/cmov/libc-2.3.5.so
b7eaf000      4K r----  /lib/tls/i686/cmov/libc-2.3.5.so
b7eb0000     12K rw---  /lib/tls/i686/cmov/libc-2.3.5.so
b7eb3000      8K rw---    [ anon ]
b7eb5000     28K r-x--  /lib/tls/i686/cmov/librt-2.3.5.so
b7ebc000      8K rw---  /lib/tls/i686/cmov/librt-2.3.5.so
b7ebe000    140K r-x--  /lib/tls/i686/cmov/libm-2.3.5.so
```

```
b7ee1000      8K rw---  /lib/tls/i686/cmov/libm-2.3.5.so
b7f00000      4K rw---    [ anon ]
b7f01000     84K r-x--  /lib/ld-2.3.5.so
b7f16000      8K rw---  /lib/ld-2.3.5.so
bff01000     84K rw---    [ stack ]
ffffe000      4K -----    [ anon ]
total      3896K
```

通过使用 pmap 命令的详细信息的选项，可得到更详细的信息。

从 pmap 命令输出的第 1 列和第 2 列中，可以知道使用了该进程的虚拟内存中的哪个范围。若向该范围外进行memory访问的话，则会发出segmentation fault等信号。在第3列中可以知道面向该范围的访问许可。和文件的许可相同，变为 r 后则读取，变为 w 后则读入，变为 x 则运行一一予以表示。在第4列中可以知道该虚拟memory的使用目的。那么，来仔细看看这些信息吧。

首先是关于运行文件的 /bin/sleep，它在输出的第 2 列和第 3 列中。变为 r-x 的则以 16KB，变为 rw- 的则以 4KB 的形式存在于用 readelf -S 表示的和地址一致的范围中。

```
% readelf -S /bin/sleep
There are 24 section headers, starting at offset 0x3498:

Section Headers:
  [Nr] Name              Type        Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL        00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS    08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE        08048128 000128 000020 00   A  0   0  4
  [ 3] .hash             HASH        08048148 000148 000148 04   A  4   0  4
  [ 4] .dynsym           DYNSYM      08048290 000290 0002b0 10   A  5   1  4
  [ 5] .dynstr           STRTAB      08048540 000540 0001f5 00   A  0   0  1
  [ 6] .gnu.version      VERSYM      08048736 000736 000056 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED     0804878c 00078c 000080 00   A  5   2  4
  [ 8] .rel.dyn          REL         0804880c 00080c 000028 08   A  4   0  4
  [ 9] .rel.plt          REL         08048834 000834 000118 08   A  4  11  4
  [10] .init             PROGBITS    0804894c 00094c 000017 00  AX  0   0  4
  [11] .plt              PROGBITS    08048964 000964 000240 04  AX  0   0  4
  [12] .text             PROGBITS    08048bb0 000bb0 001cf0 00  AX  0   0 16
  [13] .fini             PROGBITS    0804a8a0 0028a0 00001b 00  AX  0   0  4
  [14] .rodata           PROGBITS    0804a8c0 0028c0 000958 00   A  0   0 32
  [15] .data             PROGBITS    0804c218 003218 000024 00  WA  0   0  4
  [16] .eh_frame         PROGBITS    0804c23c 00323c 000004 00   A  0   0  4
  [17] .dynamic          DYNAMIC     0804c240 003240 0000d8 08  WA  5   0  4
  [18] .ctors            PROGBITS    0804c318 003318 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS    0804c320 003320 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS    0804c328 003328 000004 00  WA  0   0  4
  [21] .got              PROGBITS    0804c32c 00332c 0000a0 04  WA  0   0  4
  [22] .bss              NOBITS      0804c3e0 0033e0 00014c 00  WA  0   0 32
  [23] .shstrtab         STRTAB      00000000 0033e0 0000b5 00      0   0  1
```

从以上的表示中我们可以知道，在上述/bin/sleep程序的内部，节interp...rodata分配到了0x08048000-0x0804bfff的地址范围中。section.interp~.wdate被分割。在这些区域不能读入但是可以运行。另外，在0x0804c000-0x0804cfff的地址范围中，存在着能读入却不能运行的节data~.bss。.plt、.got、.text、.rodata、.data存在于某个区域中。

另外，由于这是x86的例子，区域的页面大小均为4KB。在架构变化了时，这些地址范围也会发生变化。可以发现作为其他文件名的动态链接器ld-2.3.5.so和共享库libc-2.3.5.so。这些文件当然都是共享库，因此即使是用readelf -S发现了，也并不是像上述那样设定的固定的地址。但在每个区域上都设定地址范围的这一点却和/bin/sleep的情况相同。

用 pmap 命令确认程序保存的内存

接下来，为了详细了解如何使用程序保存的anon和stack这些区域，来看看运行了执行内存获得的样本程序。以下为样本程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define SLEEP 7

void do_malloc(size_t size)
{
  void *mem;
  mem = malloc(size * 1024);
  if (mem == NULL) {
    printf("memory exhausted\n");
    exit(1);
  }
  printf("%u KB allocated\n", size);
  sleep(SLEEP);
}

#define STACKSIZE (1024 * 1024)
void do_stack(void)
{
  int stack[STACKSIZE];
  stack[STACKSIZE - 1] = 0;
  printf("stack 1MB allocated\n");
  sleep(SLEEP);
}

int main(void)
{
```

```
    printf("process %u started\n", getpid());
    sleep(SLEEP);
    do_malloc(8);     /* 确保8KB*/
    do_malloc(100);   /* 确保100KB */
    do_malloc(100);   /* 确保100KB */
    do_malloc(1024);  /* 确保1MB */
    do_stack();       /* 将stack 变大*/
    exit(0);
}
```

这个程序在每次启动后7秒时，在停止的同时会运行各种size的malloc()。然后在运行中横向运行pmap命令并观察哪个区域发生了变化。

```
% ./malloc
process 30939 started
8 KB allocated
100 KB allocated
100 KB allocated
1024 KB allocated
stack 1MB allocated
```

此时，运行strace则会表示为如下这样（由于较长所以适当省略）。

```
% strace ./malloc
execve("./malloc", ["./malloc"], [/* 31 vars */]) = 0
uname({sys="Linux", node="celesta", ...}) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7fbe000
open("/etc/ld.so.cache", O_RDONLY)      = 3
write(1, "process 30956 started\n", 22process 30956 started ) = 22
nanosleep({7, 0}, {7, 0})               = 0
brk(0)                                  = 0x804a000
brk(0x806d000)                          = 0x806d000
write(1, "8 KB allocated\n", 15)        = 15
nanosleep({7, 0}, {7, 0})               = 0
write(1, "100 KB allocated\n", 17)      = 17
nanosleep({7, 0}, {7, 0})               = 0
brk(0x809f000) = 0x809f000
write(1, "100 KB allocated\n", 17)      = 17
nanosleep({7, 0}, {7, 0})               = 0
mmap2(NULL, 1052672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7d68000
write(1, "1024 KB allocated\n", 18)     = 18
nanosleep({7, 0}, {7, 0})               = 0
write(1, "stack 1MB allocated\n", 20)   = 20
nanosleep({7, 0}, {7, 0})               = 0
munmap(0xb7fbd000, 4096)                = 0
```

来看看操作中pmap命令的输出会怎样变化。由于页面有限所以只能看到有变化的入口。首先，获得了8KB之后，会更新出现以下的入口，这是通过在malloc()的延长线上读出brk() heap领域。

```
0804a000    140K rw---    [ anon ]
b7dd3000      4K rw---    [ anon ]
```

接下来，2次获得100KB的话则会变为如下这样。这是由于heap的不足而被brk()系统调用扩展的领域。

```
0804a000    340K rw---    [ anon ]
b7dd3000      4K rw---    [ anon ]
```

再次获得1MB后，会变为如下这样。不是扩大heap，而是增加从0xb7cd2000开始的地址，这是由于glibc的malloc allocater通过anonymous mmap取得了内存区域。

```
0804a000    340K rw---    [ anon ]
b7cd2000   1032K rw---    [ anon ]
```

这是获得了stack之后，会变为如下这样。

```
bf73c000   4104K rw---    [ stack ]
```

stack区域增大了。

总结

介绍了通过使用pmap命令和/proc/<pid>/maps文件，可确认将怎样使用memory（heap、stack等）。这个Hack也可考虑运用到共享库中。

—— *Masanori Goto*

HACK #67 用libbfd取得符号的一览表

在本Hack中，对libbfd取得符号的名称和地址的一览表的方法以及从地址中取得文件名和行信息的方法进行了介绍。

GNU binutils的大部分都作为库的interface命令被实现。BFD是Binary File Descriptor的省略，正如其名，它便是用于读写二进制文件的库。

在本Hack中，简单介绍了libbfd使用法的一个例子，介绍了为了实现在native二进制中的reflection所必须的取得符号的名称和地址的一览表的方法。

libbfd 的概要

如前所述，libbfd是作为GNU binutils的一部分而开发出的库，在GPL上被分配。

libbfd在Fedora Core 4 中作为binutils package的一部分，在Debian GNU/Linux 中包含在 binutils-dev package 里。即便是没被当作 package 储存的环境中，在自动编译 binutils 并安装后也可利用。

libbfd 可以和 GCC 同时使用，所以非常多的二进制格式以及架构都支持它。在大多数情况下，会在libbfd内部吸收由这个格式和架构产生的差异，使用者可用一个代码来书写能在多种环境中运作的软件。

libbfd 的文献在 *http://www.sra.co.jp/wingnut/bfd/bfd-ja.html* 中有相关的翻译。另外，在使用方法不明时，binutils 的同一处理部分为参考是不错的，例如：这个 Hack 的内容以 binutils 内的 nm(1) 为参考。

利用 minisymbol 系的 API 取得符号一览表

libbfd 中有 2 个取得符号一览表的 API 。首先，将介绍没有符号表的省 memory 的 minisymbolAPI。下面为样本代码。

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

void dump_symbols(const char *filename) {
    bfd *abfd;
    asymbol *store;
    char *p;
    void *minisyms;
    int symnum, i;
    size_t size;
    /* 取得动态符号时为 1 */
    int dyn = 0;
    int ret;

    abfd = bfd_openr(filename, NULL);
    assert(abfd);
    ret = bfd_check_format(abfd, bfd_object);
    assert(ret);

    if (!(bfd_get_file_flags(abfd) & HAS_SYMS)) {
        assert(bfd_get_error() == bfd_error_no_error);
        /* there are no symbols */
```

```
        bfd_close(abfd);
        return;
    }

    store = bfd_make_empty_symbol(abfd);

    symnum = bfd_read_minisymbols(abfd, dyn, &minisyms, &size);
    assert(symnum >= 0);

    p = (char *)minisyms;
    for (i = 0; i < symnum; i++) {
        asymbol *sym = bfd_minisymbol_to_symbol(abfd, dyn, p, store);
        const char *name = bfd_asymbol_name(sym);
        int value = bfd_asymbol_value(sym);
        printf("%08x %s\n", value, name);
        p += size;
    }

    free(minisyms);
    bfd_close(abfd);
}

int main(int argc, char *argv[]) {
    /* 与“[Hack #64]检索运行中进程的路径名”相配合 */
    dump_symbols(argv[0]);

    return 0;
}
```

像在程序内的说明中介绍的那样，在要得到与nm -D相当的动态符号时，在dyn上设置1，这在从strip过后共享库中搜索符号时相当有效。

利用symtab系的API

也有使用bfd_get_symtab_upper_bound和bfd_canonicalize_symtab函数的方法，用这个API可以迅速取得asymbol构造体的排列。由于在同一程序中不怎么有趣，所以这次试着取得符号一览表以及文件名和行数。以下为表示符号一览表以及被定义的文件名和行数的程序。

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

void dump_symbols(const char *filename) {
    bfd *abfd;
    long storage;
    asymbol **syms;
    int symnum;
    int i;
    int ret;
```

```
    abfd = bfd_openr(filename, NULL);
    assert(abfd);
    ret = bfd_check_format(abfd, bfd_object);
    assert(ret);

    if (!(bfd_get_file_flags(abfd) & HAS_SYMS)) {
        assert(bfd_get_error() == bfd_error_no_error);
        /* there are no symbols */
        bfd_close(abfd);
        return;
    }

    storage = bfd_get_symtab_upper_bound(abfd);
    assert(storage >= 0);
    if (storage) syms = (asymbol **)malloc(storage);

    symnum = bfd_canonicalize_symtab(abfd, syms);

    assert(symnum >= 0);

    for (i = 0; i < symnum; i++) {
        asymbol *sym = syms[i];
        int value = bfd_asymbol_value(sym);
        const char *file, *name;
        int lineno;
        asection *dbgsec = bfd_get_section_by_name(abfd, ".debug_info");

        ret = bfd_find_nearest_line(abfd, dbgsec, syms, value,
                                    &file, &name, &lineno);

        if (ret && file && name) {
            printf("%08x %s (%s:%d)\n", value, name, file, lineno);
        }
        else {
            name = bfd_asymbol_name(sym);
            printf("%08x %s\n", value, name);
        }
    }

    free(syms);
    bfd_close(abfd);
}

int main(int argc, char *argv[]) {
    /* 与“[Hack #64]检索运行中进程的路径名”相配合的better*/
    dump_symbols(argv[0]);

    return 0;
}
```

用bfd_find_nearest_line取得了文件名和行信息。这个信息存在于debug_info节中，所以用bfd_get_section_by_name取得了节构造体。这与“**[Hack #15]**用 addr2line 从地址中取得文件名和行号码”中介绍了的addr2line(1)一样。

不能取得动态符号时，将bfd_get_symtab_upper_bound和bfd_canonicalize_symtab变更为bfd_get_dynamic_symtab_upper_bound和bfd_canonicalize_dynamic_symtab后就可以了。

总结

在本Hack中，介绍了libbfd取得符号的名称和地址的一览表的方法以及从地址中取得文件名和行信息的方法，libbfd是在读写二进制文件时非常便利的库，在想要运行时，运行与GNU binutils相当的处理时非常有效。

—— *Shinichiro Hamaji*

运行C++语言时进行demangle

在本Hack中介绍了在执行C++语言中demangle的方法：cplus_demangle()、abi::__cxa_demangle()。

为使符号拥有单独意义的名称，C++编译器会进行叫做name mangling的处理。本Hack中则介绍了C++语言运行demangle的方法。有关通过命令执行demangle的方法，请参照“通过**[Hack #14]**中c++filt demangle C++语言”。

mangle的方法依赖于编译器。有时即使是相同的编译器，版本不同，mangle的方法也不同。例如：在GCC 3.x中，int foo(int)mangle为_Z3fooi，int foo (const char*)mangle为_Z3fooPKc，但在GCC 2.95中分别mangle为foo__Fi和foo__FPCc。

运行时进行demangle

运行时mangle在C++程序中进行mangle有两种方法。一种是使用binutils的libiberty所包含的cplus_demangle()，另一种是使用包含在libstdc++中的abi::__cxa_demangle()，如在能使用后者的环境中，则使用后者为好。

```
#include <iostream>
#include <typeinfo>
#include <cxxabi.h>
using namespace std;

// From binutils/include/demangle.h
#define DMGL_PARAMS      (1 << 0) /* Include function args */
#define DMGL_ANSI        (1 << 1) /* Include const, volatile, etc */
```

```
#define DMGL_VERBOSE     (1 << 3) /* Include implementation details. */
#define DMGL_TYPES       (1 << 4) /* Also try to demangle type
encodings. */
extern "C" char *cplus_demangle(const char *mangled, int options);

int main() {
    // using libiberty
    int options = DMGL_PARAMS | DMGL_ANSI | DMGL_TYPES;
    cout << cplus_demangle("_Z3fooPKc", options) << endl;
 cout << cplus_demangle("i", options) << endl;

    // using libstdc++
    int status;
    cout << abi::__cxa_demangle("_Z3fooPKc", 0, 0, &status) << endl;
    cout << abi::__cxa_demangle("i", 0, 0, &status) << endl;
    return 0;
}
```

运行结果如下。

```
% g++ test.cpp -liberty && ./a.out
foo(char const*)
int
foo(char const*)
int
```

cplus_demangle()虽已包含在libiberty.a中，但由于没有被libiberty.h声明出来，所以必须从binutils/include/demangle.h中复制必要的声明才能使用。同时，需要链接libiberty.a。另外，cplus_demangle()为返回已 malloc 的memory，在上面的代码中都会含有memory leak。

abi::__cxa_demangle()中如果包含cxxabi.h即可使用。但诸如GCC 2.95，在旧的GCC中并不存在，这就是难点。abi::__cxa_demangle()的代码可以从binutils放入GCC，所以可以认为处理的重点大体相同。此外，abi::__cxa_demangle(mangled, 0, 0, &status)读出的时候要通过已malloc 的内存返回结果，以上的代码中包含了memory leak。

typeid与匹配

在C++可利用typeid运算符得知运行时的类型信息。读出对应typeid()所返回的type_info对象（的参照）的name()成员函数，就可以通过字符串得到类型名。

在GCC（至少为3.3）中，由于类型名已mangle，需要易懂的形式则要demangle。

```
#include <iostream>
#include <typeinfo>
#include <cxxabi.h>
using namespace std;
struct Foo {
    virtual ~Foo() {};
};
struct Bar : public Foo {
    virtual ~Bar() {};
};

// 内存泄漏
char* demangle(const char *demangle) {
    int status;
    return abi::__cxa_demangle(demangle, 0, 0, &status);
}

int main() {
    Bar bar;
    Foo *p = &bar;
    cout << typeid(int).name() << endl;
    cout << typeid(Foo).name() << endl;
    cout << typeid(p).name() << endl;
    cout << typeid(*p).name() << endl;
    cout << endl;

    cout << demangle(typeid(int).name()) << endl;
    cout << demangle(typeid(Foo).name()) << endl;
    cout << demangle(typeid(p).name()) << endl;
    cout << demangle(typeid(*p).name()) << endl;
    return 0;
}
```

运行结果如下。

```
% g++ test.cpp && ./a.out
i
3Foo
P3Foo
3Bar

int
Foo
Foo*
Bar
```

Foo 和 Bar 附有虚拟 destructor，虚拟函数如果一个都不存在的话，typeid(*p).name()就不是 "Bar" ，因为已返回了 "Foo"。

总结

本 Hack 介绍了作为运用 C++ 语言符号时的 demangle 方法 cplus_

demangle()、abi::__cxa_demangle()。掌握了这些，被mangle的符号也没有可怕的了。

—— *Satoru Takabayashi*

用 ffcall 动态决定签名，读出函数

利用ffcall和libffi，运行时利用signature签名以便指定签名，便可执行函数。

考虑使用函数地址读出函数。编译时，如果签名已经确定，则只要cast后执行即可。但是，如果是已生成执行包含丰富动态信息的native binary的程序语言，也有在运行时决定函数签名的。在本Hack中将介绍利用ffcall的库实现此操作的方法。

ffcall

ffcall是Foreign Function CALL的简称。ffcall通过GPL2发布，撰写时的版本是1.10。ffcall是GNUstep（汇集了Objective-C用的系统的基本类库）所利用的库。Objective-C持有丰富的动态信息，同时是通过native binary运行的程序语言，所以为实现如映射这样的结构要用到这样的库便不足为奇了。

在此虽没有相关陈述，但在ffcall中已包含了实现trampoline的API。有关trampoline请参照“**[Hack #32]**依靠GCC所生成的代码运行时代码生成”。先来看个简单的例子。

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

/* 返回文字列中第 n 列文字的函数 */
char nth(const char *str, int i) {
    return str[i-1];
}

int main() {
    int n = 5;
    const char *msg = "binary";
    char ret;
    av_alist alist;

    /* 通常的调用 */
    printf("NORMAL: %c\n", nth(msg, n));
```

```
    /* 利用ffcall的调用*/
    av_start_char(alist, &nth, &ret);
    av_ptr(alist, const char *, msg);
    av_int(alist, n);
    av_call(alist);

    printf("FFCALL: %c\n", ret);

    return 0;
}
```

nth是返回字符串中第n列文字的函数。首先用一般的函数读出它，再利用ffcall读出函数。通过最开始的av-start-char在alist里设置char返回值的函数nth和该返回值的保存位置ret。通过其次的av.ptr.av-int指定类型同时设定参数，通过av_call读出函数。而且结果应该会进入ret，所以输出该结果，运行结果如下所示。

```
% ./a.out
NORMAL: r
FFCALL: r
```

这下完全明白一般读出可得到同样的结果。

运行时签名的变更

仅此而已并不有趣，所以试着在实际执行时指定签名。以下的例子会根据命令行参数指定并运行函数签名。第1参数为函数名，以下的则用了s或i指定字符串或整数后，把内容放入之后的参数里。在此并不支持其他的类型。

```
#include <avcall.h>
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char *argv[]) {
    int ret;
    av_alist alist;
    void *dlh;
    void *fp;
    int i;

    if (argc < 2) return 1;

    dlh = dlopen(argv[0], RTLD_LAZY);
    fp = dlsym(dlh, argv[1]);

    av_start_int(alist, fp, &ret);

    for (i = 2; i < argc; i += 2) {
        /* string */
        if (argv[i][0] == 's') {
```

```
            av_ptr(alist, char *, argv[i+1]);
        }
        /* int */
        else if (argv[i][0] == 'i') {
            av_int(alist, atoi(argv[i+1]));
        }
    }

    av_call(alist);

    printf("\nRESULT: %d\n", ret);

    return 0;
}
```

在这个例子中，argv[0]假定为自身的名称并打开它，通过dlsym从用argv[1]指定了的函数名中引出地址。真正运行这些的方法到此Hack里所述为止。接着，试着进行各种实验。

```
% ./a.out puts s "hello world"
hello world
RESULT: 12
```

首先是hello world已经正确表示出来了。结果12为puts的输出字符数。其次也试用一下 int。

```
% ./a.out printf s %d i 7
7
RESULT: 1
```

已经读出printf("%d", 7)，这表面也是正确的结果出来了。最后，再试着增加一些参数。

```
% ./a.out printf s %d+%d=%s i 3 i 4 s SEVEN
3+4=SEVEN
RESULT: 9
```

虽增加了参数，但这个例程也可以很好地运行。

libffi

实现这样功能的库并不只ffcall。同样持有动态系统、运行native binary的语言环境，依靠通过GCJ（GCC的Java binding）使用的libffi，运行时指定签名，即可运行函数。ffi是Foreign Function Interface的缩写，通过MIT许可分发、安装，安装作为GCC的一部分，装在GCJ的源代码里。

由于libffi和ffcall非常类似，在此只用ffcall把libffi最开始的例子安装一下。

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

char nth(const char *str, int i) {
    return str[i-1];
}

int main() {
    int n = 5;
    const char *msg = "binary";

    ffi_cif cif;
    int arg_num;
    ffi_type *arg_types[2];
    void *arg_values[2];
    ffi_arg ret;
    ffi_type *ret_type;

    /* 通常的读出 */
    printf("NORMAL: %c\n", nth(msg, n));

    /* 使用libffi的读出 */
    ret_type = &ffi_type_schar;
    arg_num = 2;
    arg_types[0] = &ffi_type_pointer;
    arg_values[0] = &msg;
    arg_types[1] = &ffi_type_sint;
    arg_values[1] = &n;

    ffi_prep_cif(&cif, FFI_DEFAULT_ABI, arg_num, ret_type, arg_types);
    ffi_call(&cif, FFI_FN(nth), &ret, arg_values);

    printf("LIBFFI: %c\n", ret);

    return 0;
}
```

在libffi中，虽有几个参数，但用ffi_prep_cif准备与用ffi_call读出，只能使用这两个API。作为替代需在排列上准备好参数的类型信息。

和其他Hack的搭配

试与“**[Hack #68]**运行C++语言时进行demangle”相配合。比如在C++中，由于在已mangling的符号中存在参数的信息，这些信息如果能够好好利用，签名的指定便变得轻松了。这也就说存在务须通过s和i指定参数型的可能性。遗憾的是，GCC的C++ mangle规则中，返回值的类型并不包含在符号里。要想确实地得到这些信息，请参照“**[Hack #70]**用libdwarf取得调试信息”。

总结

利用 ffcall 和 libffi，在运行时指定签名便可能运行函数。本 Hack 主要是在语言环境中发挥作用。

—— *Shinichiro Hamaji*

用 libdwarf 取得调试信息

在 Hack 中，介绍 GCC 处理环境中所使用的调试信息 DWARF2 的库 libdwarf 。

DWARF2

DWARF2 把调试信息保存在目标文件格式化。正如“**[Hack #8]**用 readelf 表示 ELF 文件信息”所解说的那样，DWARF2 格式化的信息可通过 readelf 的 -w 选项来查看。

在 DWARF2 中，各个不同种类的调试信息保存在以 debug 为开头的复数节里。这里为包含类型信息、函数以及变量的类型、名称、运行信息等节，处理 debug_info 节。

这个信息是以编译单位（compile unit 在 libdwarf 中省略为 cu）为路径的树型构造、层次构造基本上与程序的层次结构相对应。

libdwarf 的安装

libdwarf 通过 *http://reality.sgiweb.org/davea/dwarf.html* 发布，LGPL 为 license，撰写时的版本是 20051201 。

libdwarf 的编译需要 libelf， 在 Debian 安装 libelfg0-dev 包和 libdwarf-dev 包。在 Fedora Core 4 中安装 elfutils-libelf-devel 之后，才能自行进行编译。

此外，在 libdwarf 中也带有 dump 称为 dwarfdump 的 DWARF2 信息的工具，可以作为替代 readelf -w 或者查阅 libdwarf 使用方法的例子代码来使用。

查阅变量名一览表的例程

以下举出例子：利用 libdwarf 来表示可执行文件所声明的变量及其行数。

```
#include <libdwarf/libdwarf.h>
#include <libdwarf/dwarf.h>
#include <libelf.h>

#include <stdio.h>
#include <fcntl.h>
#include <assert.h>

/* 显示变量名同时再次查看 DWARF2 信息 */
static void process_one_die(Dwarf_Debug dbg, Dwarf_Die die, int d) {
    Dwarf_Error err;
    int ret;

    while (1) {
        Dwarf_Half tag;
        Dwarf_Die child;

        ret = dwarf_tag(die, &tag, &err);
        assert(ret == DW_DLV_OK);

        if (tag == DW_TAG_variable ||
            tag == DW_TAG_formal_parameter)
        {
            Dwarf_Attribute attr;
            Dwarf_Unsigned line;
            char *str;

            ret = dwarf_attr(die, DW_AT_decl_line, &attr, &err);
            /* 在特殊变量的条件下，有时也不存在行信息 */
            if (ret == DW_DLV_NO_ENTRY) goto next;
            assert(ret == DW_DLV_OK);
            ret = dwarf_formudata(attr, &line, &err);
            assert(ret == DW_DLV_OK);

            ret = dwarf_attr(die, DW_AT_name, &attr, &err);
            assert(ret == DW_DLV_OK);
            ret = dwarf_formstring(attr, &str, &err);
            assert(ret == DW_DLV_OK);

            printf("%d: %s\n", (int)line, str);
        }

    next:
        ret = dwarf_child(die, &child, &err);
        assert(ret != DW_DLV_ERROR);
        if (ret == DW_DLV_OK) {
            process_one_die(dbg, child, d+1);
        }

        ret = dwarf_siblingof(dbg, die, &die, &err);
        if (ret == DW_DLV_NO_ENTRY) break;
        assert(ret == DW_DLV_OK);
    }
}

static void process_one_file(Elf *elf, const char *filename) {
    /* 取得 DWARF2 信息 */
```

```
    Dwarf_Debug dbg;
    Dwarf_Die die;
    Dwarf_Error err;
    int ret;

    Dwarf_Unsigned cu_header_length = 0;
    Dwarf_Unsigned abbrev_offset = 0;
    Dwarf_Half version_stamp = 0;
    Dwarf_Half address_size = 0;
    Dwarf_Unsigned next_cu_offset = 0;

    ret = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL, &dbg, &err);
    assert(ret == DW_DLV_OK);

    while ((ret =
            dwarf_next_cu_header(dbg, &cu_header_length, &version_stamp,
                                 &abbrev_offset, &address_size,
                                 &next_cu_offset, &err))
            == DW_DLV_OK)
    {
        ret = dwarf_siblingof(dbg, NULL, &die, &err);

        if (ret == DW_DLV_OK) {
            /* 把取得的DWARF2 信息带入下一个函数 */
            process_one_die(dbg, die, 0);
        }
        else if (ret == DW_DLV_NO_ENTRY) {
            continue;
        }
        assert(ret == DW_DLV_OK);
    }

    assert(ret != DW_DLV_ERROR);
}

void dump_variables(const char *filename) {
    /* 首先，获得ELF信息 */
    int f;
    Elf_Cmd cmd;
    Elf *elf;
    elf_version(EV_CURRENT);

    f = open(filename, O_RDONLY);
    assert(f != -1);

    cmd = ELF_C_READ;
    elf = elf_begin(f, cmd, (Elf *) 0);
    process_one_file(elf, filename);
    elf_end(elf);
}

int main(int argc, char *argv[]) {
    dump_variables(argv[0]);
    return 0;
}
```

在main中，把argv[0]带入dump_variables函数。在dump_variables函数中，利用libelf取得ELF信息。在此省略a文件的处理、64bit ELF及Cygwin的对应等。详细的内容请参照dwarfdump的代码等。

接下来在process_one_file函数中，从ELF信息中查询DWARF2信息，在这里用dwarf_next_cu_header将各编译单元。然后将得到的Dwarf_Die型变量转移到process_one_die上。die是Debug Information Entry（调试信息入口）的省略，与调试信息树的各节点相对应。

process_one_file取得die，如果那是变量的话则表示该变量名及声明行数。此外，die的同级变量，相对die的子变量再次读出process_one_die。在源代码中，next:以前的是变量名表示处理，next:以后的是追踪树的部分。next标签是代替continue使用goto next;设置的，特别是发生无问题误差的时候。

Sample的运行结果

请按如下编译sample。

```
% gcc -g dwarf.c -lelf -ldwarf
```

请注意不附加上调试信息，点击-g不会输出什么。以下显示sample的执行结果。

```
10: dbg
10: die
10: d
11: err
12: ret
15: tag
16: child
24: attr
25: line
26: str
56: elf
56: filename
59: dbg
60: die
61: err
62: ret
64: cu_header_length
65: abbrev_offset
66: version_stamp
67: address_size
68: next_cu_offset
```

```
94: filename
96: f
97: cmd
98: elf
111: argc
111: argv
```

可看到这是一个正确的变量一览表。

总结

这里介绍了处理在GCC环境等中所使用的调试信息DWARF2的库libdwarf。

—— *Shinichiro Hamaji*

通过dumper简化dump结构体的数据

dumper库是运用“[Hack#70] 通过libdwarf取得调试信息”做成的。

本Hack将介绍它的使用方法和实际安装方法，利用dumper进行printf调试和loging时，可以简单地表示较难以显示的结构体。

dumper的使用方法

dumper发布在*http://shinh.skr.jp/binary/dumper.tgz*上，许可证是LGPL。使用C++并且要有C时就能用的库，要编译的话得有libelf和libdwarf。

dumper可以非常简单地使用。基本上是先包含头，读出dump_open，再把变量带到宏p，就可以dump该变量了。下面给出个简单的例子。

```
#include "dump.h"

#include <string.h>

typedef enum { ENUM1, ENUM2 } TestEnum;
typedef union {
    int i;
    char b[4];
} TestUnion;

typedef struct TestDump_ {
    short s;
    int i;
    long l;
```

```
    unsigned long long ll;
    char c;
    char *str;
    void *ptr;
    void *const volatile *cvptr;
    struct TestDump_ *dump;
    int (*fp) (int, char*[]);
    int (*ifp) (int, char*[]);
    int array[10];
    TestEnum en;
    TestUnion un;
    struct {
    } no_name_struct;
} TestDump;

int main(int argc, char *argv[]) {
    TestDump d;
    d.s = 2;
    d.i = 3;
    d.l = 4;
    d.ll = 0xfffffffffffll;
    d.c = 'c';
    d.str = "hoge-";
    d.ptr = &d;
    d.cvptr = &d.ptr;
    d.fp = main;
    d.en = ENUM2;
    d.array[0] = 1;
    d.dump = &d;
    strcpy(d.un.b, "abc");

    dump_open(argv[0]);
    p(d);

    return 0;
}
```

通过dump.o上的目录，请照如下所示编译这个代码。dump.c是用 C++ 编写的，要用 g++ 来链接。

```
% gcc -c -g dump_sample.c
% g++ dump_sample.o dump.o -lelf -ldwarf
```

TestDump是为观察结构体中各个成员是如何被dump的，上述代码的执行结果如下：

```
d = {
  s = 2 (0x0002) : short int
  i = 3 (0x00000003) : int
  l = 4 (0x00000004) : long int
  ll = 17592186044415 (0x00000fffffffffff) : long long unsigned int
  c = 'c' (63) : char
```

```
    str = "hoge-" [0x805e328] : char*
    ptr = 0xbfd42270 : void*
    cvptr = 0xbfd42270 [0xbfd4228c] : void**
    dump = 0xbfd42270 <previously shown> : TestDump_*
    fp = int main(int, char**) [0x8049bd4] : func*
    ifp = int ???(int, char**) [0x805e2ea] : func*
    array = { 1 (0x00000001), ... } : int[10]
    en = ENUM2 : TestEnum
    un = {
      i = 6513249 (0x00636261) : int
      b = "abc\x00" [0xbfd422cc] : char[4]
    } : TestUnion
    no_name_struct = {
    } : <no name>
} [0xbfd42270] : TestDump*
```

可以看到，设置在结构体里的内容同类型信息一起被显示了出来。

dumper 的安装

dumper 是作为“**[Hack #70]**利用 libdwarf 取得调试信息”的应用而安装的。在 dump_open 中，打开指定的文件，再次返回查看 DWARF2 信息，然后查阅所有变量的名称、类型、所声明的行数，所有的类型信息，函数签名及地址。

此外，可用宏 p 进行变数的 dump。宏 p 如下所显示的：

```
# define p(v)                                          \
    do {                                               \
        typeof(v) *DUMP_TEMPVAL_NAME = &(v);           \
        dump_s(&DUMP_TEMPVAL_NAME, __STRING(v), __FILE__, __LINE__); \
} while(0)
```

首先代入刚开始时对应参数型的指针。宏的参数是任意类型，所以要用 GCC 扩展的 typeof 查看类型名。接着，在接下来的行里，把对于想 dump 的变量的参照，利用宏 __STRING 把想 dump 的变量的名称字符化后使用 __FILE__和__LINE__的行信息被带入实际的函数中。在 dump_s 中，追寻行信息，查询变量声明中的类型，进行与其类型信息相符的 dump。

其他应用

在本 Hack 中，从所遗留的调试信息可以看出，利用 C 等在运行时可以得到的类型信息。使用这样的方法，也就可用 C 实现滑动。另外，解析执行源代码，根据调试信息实际安装统一开发环境的 intelly sence 补充等，可能会比较有意思。

只是，必须带着调试信息编译，是一个大的限制。只有用来调试的才必须有调试信息 dumper 就是从这一点考虑做出来的。

另外，作为取得其他类型信息的途径，也可以被认为是在 GCC 4 以后的版本中通过跳过使用-fdump-tree-generic-raw和-fdump-tree-gimple-raw选项来取得 GENERIC 和 GIMPLE（GCC 的内部表现）来取得的方法。

—— *Shinichiro Hamaji*

#72 自行加载目标文件

在本Hack中将介绍加载目标文件的方法。利用这点就可以实现自己制作插件结构。

在运行时除了用链接选项加载外，也可以利用dlopen(3)加载共有目标文件。本Hack会介绍以libbfd得到的信息为基础自行再配置加载以.o为扩展名的目标文件的方法。有关dlopen(3)，请参照“**[Hack #62]**利用 dlopen 在运行时动态链接”的内容。

基本要点

在“**[Hack #34]**中，运行堆(heap)中设置的代码”中，讲述了利用mprotect(2)运行堆(heap)的代码的方法。加载器首先把.o文件和.a文件（虽不存在任何意义，但公共.so文件进行同样的操作）整个复制到堆上，或者mmap(2)后，利用mprotect(2)把执行属性添加到该部分即可。此外，复制到堆时，只复制必要的节可以节省内存。再者，在“**[Hack #67]**利用 libbfd 取得符号名称和地址一览表的方法”。与dlsym(3)相当的内容可利用这个信息来实际安装。

剩下的问题，就是在内存中的.o文件及.a文件还没有进行重定位。本Hack中，主要讨论这个问题。

查看重定位信息

```
#include <stdio.h>

void hello() {
    puts("hello");
    puts("world!");
}
```

把这个非常简单的程序按照以下的方法来编译，用 objdump 查询信息。

```
% gcc -g -c hello.c
% objdump -Sr hello.o

hello.o: 文件形式 elf32-i386

节 .text 的反汇编结果：

00000000 <hello>:
#include <stdio.h>

void hello() {
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 ec 08                sub    $0x8,%esp
    puts("hello");
   6:    83 ec 0c                sub    $0xc,%esp
   9:    68 00 00 00 00          push   $0x0
            a: R_386_32  .rodata
   e:    e8 fc ff ff ff          call   f <hello+0xf>
            f: R_386_PC32  puts
   13:   83 c4 10                add    $0x10,%esp
     puts("world!");
   16:   83 ec 0c                sub    $0xc,%esp
   19:   68 06 00 00 00          push   $0x6
   1a:   R_386_32 .rodata
   1e:   e8 fc ff ff ff          call   1f <hello+0x1f>
   1f:   R_386_PC32 puts
   23:   83 c4 10                add    $0x10,%esp
}
   26:   c9                      leave
   27:   c3                      ret
```

在此，-S 是有来源的进行反汇编的选项，-r 是查看重定位信息的选项。联合使用这两个选项，重定位信息明显更容易查看。从这里可以看出，0x09 地址的 push 命令和 0x0e 地址的 call 命令上存在重定位信息，还没有设置正确的地址，各个 0x00000000 及 0xfffffffc 也包含在其中。依次观察一下。

首先是 rodata 一方，它 push 了参数 "hello"。由于这是字符串常量，所以可认为是存在于 rodata（read only data 等）中，试确认是否真的存在。最先要通过 objdump -h 确认 .rodata 的位置。

```
% objdump -h hello.o

hello.o:     文件形式 elf32-i386

节：
索引名:      size        VMA       LMA       File off    Algn
```

```
... 中略 ...

6 .rodata     0000000d    00000000    00000000    000001ee    2**0
              CONTENTS, ALLOC, LOAD, READONLY, DATA

... 后略 ...
```

观察这个可以发现 .rodata 的开始位置是 0x1ee。试利用“**[Hack #4]**通过 od dump 二进制文件”中介绍的 od 查看一下这部分的内容。

```
% od -t x1z -j 0x1ee hello.o | head -1
0000756 68 65 6c 6c 6f 00 77 6f 72 6c 64 21 00 00 10 00
>hello.world!....<
```

可看到 hello 和 world! 依次包含在其中。od 的 -j 选项是指跳过指定位数的选项。第1个 push 中为 0x0，第2个则包含了数字 0x6。这些表示了以 rodata 的起始地址开始的 offset。在重定位 R_386_32 和已指定了地址的时候，加上已指定为原值的节的地址会更好一些。

其次是 call 一方的重定位。这要试着链接一下可执行文件，再读出其运行重定位的输出为好。通过 objdump -S 查看适合的特定部分，结果如下：

```
   puts("hello");
80483aa:       83 ec 0c                sub    $0xc,%esp
80483ad:       68 7c 84 04 08          push   $0x804847c
80483b2:       e8 f1 fe ff ff          call   80482a8 <puts@plt>
80483b7:       83 c4 10                add    $0x10,%esp
   puts("world!");
80483ba:       83 ec 0c                sub    $0xc,%esp
80483bd:       68 82 84 04 08          push   $0x8048482
80483c2:       e8 e1 fe ff ff          call   80482a8 <puts@plt>
80483c7:       83 c4 10                add    $0x10,%esp
}
```

正如右边的输出所示，在 0x080482a8 中存在 puts@plt。虽然两次读出同一个函数，指定的数值却不同，分别为 0xfffffef1 (-271) 和 0xfffffee1 (-287)。也许已经看出了，因为这是相对地址指定的 call。为查看从下一个命令的开始位置起的相对地址，要通过 0x80483b7-271 = 0x80483c7-287 = 0x080482a8 正确指向 puts@plt 的位置。

最终可以发现，在找到重定位信息 R_386_PC32 的情况下，加上已指定为原值的函数的地址，减去重定位信息中的地址会更好。在这里，虽然原本已包含了 0xfffffffc (-4)，但它是从 x86 的相对 call 从下一个命令起的相对地址指定这一点上来说的。

实际安装

到这里的话，实际的安装并没有那么难，应该是一点也不麻烦。显示加载上述的hello.o的程序：

```
#define _GNU_SOURCE

#include <bfd.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include <unistd.h>
#include <sys/mman.h>
#include <dlfcn.h>
bfd *abfd;
asymbol **syms;

/* 找出"hello"符号*/
int get_hello_pos() {
    long storage;
    int symnum;
    int i;

    int hello_pos;

    storage = bfd_get_symtab_upper_bound(abfd);
    assert(storage >= 0);
    if (storage) syms = (asymbol**)malloc(storage);

    symnum = bfd_canonicalize_symtab(abfd, syms);
    assert(symnum >= 0);

    for (i = 0; i < symnum; i++) {
        asymbol *sym = syms[i];
        const char *name = bfd_asymbol_name(sym);
        if (strcmp(name, "hello") == 0) {
            /*这是在hello符号的hello.o中的位置*/
            hello_pos = abfd->origin + sym->section->filepos;
            break;
        }
    }

    return hello_pos;
}

unsigned char *load_hello_o(char *filename) {
    FILE *fp;
    int size = 0;
    unsigned char *hello_o;

    fp = fopen(filename, "rb");
    fseek(fp, 0, SEEK_END);
```

```
    size = ftell(fp);
    fseek(fp, 0, SEEK_SET);
    hello_o = (unsigned char *)malloc(size);
    fread(hello_o, 1, size, fp);
    fclose(fp);

    return hello_o;
}

void reloc_hello_o(unsigned char *hello_o) {
    asection *sect;
    arelent **loc;
    int size;
    int i;

    /* 重定位运行代码 */
    sect = bfd_get_section_by_name(abfd, ".text");

    size = bfd_get_reloc_upper_bound(abfd, sect);
    assert(size >= 0);

    loc = (arelent **)malloc(size);

    size = bfd_canonicalize_reloc(abfd, sect, loc, syms);
    assert(size >= 0);

    for (i = 0; i < size; i++) {
        arelent *rel = loc[i];
        int *p = (int *)(hello_o + sect->filepos + rel->address);
        asymbol *sym = *rel->sym_ptr_ptr;
        const char *name = sym->name;

        /* 重定位节 */
        if ((sym->flags & BSF_SECTION_SYM) != 0) {
            asection *s = bfd_get_section_by_name(abfd, name);
            *p += (int)hello_o + s->filepos;
        }
        /* 重定位函数 */
        else {
            /* hello.o的 hello要读出 hello.o 中的其他函数时,
               该重定位地点需在 syms 中得到，而不是在 dlsym 中得到 */
            *p += (int)dlsym(RTLD_DEFAULT, name);
            if (rel->howto->pc_relative) *p -= (int)p;
        }
    }

    free(loc);
}

void invoke_hello(unsigned char *hello_o, int hello_pos) {
    void (*hello_fp) () = (void (*) ())(hello_o + hello_pos);

    /* 解除内存保护
       参照“[Hack #34]运行堆中设置的代码*/
    int pagesize = (int)sysconf(_SC_PAGESIZE);
```

```
    char *p = (char *)((long)hello_fp & ~(pagesize - 1L));
    mprotect(p, pagesize * 10L, PROT_READ|PROT_WRITE|PROT_EXEC);

    hello_fp();
}

int main() {
    int ret;
    int hello_pos, hello_size;
    unsigned char *hello_o;

    char *filename = "hello.o";

    /* 打开hello.o提前准备*/
    abfd = bfd_openr(filename, NULL);
    assert(abfd);
    ret = bfd_check_format(abfd, bfd_object);
    assert(ret);

    if (!(bfd_get_file_flags(abfd) & HAS_SYMS)) {
        assert(bfd_get_error() == bfd_error_no_error);
        /* there are no symbols */
        bfd_close(abfd);
        return 1;
    }

    hello_pos = get_hello_pos();

    hello_o = load_hello_o(filename);

    reloc_hello_o(hello_o);

    free(syms);
    bfd_close(abfd);

    invoke_hello(hello_o, hello_pos);

    free(hello_o);
}
```

内容有点长，但请依次观察。首先main最初的部分是bfd的初始化部分。此外，get_hello_pos从符号一览表中搜索符号hello。这个部分与“**[Hack #67]**通过libbfd获得符号一览表”类似，故可参照那一部分。在load_hello_o中，读入hello.o，单纯复制到堆上。

reloc_hello_o是这个Hack的主函数。首先，在此可以不用重定位调试信息节等，故取得运行代码上的.text节。同时，通过bfd_get_reloc_upper_bound和bfd_canonicalize_reloc取得重定位一览表。这个过程与取得符号一览表相似。应该注意的是bfd_canonicalize_reloc作为第4变量需要通过bfd_ canonicalize_symtab取得的变量xsyms。虽不明示，这是

很必要的事情，但根据 libbfd 文档（*http: //www.sra.co.jp/wingnut/bfd/bfd-ja_2.html#SEC21*），“表格 syms 也因某些内部原因是必须的”，这里就避免深究了。

再者，依次观察下来取得的重定位信息并进行重定位处理。首先，计算 hello.o 里 .text 节的位置和 rel->address 中作为重定位对象的地址。接着，从 rel->sym_ptr_ptr 中取得符号信息，查看符号名称，再根据符号类型进行条件分类。

符号为节时，要从 bfd 结构体中得到适合的节，加上该位置再进行重定位。符号为函数时，使用 dlsym(3) 查看函数地址，加上该符号的位置再进行重定位。而且，rel->howto->pc->relative 为真时，这就是相对地址指定的函数输出，故减去自身的地址作为对应。在此，由于 hello 函数只读出标准 C 库的函数，故可通过 dlsym 取得 O 函数地址，但读出 hello.o 的别的函数时，需从 syms 的信息中查看地址。

到此，所有的准备都已完成了，最后通过 invoke hello 解除 hello 函数的内存保护再运行。

补充

在本 Hack 中，为只运行 hello.o 中的标准库函数以外没有调用的 hello 函数进行了重定位。实际的重定位会是更麻烦的处理。例如，前面提到的在加载后的目标文件中个别函数的读出，以及 .a 文件的输入等，却并不得到支持。另外，重定位的种类并不限于 R_386_32 和 R_386_PC32，总之虽不够充分，但在笔者所写的 DTR（*http:// shinh.skr.jp/binary/dtr.html*）中已进行了少许较正式的重定位。

根据这次的 Hack，要加载 Java 的 .class 文件，不用特意变更编译器选项也可进行加载文件。另外，要在不支持共享库的环境中实现插入，也可使用本次的 Hack 。

在 XFree86 中，plug-in 文件为 .a 文件（libdri.a 等）。这确实可用如本 Hack 的自行加载来实现，在 x 的源代码中带着各环境目标文件的加载器（hw/xfree86/loader/）已变成安装的参考。

总结

在本Hack中，介绍了加载目标文件的方法。这个技术可用在自行plug-in结构的实现等上。

—— *Shinichiro Hamaji*

通过 libunwind 控制 call chain

使用libunwind可得到call chain的信息，也可使用该信息进行unwind。

在这里，即将介绍控制call chain的库libunwind（*http://www.hpl.hp.com/research/linux/libunwind/*）。libunwind是由HP开发出来的库，通过MIT许可证分布。如今，libunwind在IA-64 Linux中完全得到支持，在x86 Linux和IA-64 HP-UX中也可得到基本的支持。

一般来说unwind表示堆栈的反处理。作为典型的反处理，虽有C语言的return句，但使用libunwind的话，可以对多个函数一次性进行反处理。此外，由于可以取得call chain的信息，back trace可也可轻易得知是从哪里读出的。

本Hack介绍libunwind的简单功能。

用 libunwind 表示 backtrace

以下列出了使用libunwind表示back trace的程序。

```
#include <libunwind.h>

void show_backtrace() {
    unw_cursor_t cursor;
    unw_context_t uc;
    unw_word_t ip, sp;
    char buf[4096];
    int offset;

    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);
    while (unw_step(&cursor) > 0) {
        unw_get_reg(&cursor, UNW_REG_IP, &ip);
        unw_get_reg(&cursor, UNW_REG_SP, &sp);
        unw_get_proc_name(&cursor, buf, 4095, &offset);
        printf("0x%08x <%s+0x%x>\n", (long)ip, buf, offset);
```

```
    }
}

void func() {
    show_backtrace();
}

int main() {
    func();
    return 0;
}
```

应该没什么特别难的地方，运用结果如下：

```
% ./a.out
0x080489c9 <func+0xb>
0x080489ec <main+0x21>
0x41032d5f <__libc_start_main+0xdf>
0x0804886d <_start+0x21>
```

通过 libunwind 进行 unwind

接下来也看看通过 libunwind 多次 return 的例子。下面列出了样本代码。

```
#include <libunwind.h>

void skip_func() {
    unw_cursor_t cursor;
    unw_context_t uc;

    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);
    unw_step(&cursor);
    unw_step(&cursor);
    unw_resume(&cursor);
    printf("will be skipped.\n");
}

void skipped_func() {
    skip_func();
    printf("will be skipped.\n");
}

int main() {
    printf("start.\n");
    skipped_func();
    printf("end.\n");
    return 0;
}
```

在 skip_func 中，在两次读出 unw_step，反处理 skip_func => skipped_func => main 与堆栈框架的光标之后，通过 unw_resume 再次

返回。利用这个立即便回到 main，所以两个 "will be skipped.\n" 无法输出。

自行进行 unwind

在此，介绍使用 getcontext/setcontext(2) 自行进行简单 unwind 的方法。

```
#define _GNU_SOURCE
#include <stdio.h>
#include <ucontext.h>

typedef struct layout {
    struct layout *ebp;
    void *ret;
} layout;

void skip_func() {
    ucontext_t uc;
    layout *ebp = __builtin_frame_address(0);
    ebp = ebp->ebp;

    getcontext(&uc);
    uc.uc_mcontext.gregs[REG_EIP] = (unsigned int)ebp->ret;
    uc.uc_mcontext.gregs[REG_EBP] = (unsigned int)ebp->ebp;
    setcontext(&uc);

    printf("will be skipped.\n");
}

void skipped_func() {
    skip_func();
    printf("will be skipped.\n");
}

int main() {
    printf("start.\n");
    skipped_func();
    printf("end.\n");
    return 0;
}
```

在这，利用“[Hack #63]中用 C 来表示回溯”介绍的追溯堆栈帧的方法。可获得返回地点的 ebp 和 eip。在 getcontext/setcontext(2) 中使用的 ucontext 的构造体的 uc_mcontext 成员。正如“[Hack #78]从信号 handler 着手改变程序的脉络”说明的一样，必须依靠安装。请结合环境改换程序中的代入部分。

在libunwind中，包括这里介绍的方法，各个架构都可实行各种各样有效的方法。

其他的功能

在libunwind中，因为以上的处理都是跳过ptrace(2)进行的，所以可获得其他process call chain的信息，还提供有操作方法。

另外，也提供高效率的setjmp/longjmp。setjmp高速，longjmp则会变成低速，但若在例外处理等方面，可说得上是很好的trade-off。

总结

所谓unwind，是像return的可跨越多个函数完全回溯工作。使用libunwind，可获得call chain的信息，并可使用这些信息进行unwind。

—— *Shinichiro Hamaji*

用GNU lightning Portable生成运行编码

使用GNU lighting，从可移植的汇编程序编码入手可在运行时生成机器语言。

“**[Hack #34]**运行heap上的代码”中介绍了为了运行heap上的编码而去除内存保护的方法。使用这种方法，运行时虽可生成native编码并能运行，但如此一来无论如何都得依靠处理器，在本章Hack中将介绍一种用GNU lighting库不依靠处理器来进行上述工作的方法。

GNU lightning

GNU lightning（*http://www.gnu.org/software/lightning/*）是用来简单编写汇编程序运行的库。用LGPL分布。属于同一概念，还有libjit（*http://www.southern-storm.com.au/libjit.html*）这么一个库，但在这里不做介绍。

在使用GNU lightning的编程中，在C编码上将记述不依赖处理器的抽象性的汇编程序。GNU lightning将帮我们使汇编程序符合CPU，转化为运行编码。GNU lightning与x86、SPARC、PowerPCLGPL互相支持，浮动小数点的功能只有x86支持。

使用GNU lightning 的编程虽与一般的汇编程序编程相似，但也有两个很大的不同之处。一个是由于所有的命令都被抽象化，架构的“习惯”几乎没有，特别是为了使函数调用抽象化而不考虑调用规则会好一些。第 2 个是，为了使它抽象化，寄存器的个数与少数架构相配合，只有 6 个使用 GNU lightning。

用 C 语言尝试一下便携安装简单化的 curry

在GNU Lightning中用C语言尝试安装简单并且可移植的curry。在这里，所谓curry化，就是先设置函数的参数的一部分再得到参数的个数减少后的函数的方法。了解C++的STL的各位回想起bind1st、bind2nd就好了。以下是安装后的代码。

```
#include <stdio.h>

#include <lightning.h>

typedef int (*pifi)(int);
static jit_insn buf[1024];

pifi curry(int (*fp)(int, int), int a) {
    pifi code = (pifi)(jit_set_ip(buf).iptr);
    int i;

    jit_prolog(1);
    i = jit_arg_ui();
    jit_getarg_ui(JIT_V0, i);
    jit_movi_ui(JIT_V1, a);
    jit_prepare(2);
    jit_pusharg_ui(JIT_V1);
    jit_pusharg_ui(JIT_V0);
    jit_finish(fp);
    jit_retval(JIT_RET);
    jit_ret();
    jit_flush_code(buf, jit_get_ip().ptr);

    return code;
}

int add(int x, int y) {
    return(x + y);
}

int main() {
    pifi c;

    c = curry(add, 100);
    printf("%d\n", c(10));
}
```

在main中，可知把进行加法运算的二元函数和100这个数curry 化将生成原函数。由于用10这个参数运行出的这个结果函数。计算100+10，则输出110。那么就让我们依次来看重要的curry函数。

首先，用jit_prolog(1)显示定义一个参数的函数，然后用jit_arg_ui()和jit_getarg_ui获得参数。参数的获得地址是JIT_V0这个GNU lightning形式的寄存器。

其次，用jit_movi_ui(JIT_V1, a)把curry的第2个参数转移到JIT_V1寄存器。因为这个阶段还是编码生成的阶段，请注意a的值以即值形式转移。

因为必须的都已经聚集，可进入函数调用这一环节。用jit_pusharg_ui把两个值作为参数set。并且用jit_finish(fp)调用函数。调用的函数的返回值将进入专用的寄存器。用JIT_RET把这个函数的返回值作为生成的函数的返回值，用jit_ret()完成函数jit_retval(JIT_RET)。

最后，用jit_flush_code把这之前编写的编码写到buf上。这时，mprotect(2)的去除内存保护功能也将自动启动，只需运行函数就可以了。

GNU lightning 的方法

以上是非常简单的例程，GNU lightning另外还支持许多的汇编程序的基本的演算。例如加减乘除、理论演算、条件选择等。这些GNU lightning的命令基本上也和jit_movi_ui一样遵从jit_<op><immediate or register>_<type>这一规则。例如：在float寄存器中进行的加法运算要转换为jit_addr_f。

能使用的int型寄存器有JIT_V0、JIT_V1、JIT_V2、JIT_R0、JIT_R1、JIT_R2、JIT_RET 7个。JIT_V系通过函数保证数值的保存，JIT_R系通过函数调用等可能改变数值。JIT_RET是存储返回值的寄存器，在JIT_RET中代入不同数值，根据架构的不同可能出现破坏其他的多用途寄存器的值。(例如在x86中，JIT_RET和JIT_R0两者都是eax寄存器)。只用于返回值的接收也很好吧。

另外，disassemble也附属于GNU lightning。它是disassemble生成的编码的补助函数。由于运行/configure --enable-disassembling导致生成opcode/ libdisass.a。

总结

使用GNU lightning后，可以在运行时从可移植的汇编代码中生成机器语言。

—— *Shinichiro Hamaji*

获得 stack 的地址

本Hack介绍在主要的操作系统上获取进程堆栈和线程堆栈的内存地址的方法。

在实际安装中充分采用了stack的垃圾回收时或想要事先补足堆栈溢出的话，就有必要查清堆栈区域处于内存空间的哪个位置，本 Hack 将介绍运行中的 stack 及取得堆栈地址信息的方法。

不使用多线程时的 stack 信息

UNIX 操作系统（未多线程化）进程的 stack 依赖于进程的存储空间的使用方法。stack 的大小一般可通过 `getrlimit(2)` 取得。

```
struct rlimit rlim;
getrlimit(RLIMIT_STACK, &rlim);
size_t process_stack_size = (size_t)rlim.rlim_cur;
```

但并没有获知stack源地址的标准方法。有很多操作系统从既定的地址开始。这里介绍与几个 OS 有关的 stack 源地址的程序取得方法。

Linux 系统

Linux的每个平台都有固定的stack源地址(Linux/i686型的是0xbfffffff)。但从 2.6.12 内核开始，随机错离 stack 源地址的 stack 保护功能变成了缺省。(由 */proc/sys/kernel/randomize_va_space* 控制。从固定值向 stack 的发展偏离最多 8MB。)

因此为了取得 stack 源地址，会用到 glibc 内的符号 `_libc_stack_end`。

```
#include <unistd.h> /* for sysconf(3) */
#include <stdint.h> /* for uintptr_t */

#pragma weak __libc_stack_end
extern void* __libc_stack_end;

void* get_linux_stack_base() {
  long pagesize = sysconf(_SC_PAGESIZE);
```

```
  return (void *)(((uintptr_t)_libc_stack_end + pagesize) &
~(pagesize -1));
}
```

FreeBSD 系统

FreeBSD 的 stack 源地址可通过 `sysctl` 获得。

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#include <assert.h>

void* get_freebsd_stack_base() {
  int nm[2] = {CTL_KERN, KERN_USRSTACK};
  void* base;
  size_t len = sizeof(void*);
  int r = sysctl(nm, 2, &base, &len, NULL, 0);
  assert(r);
  return base;
}
```

能使用 procfs 的情况下

通过在“**[Hack #65]**核对已装载的共享数据库”介绍的方法，能找到 stack 区域。先从本地变量的地址求出堆栈指针（SP），再寻找包含 SP 的领域。有些操作系统可通过 API 得到和 procfs 同样的信息。在 HP-UX 上能使用 `pstat(2)`。

```
#include <sys/param.h>
#include <sys/pstat.h>

struct pst_vm_status vm_status;
int i = 0;
while (pstat_getprocvm(&vm_status, sizeof(vm_status), 0, i++) == 1) {
  if (vm_status.pst_type == PS_RSESTACK) {
    void* base = (void*)vm_status.pst_vaddr;
    size_t size = (size_t)vm_status.pst_length;
 }
}
```

其他

下面介绍用以取得线程堆栈信息的 API。有的操作系统可取得进程堆栈，连接线程库也是手段之一。

多线程的线程堆栈呢？

在进程中新生成的线程，线程域会自动分配 stack 领域，可是并没有取得此线程领域的标准方法。几个操作系统有各自的 API。

表 5-1：各个 OS 获得 stack 领域情况的 API

OS	API 名称	include 文件
Linux	pthread_getattr_np	pthread.h
FreeBSD	pthread_attr_get_np	pthread_np.h
OpenBSD	pthread_stackseg_np(3)	pthread_np h & sys/signal.h
Mac OS X	pthread_get_stacksize_np, pthread_get_stackaddr_np	pthread.h
Solaris	thr_stksegment(3thr)	thread.h & sys/signal.h

上表中的 _np 表示意思是 non-portable，因为是不可移植的 API，所以必须注意。

Linux 系统的 Hack

Linux 通过 pthread_getattr_np 可从生成后的线程得知属性信息。但有的线程或库并未定义 pthread_getattr_np，有时不能得到正确信息，所以有必要分条件讨论。

1. 一开始就存在的进程的线程，从 pthread_getattr_np 无法得到正确信息。要使用从进程主线程的 getrlimit 得到的信息。
2. 旧的线程库中没有 pthread_getattr_np。这种旧线程库，thread stack 被固定在半径为2MB的圆上的某点。从stack指针的位置能逆算出stack领域。

根据上述情况，体现 Linux 线程区域的 get_linux_stack_info 变成以下符号。

```
#include <pthread.h>
#include <sys/resource.h>
#include <unistd.h>
#include <dlfcn.h>
#include <unistd.h>
#include <stdint.h>
#pragma weak __libc_stack_end
extern void* __libc_stack_end;

#define INITIAL_PROCESS_STACK_END ((char*)0xC0000000U)
#define DEFAULT_FIXED_STACK_SIZE (2 * 1024 * 1024)

typedef int  (*GETATTR_NP_FUNC)(pthread_t, pthread_attr_t *);
typedef int  (*ATTR_GETSTACKBASE_FUNC)(pthread_attr_t *, void**);
typedef int  (*ATTR_GETSTACKSIZE_FUNC)(pthread_attr_t *, size_t*);
```

```
typedef int  (*ATTR_GETSTACK_FUNC)(pthread_attr_t *, void**
stackaddr, size_t* stacksize);

int get_linux_stack_info(void** stackaddr, size_t* stacksize) {
  char dummy;
  char* p = &dummy;
  char* initial_process_stack_end = INITIAL_PROCESS_STACK_END;
  size_t process_stack_size = 0;
  long pagesize = sysconf(_SC_PAGESIZE);
  struct rlimit rlim;

  getrlimit(RLIMIT_STACK, &rlim);
  process_stack_size = (size_t)rlim.rlim_cur ;

  if (&__libc_stack_end && __libc_stack_end) {
    initial_process_stack_end = (char*)(((uintptr_t)__libc_stack_end
+ pagesize) & ~(pagesize -1));
}

if (initial_process_stack_end - process_stack_size <= p &&
    p <= initial_process_stack_end) {
  /* process thread的情况 */
  *stackaddr = (void*)(initial_process_stack_end - process_stack_size);
  *stacksize = process_stack_size;
  return 0;
} else {
  GETATTR_NP_FUNC getattr_np_func =
    (GETATTR_NP_FUNC)dlsym(NULL, "pthread_getattr_np");

  if (!getattr_np_func) {
    /*在旧的thread系统中stack的大小固定为2MB*/
    *stackaddr = (void*)( (size_t)p & (DEFAULT_FIXED_STACK_SIZE - 1));
    *stacksize = DEFAULT_FIXED_STACK_SIZE;
    return 0;
  } else {
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    if (!getattr_np_func(pthread_self(), &attr)) {
      ATTR_GETSTACK_FUNC attr_getstack_func =
        (ATTR_GETSTACK_FUNC) dlsym(NULL, "pthread_attr_getstack" );
      ATTR_GETSTACKBASE_FUNC attr_getstackaddr_func =
        (ATTR_GETSTACKBASE_FUNC) dlsym(NULL, "pthread_attr_getstackaddr");
      ATTR_GETSTACKSIZE_FUNC attr_getstacksize_func =
        (ATTR_GETSTACKSIZE_FUNC) dlsym(NULL, "pthread_attr_getstacksize");
      if (attr_getstack_func) {
        int ret = attr_getstack_func(&attr, stackaddr, stacksize);
        pthread_attr_destroy(&attr);
        return ret;
       } else if (attr_getstackaddr_func && attr_getstacksize_func) {
         int ret = attr_getstackaddr_func(&attr, stackaddr) ||
                   attr_getstacksize_func(&attr, stacksize);
         pthread_attr_destroy(&attr);
         return ret;
       }
```

```
        }
        pthread_attr_destroy(&attr);
      }
    }

    return -1;
}
```

Windows 系统的 Hack

Windows 系统中，在 TIB（Thread Information Block 的缩写）中存储了关于线程的信息，x86/Windows 系统中，段寄存器 FS 被设置成在每段线程指向不同的段。通过以下符号可取得 TIB。

```
/* 取得TIB的函数*/
NT_TIB* getTIB(void) {
  NT_TIB* pTib;
  __asm {
    mov eax, dword ptr FS:[18H];
    mov pTib, eax;
  }
  return pTib;
}

/* stack领域的判定 */
NT_TIB* pTIB = getTIB();
printf("[%p %p]\n", pTIB->StackBase, pTIB->StackLimit);
```

关于 NT_TIB 请参考 SDK 平台的 `WinNT.h`。

总结

本章介绍了在主要的操作系统中取得进程堆栈和线程堆栈的内存地址的方法。调查 stack 领域的方法在不同版本的操作系统上有时无法运用。在实际使用前，检验其能否在目标环境中运行的测试是必不可少的。

—— *Minoru Nakamura*

HACK #76 用 sigaltstack 处理 stack overflow

本节介绍使用signalstack处理堆栈溢出的方法。对于编译器和解释器runtime等不清楚使用了多少stack的程序，不要忘记堆栈溢出方面的维护。

充分利用反呼叫或 `alloca(3)` 的程序，有时 stack 会超过上限，引起堆栈溢出。本节介绍如何捕捉堆栈溢出的错误，并适当进行修复的方法。

操作系统检测出堆栈溢出的结构

stack 领域被配置在处理器空间的哪个部分，这在线程生成时就已经被安排好。几乎所有的 UNIX 系统都是如图 5-1 所示，从高位地址开始使用 stack，然后向下（低位）延伸。stack 领域的终端是最低位的虚拟存储页（有时是几页）。通过把这些页设为 PROT_NONE 禁止访问来实现栈溢出的检测。如果 stack 延伸出这一领域，memory 保护功能就会启动，发出 SEGV 信号。这个禁止领域叫做保护页或红色区域。

PA-RISC 和 IA-64 需要特殊的 stack 配置。因为 PA-RISC 的 stack 是从低位开始向高位延伸。红色区域被配置在最高位。IA-64 和通常的线程堆栈不同，因为必须有寄存器堆栈的退避领域，所以把 red zone 配置在中间，从高位到低位，程序把它用作线程堆栈，从低位到高位寄存器 stack engine 也可以使用。

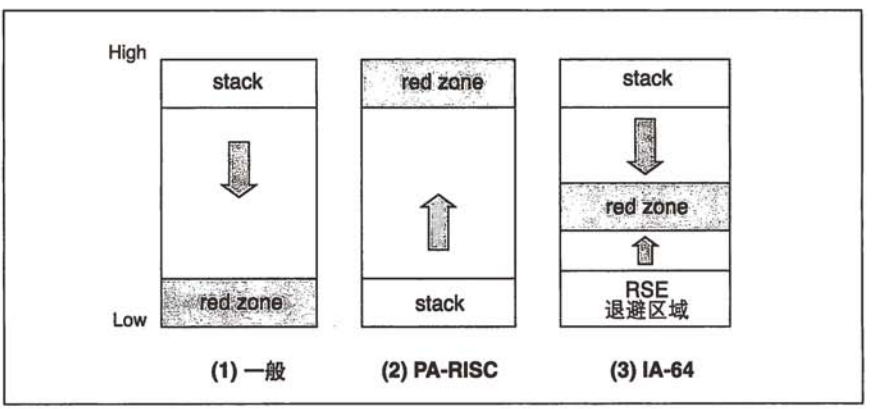


图 5-1：stack 领域

使用 sigaltstack

因为 SEGV 等同期信号利用的是原线程的 stack，所以堆栈溢出后没有驱动信号处理器的堆栈溢出，用 signal(2) 或 sigaction(2) 无法捕捉堆栈溢出的 SEGV 信号。 有必要设定 alternate signal stack 代替原 stack thread，可通过使用 sigaltstack(2) 在每个线程中设定。

作为 alternate signal stack 的领域，由 malloc(3) 确保传递，以下的符号是 stack 达到 red zone，发生 SEGV 信号时，使用 sigsetjmp/siglongjmp 来恢复到原程序的例子。

样本代码 1

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <pthread.h>
#include <setjmp.h>

#define ALT_STACK_SIZE (64*1024)

static sigjmp_buf return_point;

static void signal_handler(int sig, siginfo_t* sig_info, void* sig_data) {
  if (sig == SIGSEGV) {
    siglongjmp(return_point, 1);
  }
}

static void meaningless_recursive_func() {
  meaningless_recursive_func();
}

static void register_sigaltstack() {
  stack_t newSS, oldSS;

  newSS.ss_sp = malloc(ALT_STACK_SIZE);
  newSS.ss_size = ALT_STACK_SIZE;
  newSS.ss_flags = 0;

  sigaltstack(&newSS, &oldSS);
}

int main(int argc, char** argv) {
  struct sigaction newAct, oldAct;

  /* 代替信号stack 的设定 */
  register_sigaltstack();

  /* 信号handle的设定 */
  sigemptyset(&newAct.sa_mask);
  sigaddset(&newAct.sa_mask, SIGSEGV);
  newAct.sa_sigaction = signal_handler;
  newAct.sa_flags = SA_SIGINFO|SA_RESTART|SA_ONSTACK;

  sigaction(SIGSEGV, &newAct, &oldAct);

  /* 特意使stack overflow 发生进行handle */
  if (sigsetjmp(return_point, 1) == 0) {
    meaningless_recursive_func();
  } else {
    fprintf(stderr, "stack overflow error\n");
  }

  return 0;
}
```

设置 yellow zone

上述的直接捕捉 red zone 的 SEGV 信号是捕捉堆栈溢出的基本方法，但在实际应用程序时会遇到各种问题。从信号处里通过 siglongjmp(3) 返回的话，无法对线程内已打开资源进行善后处理。

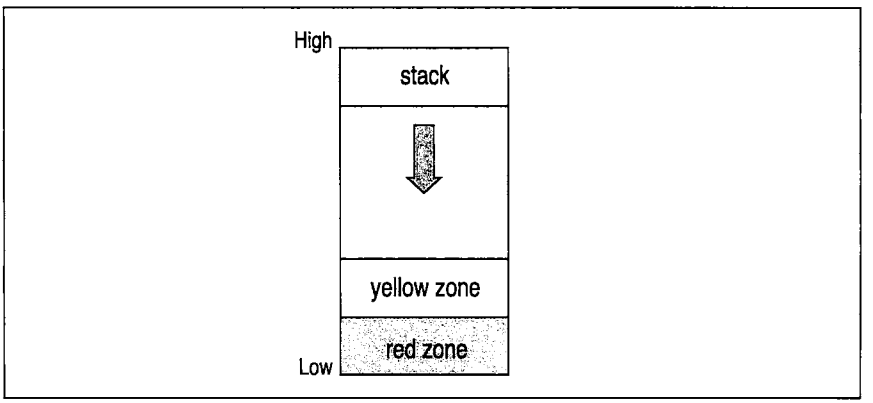


图 5-2：夹入了 yellow zone 的 stack 领域的形象

为了解决这个问题，采取了yellow zone的手段。这个手段就是在stack内red zone的前面外设置额外的一页（或数页）具有 PROT_NONE 属性的虚拟存储页，再用 SEGV 信号来捕捉它。这额外的一页叫做yellow zone。yellow zone 的 SEGV 信号一旦发生，就会在信号处理器中用 mmap(2) 将 memory map 化，并设定能用 mprotect(2) 写入、读入的属性。从而恢复到原来环境。复原的 stack 能额外使用的部分只有 yellow zone。

以下是在 Linux 系统下使用 yellow zone 的例子。用 i386/RHEL4..i386/RHEL3..Vine Linux 3.1 等来确定其运行。这个方法有必要正确获知 stack 领域在 memory 空间中处于什么位置。但获得 stack 领域信息的方法其写法随 OS 的不同而不同。请参考“**[Hack #75]**将 get_stack_info() 进行变换”。yellow_zone_hook() 记载了 yellow zone 的 SEGV 信号发生时的处理，但通过并用像**[Hack #78]**一样的 Hack，在环境这方面产生 C++ 的例外这样的功能可得到应用。

样本代码 2

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

```
#include <sys/mman.h>
#include <unistd.h>
#include <assert.h>
#include <sys/resource.h>

#define ALT_STACK_SIZE (64*1024)
#define YELLOW_ZONE_PAGES (1)

/* 各个thread的stack信息用Thread-specific data记载
 * 这个构造体想象设定downward-growing stack 3个pointer
   都需处在页码边界。
 *
 * +---------------+ <- stack_pointer + stack_size
 * | |
 * +---------------+ <- yellow_zone_boundary
 * | |
 * +---------------+ <- red_zone_boundary
 * | |
 * +---------------+ <- stack_pointer(stack 的最低位的地址)
 *
 */
typedef struct {
 size_t      stack_size;           /* stack领域的大小 */
 char*       stack_pointer;        /* stack的最低位地址 */
 char*       red_zone_boundary;    /* red zone的边界
                                      (不含red zone本身) */
 char*       yellow_zone_boundary; /* yellow zone的边界
                                      /* 无yellow zone的情况NULL */
 sigjmp_buf return_point;          /* 超出red zone 时的返回地址*/
 size_t red_zone_size;             /* 操作用 */
} ThreadInfo;

/* ThreadInfo用来记录的threadinfo的Thread-specific key记录*/
static pthread_key_t thread_info_key;

/* sig 信号 handle */
static struct sigaction newAct, oldAct;

/*********************************************************************/
/* user 定义routine */
/*********************************************************************/

/* main的处理 */
static void main_routine() {
  /* 使栈溢出 */
  main_routine();
}

/* 到达red zone时的处理 */
static void stackoverflow_routine() {
  fprintf(stderr, "stack overflow error.\n");
  fflush(stderr);
}

/* 超出yellow zone 情况下的处理 */
```

```
static void yellow_zone_hook(/* 用参数来获取想从信号handle获得的信息 */) {
  /* 在此记录喜欢的处理 */
  fprintf(stderr, "exceeded yellow zone.\n");
  fflush(stderr);
}

/* stack领域信息获得 (各个书写方法各异) */
static int get_stack_info(void** stackaddr, size_t* stacksize) {
  int ret = -1;
  pthread_attr_t attr;
  pthread_attr_init (&attr);

  if (pthread_getattr_np(pthread_self(), &attr) == 0) {
    ret = pthread_attr_getstack(&attr, stackaddr, stacksize);
  }
  pthread_attr_destroy (&attr);

  return ret;
}

/*********************************************************************/
/* stack overflow handlling的框架 */
/*********************************************************************/

/* pointer处在stack领域内 */
static int is_in_stack(const ThreadInfo* tinfo, char* pointer) {
  return (tinfo->stack_pointer <= pointer) && (pointer < tinfo-
>stack_pointer + tinfo->stack_size);
}

/* pointer处在red zone中 */
static int is_in_red_zone(const ThreadInfo* tinfo, char* pointer) {
  assert(tinfo->red_zone_boundary);
  return (tinfo->stack_pointer <= pointer) && (pointer < tinfo-
>red_zone_boundary);
}

/* pointer处在yellow zone中 */
static int is_in_yellow_zone(const ThreadInfo* tinfo, char* pointer) {
  if (tinfo->yellow_zone_boundary) {
    return (tinfo->red_zone_boundary <= pointer) && (pointer <
tinfo->yellow_zone_boundary);
  }
}

/* 设置 yellow zone */
static void set_yellow_zone(ThreadInfo* tinfo) {
  int pagesize = sysconf(_SC_PAGE_SIZE);
  assert(pagesize > 0);
  tinfo->yellow_zone_boundary = tinfo->red_zone_boundary + pagesize *
YELLOW_ZONE_PAGES;
  mprotect(tinfo->red_zone_boundary, pagesize * YELLOW_ZONE_PAGES,
PROT_NONE);
}
```

```
/* 撤销yellow zone */
static void reset_yellow_zone(ThreadInfo* tinfo) {
  size_t pagesize = tinfo->yellow_zone_boundary - tinfo->
red_zone_boundary;
  if (mmap(tinfo->red_zone_boundary, pagesize, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, 0, 0) == 0) {
    perror("mmap failed"), exit(1);
  }
  mprotect(tinfo->red_zone_boundary, pagesize, PROT_READ|PROT_WRITE);
  tinfo->yellow_zone_boundary = 0;
}

/* 信号handle */
static void signal_handler(int sig, siginfo_t* sig_info, void* sig_data) {
  if (sig == SIGSEGV) {
    ThreadInfo* tinfo = (ThreadInfo*)pthread_getspecific(thread_info_key);
    char* fault_address = (char*)sig_info->si_addr;

    if (is_in_stack(tinfo, fault_address)){
      if (is_in_red_zone(tinfo, fault_address)) {
        /* 进入red zone的情况 */
        siglongjmp(tinfo->return_point, 1 /* 在这是非0的任意整数 */ );
      } else if (is_in_yellow_zone(tinfo, fault_address)) {
        /* 进入yellow zone的情况 */

        /*
         * 由于yellow zone返回到通常的stack。现在发生的SEGV被解除
         * 因此可争取少量时间
         */
        reset_yellow_zone(tinfo);

        /*
         * 把不能回避的stack overflow接近这一情况向main程序方传达。
         */
        yellow_zone_hook(/* 列举必要的信息 */);
        return;
      } else {
        /* .在stack领域内与overflow无关连的SEGV发生 */
      }
    }
    /* 根据要求 */
    /* oldAct.sa_sigaction(sig, sig_info, sig_data); */
  }
}

/* 用application登录一次信号handle和TSkey */
static void register_application_info() {
  /* Thread-specific 的登录 */
  pthread_key_create(&thread_info_key, NULL);

  /* set SEGV信号handle */
  sigemptyset(&newAct.sa_mask);
  sigaddset(&newAct.sa_mask, SIGSEGV);
  newAct.sa_sigaction = signal_handler;
  newAct.sa_flags     = SA_SIGINFO | SA_RESTART | SA_ONSTACK;
```

```
  sigaction(SIGSEGV, &newAct, &oldAct);
}

/* 各个thread登录TSD和大体信号handle */
static void register_thread_info(ThreadInfo* tinfo) {
  stack_t ss ;

  /* 登录TSDtinfo*/
  pthread_setspecific(thread_info_key, tinfo);

  /* stack领域的登录 */
  get_stack_info((void**)&tinfo->stack_pointer, &tinfo->stack_size);

  /* red zone的登录 */
  tinfo->red_zone_boundary = tinfo->stack_pointer + tinfo->red_zone_size;

  /* yellow zone的登录 */
  set_yellow_zone(tinfo);

  /* 代替信号stack的登录 */
  ss.ss_sp = (char*)malloc(ALT_STACK_SIZE);
  ss.ss_size = ALT_STACK_SIZE;
  ss.ss_flags = 0;
  sigaltstack(&ss, NULL);
}

/* 各线程的基本部门 */
static void* thread_routine(void* p) {
  ThreadInfo* tinfo = (ThreadInfo*)p;

  /* 线程特殊的信息登录 */
  register_thread_info(tinfo);

  if (sigsetjmp(tinfo->return_point, 1) == 0) {
    /* 主要的处理 (在这其中发生的stack溢出的可能性) */
    main_routine();
  } else {
    /* 从red zone中来的return point */
    stackoverflow_routine();
  }
  free(tinfo);
  return 0;
}

int main(int argc, char** argv) {
  /* signal handler 的登录 */
  register_application_info();

  if (argc == 2) {
    int stacksize = atoi(argv[1]);

    pthread_attr_t attr;
    pthread_attr_init (&attr);
    pthread_attr_setstacksize(&attr, 1024 * 1024 * stacksize);

    {
```

```
        pthread_t pid0;
        ThreadInfo* tinfo = (ThreadInfo*)calloc(1, sizeof(ThreadInfo));
        pthread_attr_getguardsize(&attr, &tinfo->red_zone_size);
        pthread_create(&pid0, &attr, thread_routine, tinfo);
        pthread_join(pid0, NULL);
      }
    } else {
      printf("Usage: %s stacksize(mb)\n", argv[0]);
    }
    return 0;
}
```

在旧 Linux 中强制使用 sigaltstack

比较古老的 Linux 带有被称为 Fixed stack 的 2MB 固定的 stack 领域。Fixed stack 会沿着 2MB 边界将 stack 领域配置在内存空间上。利用这个特性来识别线程和把握 stack 领域。将 malloc(3) 等外侧的内存分配到 sigaltstack 上后，将发生不能明确识别线程的错误。

Red Hat Linux 7.0 使用了以前容易发生问题的内核 & 线程库。使用这些操作 sigaltstack 需要特殊的 Hack。这些 distribution 若在 2MB 领域的外侧的话会发生 Fixed stack 障碍，所以要将图 5-3 那样将代替 sigaltstack 嵌入在原来的 stack 中。将样本代码 1 复原并像以下变更 register_sigaltstack() 就 OK 了。

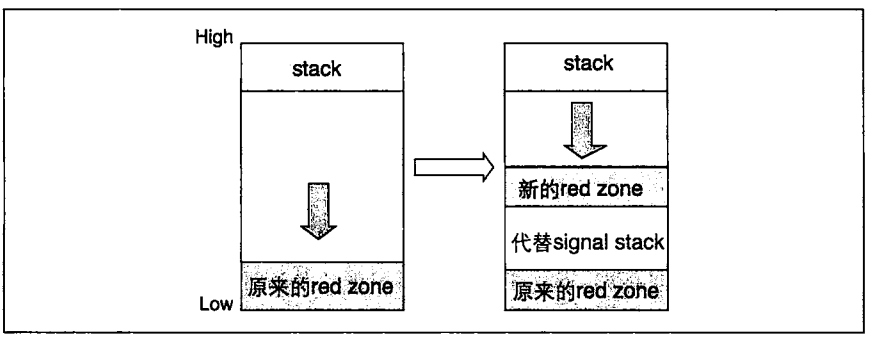


图 5-3：Fixed stack 在 Linux 中使用 signalstack

样本代码 3

```
#include <sys/mman.h>

#define ORIGINAL_RED_ZONE_SIZE (64*1024) /* ..................... */
#define NEW_RED_ZONE_SIZE    (4*1024)   /* x86 ............... */
#define FIXED_STACK_SIZE     (2*1024*1024)

static void register_sigaltstack() {
```

```
    stack_t newSS, oldSS;
    char* stack_base = (char*)((size_t)&newSS & ~(FIXED_STACK_SIZE-1));

    /* 在stack领域内设定代替signal stack */
    newSS.ss_sp = (void*)(stack_base + ORIGINAL_RED_ZONE_SIZE);
    newSS.ss_size = ALT_STACK_SIZE;
    newSS.ss_flags = 0;

    sigaltstack(&newSS, &oldSS);

    /* 将内存映射到代替signal领域和新red zone上 */
    mmap(newSS.ss_sp, newSS.ss_size + NEW_RED_ZONE_SIZE,
         PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);

    /* 将新red zone设定为PROT_NONE */
    mprotect(stack_base + ORIGINAL_RED_ZONE_SIZE + ALT_STACK_SIZE,
             NEW_RED_ZONE_SIZE, PROT_NONE);

}
```

注意事项

sigaltstack 有很多bug和带有限制的安装系统。

- Solaris的情况中，在Solaris 8以前的线程库里没有为LWP bound的线程中不能使用sigaltstack。
- 不能设想代替signal stack会溢出，要注意尽量不使其溢出。
- 用pthread_create从设定了sigaltstack的线程中生成子线程后，有很多复制代替signal stack的处理系统。此时，母线程和子线程同时发出信号的话则有发生内存破坏的意外危险。

总结

在本Hack中，介绍了通过使用sigaltstack来处理堆栈溢出的方法。在程序中没有堆栈溢出的意识，但在不能估计运用堆栈到编译器和解使器的哪里为止，要注意防止出现堆栈溢出。

—— *Minoru Nakamura*

hook面向函数的enter/exit

使用GCC-finstrument-functions选项，可以调用面向函数的enter/exit时自生成的函数。

利用了GCC的-finstrument-functions选项后，在刚读出C/C++的函数

后，以及从这个函数中 return 之前，可以读出自动生成的函数。在 本 Hack 中介绍了这个 -finstrument-functions 选项的使用方法。

使用方法

将下面的hook函数放入源代码的某处，通过附上-finstru ment-functions 选项将源代码全体进行编译，可以 hook 到函数的 enter/exit，非常简单。

```
__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *func_address, void *call_site) {
  // 在面向函数的enter时执行处理
}
__attribute__((no_instrument_function))
void __cyg_profile_func_exit(void *func_address, void *call_site)
{
  // 在从函数中 exit 时执行处理
}
```

在执行enter后，变量func_address得到即将exit的函数的地址。call_site 是调用该函数的函数地址。

运用例：测定进程的 stack 使用量

-finstrument-functions有很多用途，下面试着以“测定进程的stack使用量”为例。

生成 stack 函数和共享库化

首先，生成计算当前stack使用量的hoot函数__cyg_profile_func_enter，由于__cyg_profile_func_exit 不执行什么处理所以不被生成。

```
// stack_usage.c
#include <stdio.h>
#include <stddef.h>
static ptrdiff_t max_usage = 0;

__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *func_address, void *call_site) {
  extern void *__libc_stack_end;
  const ptrdiff_t usage = __libc_stack_end - __builtin_frame_address(0);
  if (usage > max_usage) max_usage = usage;
}

__attribute__((no_instrument_function, destructor))
static void print_usage() {
  printf(" stack使用量：大约 %td byte\n", max_usage);
}
```

print_usage是在程序终止时自动运行的函数，使用了GCC的的destructor功能。详细信息请参照“[Hack #22]GCC的GNU扩展”和“[Hack #31]在main()前呼叫函数”。在这里，将stack_usage.c进行共享库化。

```
% gcc -fPIC -shared -o stack_usage.so stack_usage.c
```

测定目标程序的 make

作为测定目标的程序，试着选择gzip，将其附上-finstrument-functions来make。

```
% tar xf gzip-1.2.4a.tar && cd gzip-1.2.4a
% CFLAGS=-finstrument-functions ./configure && make
```

__cyg_profile_func_{enter,exit}，由于空的安装包含在 glibc 中，所以即使是不改变 gzip 的源代码也能成功链接 gzip 命令。

测定

将上面的 stack_usage.so 进行 preload 之后并运行 gzip，则以标准输出的方式表示出 stack 的使用量。

```
% LD_PRELOAD=../stack_usage.so ./gzip sample.txt
```

stack 使用量约 716 byte

另外，在测定目标程序为复数线程时，以及想要更正确地测定时，必须要再次使用 stack_usage.c，详细信息请参考“[Hack #75]取得 stack 领域的地址”。作为掌握 stack 使用量的方法，还有“[Hack #66]把握进程和动态库被 map 的内存”中的利用 /proc/<pid>/maps 的方法。

另外一种 hook 方法（LD_AUDIT）

使用版本 2.4 之后的 glibc 时，可通过环境变量 LD_AUDIT，在没有再编译 gzip 等的测定目标的情况下，可以自由地 map（经过了 PLT）所有函数的读出，详细信息请参照 Sun Solaris 的在线说明书《运行时链接器的监查 interface》。在 glibc 也有可以直接适用的内容。

总结

使用了 GCC 的 -finstrument-functions 选项后，便可以在面向函数的

enter/exit 时读出自动生成的函数。活用了这个功能后，可以轻松地运行程序运作的动态解析。

—— *Yusuke Sato*

从 signal handler 中改写程序的 context

介绍了 signal 导致程序被中断时，从 signal handler 中改写被中断方的 context 的方法。

改写 context

发生了 signal 的话，程序的运行能够被中断，控制将转移到 signal handler 上。在终止了 signal handler 的运行后，一般会从被中断的地方再次开始程序的运行。在本 Hack 中，试着从 signal handler 中，改写与运行被中断方的程序相对的状态。具体而言，就是通过改写主要构成的程序计数器，以与被中断的地方不同的场所为起点再次开始运行。

下面表示的是 Linux/x86 用的程序。

```
#include <stdio.h>
#include <signal.h>
#include <asm/ucontext.h>

static unsigned long target;

void handler(int signum, siginfo_t *siginfo, void *uc0) {
  struct ucontext *uc;
  struct sigcontext *sc;

  uc = (struct ucontext *)uc0;
  sc = &uc->uc_mcontext;

  sc->eip = target;
}

int main(int argc, char **argv) {
  struct sigaction act;

  act.sa_sigaction = handler;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGTRAP, &act, NULL);

  asm("movl $skipped,%0" : : "m" (target));

  asm("int3"); /* causes SIGTRAP */
  printf("To be skipped.\n");
```

```
    asm("skipped:");
    printf("Done.\n");
}
```

main()函数将2次调用printf()，若能正常运行，则显示为下面的2行。

```
To be skipped.
Done.
```

但运行了之后，只表示出了在第1行中没有出现的Done。接下来，将发生什么事情呢？让我们从main()开始看。

在调用sigaction(2)之前，设定了signal handler。在这里，在发生了SIGTRAP这种signal时，将调用handler()。接下来的在线汇编代码asm("movl会将label skipped的地址带入到变量target 上,label skipped在这之前被写为asm("skipped:")。

到此准备完毕。接下来装配代码int3 会让SIGTRAP这种signal发生。关于int3请参照“**[Hack #92]**在C程序中设定断点”。

发生了SIGTRAP后，main()运行中的程序的状态即保存在程序里，控制将移到signal handler handler()上。handler()将取得该程序（struct sigcontext型），然后将pointer存放到变量sc中。要注意这个程序的取得方法，会因为OS的不同而不同。

通过改写这个程序，可以从signal handler中变更控制返回后程序的状态。在这里，通过改写包含在程序里的程序计数器，来变更再次开始运行的场所。

handler()将在最后1行中改写eip的值。

```
sc->eip = target;
```

这个eip是x86处理器的程序计数器，是target先前保存的label skipped的地址。也就是说，将从开始再次运行变更为从label skipped的某个地方开始再次运行。

Linux以外的情况

在Linux中，signal handler的第3变量为struct ucontext *型，程序（struct sigcontext*型）将成为其主要部分。在FreeBSD和NetBSD中还有由第3变量制定直接程序的版本。另外，在程序中表示寄存器的member的名称，在Linux中为eip时变成FreeBSD，NetBSD时成为sc_eip。

总结

从 signal handler 中改写被中断方的程序，这里改写的是程序计数器。虽然是用途有限的方法，但可以使用于语言运行系统的例外处理等。

—— *Kazuyuki Shudo*

Hack #79 取得程序计数器的值

在 x86、SPABC、PowerPC 等中，不能像访问一般的寄存器那样访问程序计数器（PC）。本 Hack 中，介绍了取得程序计数器值的方法。

subroutine 读出命令的应用

黑客们会经常想要得到程序计数器（PC）的值“[Hack #80 通过自动改写来改变程序的操作”。要改写 PC 的值很简单，可用 jump 命令来执行。但是，大多数情况下，没有用于值的取得以及面向广泛使用的寄存器或存储器的复制的命令。

在这里，例如若是 ARM.，对于 PC 和通用寄存器一样可以访问，所以很轻松。但在其他的大多数架构例如 x86、SPARC、PowerPC、MIPS. 中却不能这样，所以比较麻烦。

下面表示的是在 x86 中将 PC 作为值来取得的在线汇编代码，运行这个代码，popl 命令的地址将会存入到变量 p 里。

```
void *p;

  asm(".byte 0xe8,0,0,0,0\n\t" /* call the following popl insn */
      "popl %0\n\t"
      : "=m" (p));
```

0xe8 是读出 subroutine 的命令 call。由于相对地址的变量的值会变为 0，所以将 popl 命令作为函数读出。要注意 call 命令会将从 call 开始返回后再次开始运行的地址也就是 popl 命令的地址堆积在 stack 上，结果，被 call 继续运行的 popl 命令会将 popl 命令自身的地址从 stack 中开始 pop。

或者，也可以像以下这样使用 1 这个 label，call 1f 会 call 存在于前面的 1 这个 label。

```
void *p;
asm("call 1f; 1: popl %0" : "=m"(p));
```

在其他的架构中也可以使用同样的方法来取得 PC 的值。例如在 PowerPC 中，接着 bl 命令的 mflr 命令，在 MIPS 则通过 jal 命令，可将 PC 的值复制到通用寄存器上。

总结

在 x86 等架构中，不能像访问一般的寄存器的那样访问程序计数器（PC）。介绍了在这些处理器中取得 PC 的值的方法。

—— *Kazuyuki Shudo*

通过自动改写来改变程序的操作

介绍了在当前UNIX系的OS上可以实现自动改写的程序，还解释了为了实现这个操作而必须的技巧和注意事项。在现代的UNIX系OS上实践以前为了节约内存而经常运行的自动改写。

可改换自身操作的程序

列举了可以通过改写自身的代码来改变操作的函数，下面的为Linux/x86用的代码。

```
#include <stdio.h>
#include <unistd.h>     /* for sysconf(3) */
#include <sys/mman.h>   /* for mprotect(2) */

void func(void) {
  printf("1st.\n");

  asm("slot:\n\t"
      "nop\n\t"
      "nop\n\t"
      "after_slot:");

  printf("2nd.\n");
  /* allows code modification */
  long pagesize = (int)sysconf(_SC_PAGESIZE);
  char *p = (char *)((long)func & ~(pagesize - 1L));
  mprotect(p, pagesize * 10L, PROT_READ|PROT_WRITE|PROT_EXEC);

  /* modifies func() itself */
  asm(".byte 0xe8,0,0,0,0\n\t" /* call the following popl */
      "popl %%eax\n\t"
```

```
        "addl $0x14, %%eax\n\t"
        "subl $after_slot, %%eax\n\t"
        "shl $8, %%eax\n\t"
        "movb $0xeb, %%al\n\t" /* EB is jmp (2 byte insn) */
        "movw %%ax, slot" : : : "eax");

  printf("3rd.\n");
}

int main(int argc, char **argv) {
  func();
  func();
}
```

函数 main()两次读出了 func()函数，函数 func()调用了 printf()并表示出了信息。若没有在线汇编程序代码（asm(..)）的话，则变为以下的输出。

```
1st.
2nd.
3rd.
1st.
2nd.
3rd.
```

但实际上，不能表示出第 2 次的[2nd]，即使是 3 次以上读出了 func()，也不会两次表示出[2nd.]。

func()在初次运行时进行了自动改写，根据这个改写，在第 2 次以后的运行中要 skip printf("2nd.\n")。

说明

来依次看看 func()的处理。1st. 表示后面的 nop 命令是什么都不做的 x86 命令。在这之后，这个 2 字节的 nop 将在其他命令上被写出。

表示出了 2nd 之后，在到 mprotect(...)为止的 3 行中，可以自动改写 func()的代码。关于 mprotect(2)请参照“**[Hack #34]**运行放置在 heap 上的代码”。

跟随着 mprotect(...)的汇编程序代码是关键。跟前面所述的 nop 一样，要编写像浏览 2nd 的表示的 jump 命令，这样一来，在这之后 func()的运行中，便不会表示出 2nd。

这个汇编程序代码通过最开始的2个命令来取得存放这个代码自身的内存空间的地址。

```
.byte 0xe8,0,0,0,0
popl %%eax
```

这个命令列将popl命令自身的地址存放在了EAX.寄存器中，关于这一点请参照“**[Hack #79]**取得程序计数器的值”。

此后，以popl命令的地址为基础进行多种操作，从而算出从after_slot中发现的print("3rd.\n")的相对地址。然后，将接下来的2byte覆盖为前面所述的nop命令的2 byte。

0xeb 这个相对地址

0xeb为jump命令，这个跳跃的跳转目标为print("3rd.\n)。这样在有nop 的地方被运行了的话，则要跳跃到表示3rd.的行上去。

自动内存改写

必须要注意在复数线程的程序中执行改写代码的情况，若其他的线程运行了在改写的过程中半途而废的代码的话，则会发生危险，运行了处于运行状态的代码的线程会异常终止。

实际上上述的程序也不能说在复数线程的程序中会安全。在2byte的nop倒霉地跨过了cache line（例如32和64byte）后，便会留下从其他的线程中可以看见正在改写中的代码这样的危险。在这个程序中，使用xchg命令来代替movw命令来执行改写的话则就安全得多。

从其他的线程中不能看见运行中的状态，这样的处理被称为atomic。什么样的编写方法是atomic呢？这会因处理器种类的不同而不同。例如在相同x86处理器中，也有Pentium III 和Pentium 4 的区别。详细信息有必要通过处理器penta拿出来的技术文件来确认。

命令pipeline和cache的影响

近年的处理器具备了命令pipeline。因此，在从cache中读出了机器语言命令后，在运行该命令之前，会有几个clock周期的间隔。即使是改写了内存上的命令，在处理器完全将这个命令读出的情况下，运行的命令可能会变为原来改写前的命令。这些操作会因处理器的不同而不同。

另外，在改写了memory上的命令后，即使是改变了数据cache上的值，也不能改写命令cache上带有的全部值的处理器，在这种情况中，也将运行改写前的命令。

总结

介绍了在现代的UNIX系OS上可以自动改写的程序，以及必要技巧和注意事项。

—— *Kazuyuki Shudo*

使用 SIGSEGV 来确认地址的有效性

活用SIGSEGV，检测地址的有效性。在signal nandler上配合sigsetjmp(3)/siglongjmp(3)，能完成检测。

本Hack是反过来有效地利用作为“程序的天敌”的SIGSEGV的处理，可确认某个地址所指定的内存领域是否有效。

在signal handler上捕捉SIGSEGV

在想要对没有访问权限的内存领域进行不正确的访问时，会发生SIGSEGV这个signal。通常，SIGSEGV的发生则意味着程序上有问题，接收到它的程序会表示出"Segmentation fault"并异常终止。

在这里反过来积极活用SIGSEGV来检测地址的有效性，通过让signal handler将sigsetjmp(3)/siglongjmp(3)进行组合并使用它，将这个检测变为可能。

下面程序中的函数validate(void *addr)将检测作为变量被给予的地址是否有效，有效则为真，无效则返回假。

```
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>
#define TRUE 1
#define FALSE 0

static struct sigaction orig_act;
static sigjmp_buf env;

static void sigsegv_handler(int sig) {
  siglongjmp(env, 1);
}

int validate(void *addr) {
  int is_valid = FALSE;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  struct sigaction a
```

```
  act.sa_handler = sigsegv_handler;
  sigaction(SIGSEGV, &act, &orig_act);

  if (sigsetjmp(env, TRUE) == 0) {
    /* touch */
    volatile char c;
    c = *((char*)addr); /* read */
    *((char *)addr) = c; /* write */

    is_valid = TRUE;
  }
  else {
    is_valid = FALSE;
  }

  sigaction(SIGSEGV, &orig_act, NULL);

  return is_valid;
}

int main(int argc, char **argv) {
  int a;

  printf("variable a: %s\n", (validate(&a) ? "valid" : "invalid"));
  printf("100 : %s\n", (validate((void *)100) ? "valid" : "invalid"));
}
```

将其运行的话，会表示为下面的2行。表示的是函数main()中的本地变量a 的地址有效，以及理论地址100无效这样的运行结果。

```
variable a: valid
100       : invalid
```

说明

一般的方法，就是试着访问从而检测是否发生了SIGSEGV。即便是发生了SIGSEGV之后也不会让程序异常终止，在信号句柄上捕捉到SIGSEGV，能捕捉到SIGSEGV则从信号句柄中的返回，这样程序会陷入无限的循环中，因此，要使用siglongjmp(3)来返回控制。下面，来看看 validate()的处理。首先，用sigaction(2)在发生了信号SIGSEGV时设定作为信号句柄被读出的函数sigsegv_handler()。

接下来，调用sigsetjmp(3)，保存运行中的状态即信号屏蔽字。由于在保存之后sigsetjmp(3)会返回为0，所以继续被运行的是面向当作参数被给出的地址addr的地址的访问。addr将读出所指示的内存领域。在这里，addr若是有效的，则会成功进行访问，validate()会返回为真。若addr

不是有效地址时，则会由访问引起SIGSEGV的发生。这样一来，对SIGSEGV的控制则会转移到作为信号句柄的设定的 sigsegv_handler()上。

sigsegv_handler()将运行 siglongjmp(env, 1)，由此，先前保存在sigsetjmp(3)上的信号屏蔽字则会立即返回到读出sigsetjmp(3)的地方。此时，作为 siglongjmp(3)的第 2 变量的 1 被当作 sigsetjmp(3)的返回值被返回，所以函数 validate()返回为假。

这样一来便可清楚地将有效的地址返回为真，无效地址返回为假。

要注意前面提到的程序中的 validate()函数，由于将信号屏蔽字等保存在static变量里，故不是线程安全。

总结

介绍了使用 SIGSEGV 来确认地址的有效性的方法。

—— *Kazuyuki Shudo*

用 strace 来跟踪系统调用

用strace可以实现跟踪系统调用，对调试程序和把握操作很有帮助。

strace(http://sf.net/projects/strace/)是跟踪系统调用的Unix工具，从1991用于SunOS开始开发。除了SunOS之外，还可以在FreeBSD 和Linux 中使用。

strace 的方法

strace在 Debian GNU/Linux 中可用 sudo apt-get install strace来安装。

strace的使用方法很简单，一般而言只需排列想要 trace 到 strace命令的变量上的命令和该变量就OK了。在缺省中strace的信息将被标准错误输出，在文件输出时要用 -o 选项。例如，像以下这样来运行。

```
% strace -o log.txt emacs
```

在这个例子中，正在将 emacs 运作中系统调用的读出进行 trace。在macs完成了之后发现了 log.txt，会很清楚地知道 emacs 读出了怎样的系统调

用。例如，在emacs打开的文件中，为了一览包含在文件名中的/home/satoru要执行以下运行。

```
% grep open log.txt | grep /home/satoru
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
  :
```

查询程序打开的文件是strace的一种典型的使用方法。在“明明建成了设定文件，为什么没有反映出来？难道是正在被读取吗？”时，使用strace可以简单地确认是否用程序来打开这个设定文件。

strace的组成

strace 通过运用OS提供的调试用的interface来将系统调用trace，在Linux中使用ptrace系统调用。strace通常会用 ptrace(PTRACE_SYSCALL, ...)hook系统调用的入口和出口，从而输出系统调用的参数和返回值。

strace为了用易懂的方式表示出系统调用的参数和返回值花费了不少力气。例如上面的例子中相对于open调用，像以下这样表示出了作为参数的文件名和flag。

```
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
```

在strace中为了实现这个操作，要准备主要的系统调用、专用的表示routine。在open中第一参数为路径名，第二参数是用int的整数表示的flag，以被O_RDWR和O_LARGEFILE等常量的理论和传送的规格来安装表示routine。O_RDONLY|O_LARGEFILE在简单地用整数表示flag时会变为32768。

总结

使用了strace后可以trace系统调用，这在控制程序的调试和操作上很有用。

补充

将strace进行strace后便可以知道strace正在运用ptrace系统调用。

```
% strace -o log strace -o log2 date
% grep ptrace log |head -5
ptrace(PTRACE_SYSCALL, 8744, 0x1, SIG_0) = 0
ptrace(PTRACE_PEEKUSER, 8744, 4*ORIG_EAX, [0x7a]) = 0
```

```
ptrace(PTRACE_PEEKUSER, 8744, 4*EAX, [0xffffffda]) = 0
ptrace(PTRACE_PEEKUSER, 8744, 4*EBX, [0xbfffe45c]) = 0
ptrace(PTRACE_SYSCALL, 8744, 0x1, SIG_0) = 0
```

—— *Satoru Takabayashi*

用 ltrace 来跟踪进程调用共享库的函数

使用 ltrace 后，可以让跟踪共享库的函数的调用变成可能，同时在调试中有更好的把握。

ltrace是调用共享库的函数并进行trace的Linux用的工具。和将系统调用trace的strace相同，在调试中非常有用。关于strace请参照“**[Hack #82]** 用 strace 将系统调用 trace”。

ltrace 的使用方法

ltrace 在 Debian GNU/Linux 中可用 sudo apt-get install ltrace 来安装。

ltrace 的使用方法很简单，一般来说只须排列想要 trace 到 ltrace 名的参数上的命令和该参数就 OK 了。在缺省中 ltrace 被标准错误输出，在文件输出时要用 -o 选项。例如：进行如下运行。

```
% ltrace -o log.txt wget https://www.codeblog.org/
```

在这个例子中wget会trace在 取得*https://www.codeblog.org/*的contents时的共享库的读出。wget 用 OpenSSL 的 libssl.so 来进行与 https 的通信。很明显 ltrace 在将输出的 log.txt 在 SSL 上进行 grep 后，会读出什么样的函数。

```
% grep SSL log.txt | head
SSL_library_init(0, 0, 0, 0, 0)                    = 1
SSL_load_error_strings(0, 0, 0, 0, 0)              = 0
OPENSSL_add_all_algorithms_noconf(0, 0, 0, 0, 0) = 1
SSL_library_init(0, 0, 0, 0, 0)                    = 1
SSLv23_client_method(0, 0, 0, 0, 0)                = 0x40038880
SSL_CTX_new(0x40038880, 0, 0, 0, 0)                = 0x808b228
SSL_CTX_set_verify(0x808b228, 0, 0x8068585, 0, 0) = 0x8068585
SSL_new(0x808b228, 0x7a060ed3, 1, 0, 0)            = 0x808cd20
SSL_set_fd(0x808cd20, 3, 1, 0, 0)                  = 1
SSL_set_connect_state(0x808cd20, 3, 1, 0, 0)       = 0
SSL_connect(0x808cd20, 3, 1, 0, 0
```

用了“-p”选项后可以与既存进程接触。

ltrace的组成

ltrace的组成与调试器相似。ltrace和调试器一样，将软件的断点嵌入在成为trace对象的进程中。像上面这样运行了ltrace wget ...时，ltrace会执行如下操作。

1. 经过环境变量PATH来检测wget二进制的绝对路径。(在笔者的环境中/usr/bin/wget)。
2. 用elfutils来读出依存于/usr/bin/wget的二进制的所有共享库，并取得函数的符号名和它的PLT里的地址的目录。
3. 进行fork之后在子进程的内部设置ptrace(PTRACE_TRACEME, ...)，然后将wget进行exec。
4. SIGTRAP将传递到用wait()等待的母进程上。
5. 在母进程中将先前生成的目录还原，然后在每个函数的PLT的适用地址上写出break point (x86中为0xcc)。
6. 由于子进程在每次读出共享库的函数会发生SIGTRAP，所以母进程会在loop内用wait来等待SIGTRAP，进而一边用适当的断点输出trace，一边重复loop一直到子进程终止为止。

母进程在用断点输出了trace之后，会将带有断点命令的地址的值还原，在ptrace (PTRACE_SINGLESTEP, ...)中只用1命令来运行子进程。1命令的运行完成之后，控制会再次返回到母进程上，从而将断点复原。

在PLT (Procedure Linkage Table) 中，会为每个函数准备在读出ELF的共享库的函数必须经过的极短的代码。ltrace会通过在PLT上书写断点，将hook共享库的函数调用。

总结

使用了ltrace之后可以trace共享库的函数读出，这在控制程序的调试和操作时非常有用。

—— *Satoru Takabayashi*

用 Jockey 来记录、再生 Linux 的程序运行

使用 Jockey 能够记录、再生 Linux 的程序运行。Jockey 可以用于调试再现性比较低的 bug。

Jockey 是记录、再生 Linux 的程序运行的工具。通过将系统调用和一部分的 CPU 命令 hook 后将运行时的输入与输出记录在 log 上来实现程序的再生。主要用于调试用途。

安装

由于 Jockey 没有包含在 Debian package 中，所以从源代码中进行了 build、安装。必须在事先安装 ruby、boost、zlib。

使用方法

使用 Jockey 来记录、再生程序的运行的方法有很多种，但最简单的是使用 jockey 命令的方法。例如，在记录 /bin/date 的运行时会变成以下这样。

```
% jockey /bin/date
Warning: /bin/date is, by default, excluded from tracing.
Warning: I'm adding 'excludedprogram=-' option as a courtesy.
2006年1 月23 日星期一01:23:39 JST
```

在注意到警告时，若将--excludedprogram=-传送到 jockey 的参数上则会消失，像下面这样来运行再生记录在上面的 /bin/data 的运行。

```
% jockey --replay=1 /bin/date
2006年1 月23 日星期一01:23:39 JST
```

无论进行多少次再生，都会在与先前完全相同的时刻表示出来。这是因为在记录 /bin/data 时，系统调用 gettimeofday(2) 的值被 log 写出，并用到了再生时记录在 log 里的值，在 log 里也记录了其他许多信息。

使用 Jockey 可以记录并再生使用了 socket 的程序。

```
% jockey wget -qO- 'http://www.random.org/cgi-bin/randnum?num=5'
17      81      81      38      18
% jockey --replay=1 wget -qO- 'http://www.random.org/cgi-bin/randnum?num=5'
17      81      81      38      18
```

在这个例子中，通过使用 random.org 的 random.org 的 service 来取得 5 个随机的整数，会得到与再生时的记录时间完全相同的结束。再生时不进行实

际的网络访问，将log还原从而再现数据的输入。没有发生网上访问，所以可以高速再生。

组成

Jockey的本体为libjockey.so这一共享库。因指定到LD_PRELOAD上来再生并记录任意的程序，所以不需要源代码的编码和再链接等。上面实验中用到的Jockey命令适当地设定了环境变量LD_PRELOAD和Jockey的rapper。

Jockey在运行时会改写系统调用进行非决定性的运作的一部分CPU命令（当前为rdtsc）的读出部分的代码。然后呼出可再生记录log的Jockey准备的代码而不是呼出原来的系统调用和CPU命令。为了实现这个操作将用到x86命令的tablebase的parser和libdisasm（*http://bastard.sourceforge.net/libdisasm.html*）这个disassembler库。除此之外的部分按原来的代码进行。

详细的组成在Jockey的论文（*http://www.ysaito.com/f10-saito.pdf*）中有解说。map(2)和处理signal的方法是非常有趣的内容。

总结

使用了Jockey后便可以记录、再生任意Linux程序的运行。Jockey可在调试再现性低的bug时使用，即使是10回中只再生1次的bug，若用Jockey记录到问题再现为止的话，发生了问题的bug便可再现多次，在GDB上执行再生。

进行这种调试的情况可能不是很多，但记住它总会有用，即使是没有用也有技术上的乐趣，所以记住它不会有损失。

—— *Satoru Takabayashi*

HACK #85 用prelink将程序启动高速化

使用prelink能够缩短连接大量的共享库文件的程序启动时间。

使用prelink能够缩短链接了大量共享库的程序的启动时间。最近能用达link上，从《Gentoo Linux Prelink》（*http://www.gentoo.org/ doc/ja/prelink-howto.xml*）中可知“可以缩短典型的KDE程序的启动时间的50%”。在本

Hack中介绍了 prelink的基本使用方法之后会通过编写简单的程序来执行检测 prelink 的效果的实验。

prelink

通常，在启动动态链接后的程序时，要进行重定位以及符号的 look up。Prelink会改写依存于可执行文件的共享库，并尽量减小启动时执行这些处理的必要性。另外，由于被 prelink 过的程序会通过重定位来减小不可共享的页面，所以在进程间被共享的页面则会增大。

Prelink 的使用方法

在 Debian GNU/Linux 中使用了 prelink 的 package 后，便可设定用 cron 在暗中自动进行 prelink。但是，由于在缺省中不能进行 prelink，所以要将 /etc/default/prelink的PRELINKING=unknown修正为PRELINKING=yes。在系统全体可手动运行 prelink 时运行状况如下。

```
% /usr/sbin/prelink -amR
```

执行命令，将进行对于/etc/prelink.conf所设定的目录里所有可执行文件和共有库的 prelink 的处理。（/usr/bin、/usr/lib 等等）-a 是表示允许所有的，-m 则是允许同时未使用的地址的重复。R 表示进行地址的随机化，提高安全保障的含义。

要使 prelink 适用于特定的二进制，而不是适用于整个系统，要执行以下操作。

```
% prelink -N a.out liba.so libb.so
```

这个例子中，prelink 适用于可执行文件 a.out，a.out 所依存的共享库 liba.so和libb.so。-N 表示没有更新 /etc/prelink.cache。

测定 prelink 的效果

为了测定 prelink 的效果将进行以下实验：

1. 生成 1,000 个 .c 文件。各个文件中包含有函数 int func00XXX() { return 0; }。
2. 通过 gcc-shared 编译上述的 .c 文件，制作 1,000 个 .so 文件。
3. 把上述的 1,000 个 .so 文件链接到空的 main() 程序上。

4. 计算完成二进制的运行时间。

5. 计算 prelink 后二进制的运行时间。

6. 计算通过静态链接制成的二进制的运行时间。

结果如下：

prelink 前	prelink 后	静态链接
约 4.5 秒	约 0.5 秒	约 0.0 秒

实验从 command line 开始运行如下：

```
# 把prelink安装到Debian
% sudo apt-get install prelink

# 文件生成和编译
% time ruby prelink-test.rb

# prelink前
% repeat 3 time ./test-dynamic
4.37s total : 4.22s user 0.15s system 100% cpu
4.62s total : 4.41s user 0.19s system 99% cpu
4.45s total : 4.28s user 0.17s system 99% cpu

# 适用prelink
% time prelink -N ./test-dynamic *.so
7.06s total : 5.52s user 1.85s system 104% cpu

# prelink后
% repeat 3 time ./test-dynamic
0.51s total : 0.47s user 0.04s system 100% cpu
0.46s total : 0.43s user 0.03s system 99% cpu
0.49s total : 0.46s user 0.03s system 99% cpu

# 静态链接
% time ./test-static
0.00s total : 0.00s user 0.00s system 0% cpu
```

prelink-test.rb 的代码如下：

```
system("rm -f *.c *.so")
File.open("test.c", "w") {|f|
  f.puts('int main() { return 0; }')
}

objs = []
dsos = []
1000.times {|i|
  c_file_name = sprintf("%05d.c", i)
  puts c_file_name
  File.open(c_file_name, "w") {|f|
    # f.printf('const char *s%05d = "%s";' + "\n", i, "o" * (1<<20));
```

```
    f.printf("int func%05d() { return 0; }\n", i);
  }
  obj_file_name = sprintf("%05d.o", i)
  dso_file_name = sprintf("%05d.so", i)
  system(sprintf("gcc -c %s", c_file_name))
  system(sprintf("gcc -fPIC -shared -o %s %s", dso_file_name, c_file_name))
  objs.push(obj_file_name)
  dsos.push("./" + dso_file_name)
}

system(sprintf("gcc -o test-static test.c %s", objs.join(" ")))
system(sprintf("gcc -o test-dynamic test.c %s", dsos.join(" ")))

# prelink -N test-dynamic *.so
```

总结

可以看到，使用 prelink 可以缩短链接了大量共享库的程序的启动时间。在 OpenOffice.org 和 Firefox 等大规模实际处理中，prelink 的效果更大。尤其使在 C++ 中，符号名变长，由于存在持有通用的长的 prefix 的倾向，减少符号 look up 的效果会大些。

此外，在笔者的环境中（i386 上的 Debian GNU/Linux sarge）prelink 在 0x41000000～0x50000000 的 240MB 部分的虚拟地址空间里设计共享库。因此，如果 prelink 240MB 中无法容纳共享库，则会产生如“Could not find virtual address slot for ./foo.so”的错误。

参考文献

- prelink 的论文（*ftp://people.redhat.com/jakub/prelink/prelink.pdf*）
- GNU C Library Version 2.3（*http://people.redhat.com/drepper/lt2002talk.pdf*）有导入 prelink 经过的说明。
- How to Write Shared Libraries（*http://people.redhat.com/drepper/dsohowto.pdf*）有关于动态链接的性能的详细解说。

—— *Satoru Takabayashi*

通过 livepatch 在运行中的进程上发布补丁

为了给在运行中的进程里打补丁的 livepatch 的使用方法进行解释。

在本Hack中，介绍在运行中的进程上打补丁的livepatch的使用方法和原理。

所谓 livepatch

livepatch（*http://ukai.jp/Software/livepatch/*）通过变更执行中的 user process 的代码和数据，不中断运行中的用户进程来打补丁的程序。

livepatch 把 target process 的 process ID 及 target process 没有 strip 的执行二进制带到参数中，再从标准输入中读取如下的 patch 命令，打上补丁。

- `set` 地址类型值

  在 target process 的地址所示的内存中，输入类型所解释的值。

- `new` 的存储名大小

  在 target process 中确定 size byte 部分的内存，还要加上“存储名”。

- `load` 存储名 文件名

  把文件名所显示的文件的内容加载到 target process 中，再在内存中加上“存储名”。

- `dl` 存储名 文件名

  把文件名所示的共享对象加载到 target process 中，在内存中加上“存储名”。解释共享对象的符号信息，再和 target process 进行动态链接。

- `jmp` 地址 1 和地址 2

  在 target process 的地址 1 中，输入跳到地址 2 的命令。

地址的制定如下：

- 整数值

  指向 target process 中的虚拟地址。

- `$` 存储名

  使用`new.load.dl`所制定的存储名，指向通过这些命令确保的target process 的内存空间。

- `$` 存储名：符号

  使用 `dl` 所制定的存储名，形成更改共享目标中 target process 的符号的值。

- `$` 存储名：整数值

指向 new.load.dl 所确保的内存空间的整数值 offset 的地址。

- 符号名

  形成 target process 的符号的值。

在 set 中可使用的类型有以下几个：

- int

  值为整数值，用 strtol(value, NULL, 0) 解释

- str

  值为改行前的字符串

- hex

  值为 16 进制值

- addr

  值为地址

livepatch 的例子

下面介绍一个简单的例子。

首先将 target 进程当作下面这种单纯的无限的 loop 的程序。

```
% cat -n target.c
     1 #include <stdio.h>
     2 #include <sys/types.h>
     3 #include <unistd.h>
     4
     5 char *
     6 foo(int i)
     7 {
     8         static char buf[16];
     9         sprintf(buf, "%d", i);
    10         return buf;
    11 }
    12
    13 int n;
    14 char fmt[] = "foo->%s\n";
    15
    16 int
    17 main(int argc, char *argv[])
    18 {
    19         n = atoi(argv[1]);
    20         printf("pid=%d\n", getpid());
```

```
   21          while (1) {
   22                  int i;
   23                  for (i = 0; i < n; i++) {
   24                          printf(fmt, foo(i));
   25                          sleep(1);
   26                  }
   27          }
   28 }
% cc -o target target.c
```

将它运行后，会将数字增加到参数的数字为止并将其按行表示出来。

```
% ./target 3
pid=5059
foo->0
foo->1
foo->2
foo->0
foo->1
```

这个目标进程的进程ID为5059，接下来运行livepatch。

```
% echo 'set n int 5' | ./livepatch 5059 ./target
```

这是针对进程ID 5059的target的进程，再用符号n指定的地址上书写数值5的意思。这样一来运行中进程ID 5059的进程的输入会变成如下这样。

```
foo->0
foo->1
foo->2
foo->0
foo->1      # <用- livepatch补丁后将n变为5分配
foo->2
foo->3
foo->4
foo->0
foo->1
```

像这样本来n为3，但根据livepatch将它变成5。下面程序中不想变更foo。

```
% cat -n foo2.c
    1 #include <stdio.h>
    2
    3 extern int n;
    4
    5 char *
    6 foo(int i)
    7 {
    8         static char buf[16];
    9         sprintf(buf, "0x0%x", n - i);
   10         return buf;
```

```
    11 }
%
```

进行接下来的操作并选择共享目标。

```
% cc -shared -fPIC -o foo2.so foo2.c
```

如以下一样运行 livepatch commomd（命令）。

```
% echo 'dl foo2 ./foo2.so
jmp foo $foo2:foo' | ./livepatch 5059 ./target
dl foo2 @ 0xb7fe6000 [6368] ./foo2.so
jmp 0x8048454 0xb7fe6714
```

下载 foo2.so 共享目标，命名为 foo2，再链接到 target process 在 traget process 的 foo 里，把 jump 命令写入 foo2 共享目标里的 foo symbol。也就是说，在源 target progessde foo()的 entry（输入）里，存入新下载的 foo2.so 中的 foo()的 jump。由此，原来的 foo()变成为 foo2.so 的 foo()的运行。也因此，运行中的 process ID 5059 会如下一样改变输出。

```
foo->0
foo->1
foo->2
foo->0x02      用# livepatch改变foo()。
foo->0x01
foo->0x05
foo->0x04
foo->0x03
foo->0x02
foo->0x01
```

livepatch 的原理

livepatch命令不需要特殊的kernel（内核）。使用debug的ptrace(2)进行系统调用，进行target process内存的读写。并且，用libbfd可识别target命令和共享目标的符号表进行链接。

ptrace(2)的使用方法

用 ptrace(2)来进行 target process 的内存读写。首先必须达到目标进程。要实现attach，需要对target process的processID使用 PTRACE_ATTACH。目标进程接受 ptrace request 并用 wait(2)等待它的停止。

```
680     if (ptrace(PTRACE_ATTACH, target_pid, NULL, NULL) < 0) {
681         perror("ptrace attach");
682         exit(1);
```

```
683     }
684     DEBUG("attached %d\n", target_pid);
685     wait(NULL);
```

若attach，用PTRACE_PEEKDATA和PTRACE_POKEDATA进行目标进程的内存的读写。

```
448     int *lv = malloc(len * sizeof(int));

455         lv[i] = ptrace(PTRACE_PEEKDATA, pid,
456                        addr0 + i * sizeof(int), NULL);
```

这种情况，可读出由pid显示得target process的内存addr0 + i * sizeof(int)里的int值。

```
469         if (ptrace(PTRACE_POKEDATA, pid,
470                    addr0 + i * sizeof(int), lv[i]) < 0) {
471             perror("ptrace poke");
472             return -1;
473         }
```

因此，输入在由pid显示的目标进程的内存addr0 + i * sizeof(int)里输入由lv[i]表示的值。在进行内存的读写之外还可以进行目标进程中的寄存器的读写。

```
362     struct user_regs_struct regs, oregs;

367     if (ptrace(PTRACE_GETREGS, pid, NULL, &oregs) < 0) {
368         perror("ptrace getregs");
369         return 0;
370     }
371
372     regs = oregs;
```

因此在oregs里读出由pid显示的目标进程的寄存器。在x86架构的情况下，用regs.esp可读出%esp，用regs.eax可读出%eax等。输入时用以下方式。

```
412     if (ptrace(PTRACE_SETREGS, pid, NULL, &regs) < 0) {
413         perror("ptrace set regs");
414         return 0;
415     }
```

也可在目标进程里运行编码。

```
417     if (ptrace(PTRACE_CONT, pid, NULL, NULL) < 0) {
418         perror("ptrace cont");
419         return 0;
420     }
421     wait(NULL);
```

在 PTRACE_CONT 之前，用 PTRACE_SETREGS 指定 regs.eip 里下一个将运行的编码的初始地址，则它里面的编码开始运行。并且，隐藏这些编码里的 break point 命令 int3、0xcc。这时目标进程将停止。Attach 上的 progess 由于 wait 引起等待，处理持续进行。用最后的 PTRACE_DETACH 进行 detach。

```
878     ptrace(PTRACE_DETACH, target_pid, NULL, NULL);
879     DEBUG("detached %d\n", target_pid);
```

用 ptrace(2)确保内存领域

仅用 ptrace(2) 无法在目标进程里完成新一轮内存的确保。那么，livepatch 的 new、load、dl 命令将如何运行呢？

实际上，使用之前介绍的 ptrace，把以下这样的编码传送到目标进程里进行运行。

```
mmap(NULL, siz, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
```

在 stack 里设定变量在 %eax 里设定系统调用码 SYS_mmap，然后只要运行 int $0x80 这个编码就可实现。若在 int $0x80 的后面添加 int3，在从 int $0x80 返回的时候，遇到 break point 且目标进程停止。控制返回到 progess 处。这时，由于 mmap(2) 确保的内存空间的初始的地址作为 mmap(2) 的 return（返回值）被 %eax 设定。读出这个就可知道在目标进程哪个地方被 mmap(2)。

外部动态的链接

livepatch 从目标进程的外面进行动态链接。

首先，读出 /proc/target process 的 PID/maps。调查共享库在哪个地址被 map。/proc/$$/maps 将变为以下的 format。

```
% cat /proc/5059/maps
08048000-08049000 r-xp 00000000 03:01 5081628    /tmp/target
08049000-0804a000 rw-p 00000000 03:01 5081628    /tmp/target
b7ea3000-b7ea4000 rw-p b7ea3000 00:00 0
b7ea4000-b7fce000 r-xp 00000000 03:01 36834      /lib/tls/i686/
cmov/libc-2.3.2.so
b7fce000-b7fd7000 rw-p 00129000 03:01 36834      /lib/tls/i686/
cmov/libc-2.3.2.so
b7fd7000-b7fd9000 rw-p b7fd7000 00:00 0
b7fe8000-b7fea000 rw-p b7fe8000 00:00 0
b7fea000-b8000000 r-xp 00000000 03:01 2277452    /lib/ld-2.3.2.so
b8000000-b8001000 rw-p 00015000 03:01 2277452    /lib/ld-2.3.2.so
```

```
bfffe000-c0000000 rw-p bfffe000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

各行都含有以下这样的信息。

```
vm_start vm_end  flags pgoff  major:minor ino  文件名
```

然后将target程序以及它利用的共享库通过libbfd读出被定义的符号，以及将其值用bfd_canonicalize_symtab()和bfd_canonicalize_dynamic_symtab()读出。然后根据被map的地址变换为在目标进程上的内存地址。

用dl命令来加载共享目标时，必须将这个共享目标的未定义符号与目标进程的符号值绑定然后动态链接。由于可以像前述的那样得到目标进程的符号的值，所以之后要改写共享目标relocation table(rel.dyn .rel.plt)。

总结

可通过使用调试用的系统调用ptrace(2)来变更运作中进程的存储器和寄存器，介绍了利用这一点而开发的为了将二进制补丁分配到运行进程上的程序livepatch 。

—— *Fumitoshi Ukai*

第6章

profile 调试器 Hack

Hack #87～92

本章介绍的是profile和调试器的相关技术，随着CPU和存储器性能的提高，电脑的性能也有了大幅度的提高，但同时，即使有强大的性能，在处理不断扩大的Web站点的营运、数据库的更新与维护，以及动画处理等方面，仍不完备。编写高速的程序对于编程员来说，仍然是一个重要的课题，另外就更不必说编写无缺陷的程序是编程员最重要的课题。

本章中，首先针对在开发高速程序方面有用的模式的基本使用方法以及架构作了介绍，其次，则是对在调试器使用方面有用的技术进行介绍。

使用 gprof 检索 profile

profiling 是指检测在程序的哪个部分上处理时要花费较多时间。本 Hack 中，介绍了通过 gprof 来检索 profile 的方法。

在普通的程序中，性能上的瓶颈部分多为代码，将为了计算出程序的相对较慢的部分，并检测在程序的哪个部分花费时间称作 profiling。本章的 Hack，介绍的是使用 gprof 检索 profile 的方法。

使用方法

尝试检索下列简单程序的模式

```
void slow() {
    int i;
    for (i = 0; i < 2000000; i++);
}
```

```
void f() {
    int i;
    for (i = 0; i < 1000; i++) slow();
}

void g() {
    int i;
    for (i = 0; i < 4000; i++) slow();
}

int main() {
    f();
    g();
}
```

像以下这样将程序编译并使用 gprof。

```
% gcc -O -g -pg bench.c
% ./a.out
% gprof a.out
```

注意对 gcc 加上 -pg 选项输出 profile 信息。若是执行 /a.out，gmon.out 文件上的 profile 信息会被标出。如下图所示的是 gprof a.out 输出的部分程序

```
%    cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
100.00    25.86    25.86     5000     0.01     0.01  slow
 0.00     25.86     0.00        1     0.00     5.17  f
 0.00     25.86     0.00        1     0.00    20.69  g

index % time    self  children    called         name
                5.17    0.00    1000/5000         f [4]
               20.69    0.00    4000/5000         g [3]
[1]    100.0   25.86    0.00    5000           slow [1]
-----------------------------------------------
                                               <spontaneous>
[2]    100.0    0.00   25.86                  main [2]
                0.00   20.69       1/1           g [3]
                0.00    5.17       1/1           f [4]
-----------------------------------------------
                0.00   20.69       1/1        main [2]
[3]     80.0    0.00   20.69       1          g [3]
               20.69    0.00    4000/5000     slow [1]
-----------------------------------------------
                0.00    5.17       1/1        main [2]
[4]     20.0    0.00    5.17       1          f [4]
                5.17    0.00    1000/5000     slow [1]
-----------------------------------------------
```

第一个表格表示的是每个函数所需的时间。slow 函数的消费时间（self

seconds)，所读出的g几乎都是f的4倍slow次数。总消费时间（total s/call）也大约为4倍左右。

第二个表格（call graph）中，可以搜索出函数的调用关系，如果从main中调用了f和g则会发现f和g正在调用slow。

若gprof上附加-l，则可以检索出每行所需要的时间。这时必须在-g选项中加上调试信息。在附加-l -A -x和源代码的表示方面，可以看出每行所需的时间。想浏览特定函数的call graph的内容时，可以指定gprof -F slow。

使用gcc -pg时，将输入隐藏在每个处理上增加的计数器中，正因如此性能也变得比平常运行时要慢。

原理

通过gcc -pg被输出的二进制文件剩余有3种profile信息。虽然缺省中剩余的是gmon.out文件，但由于GMON_OUT_PREFIX环境变量，该文件的文件名也有可能改变。该文件的信息在《GUN gprof-Implementation》(*http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_node/gprof_25.html*）上有详细说明。

首先，第一行是被time检索过的每十毫秒中的program count的位置。这里使用“**[Hack #31]**执行main()之前的函数”中所介绍的方法，在main前调用setitimer(2)函数，设定每千分之十秒会发生SIGPROF。在用SIGPROF调用的处理器（glibc的sysdeps/posix/sprofil.c的profil_count）中计算计数器。事实上由于使用了glibc的profil(3)，这是可以实现的。

第二行是函数的call graph的信息。这里使用“**[Hack #77]** 将面向函数的enter/exit进行hoot”中所介绍的方法，在进入函数中时可以调用mcount函数。在mount中用架构依存的短的汇编程序（glibc的sysdeps/i386-mcount.S等）得到调用前PC和调用后PC的信息后，读出架构不依存的__mcount_internal。然后使用PC信息，一边编写call graph信息，同时正确记录调用函数的次数。

第三行是已经使用旧的GCC时必要的信息，在每个基本的块（if块和while块等）上逐一记录该块已经出现的次数。该计数器是根据编译器编写的，虽然通过它可以完整地记录平均信息，但由于速度非常慢，时间profile信息可

能会很不准确，现在已经不再记录该信息了。但想要知道平均信息的话，更好的方法是使用gcov。

执行gprof命令时，打开记录这些信息的文件，得到正确的call graph函数的读出次数，还可以得到将采样结果带入原式而计算出的大致所需时间。不仅如此，还可以从使用“[Hack #67]通过libbfd得到符号的一览表”中介绍的方法来得到执行文件里的地址－符号名，并将这些信息进行整理然后表示出来。

总结

本小节介绍的是使用gprof取得profile的方法以及原理。

—— *Shinichiro Hamaji*

使用sysprof搜索系统profile

如果使用sysprof，可以非常便捷的获取系统profile。

本小结介绍的是获得系统整体profile的软件 —— sysprof。

安装

本书在创作时使用的sysprof为1.0.2版本。许可证规定为GPL2。sysprof是组合了Linux内核模块和应用客户端来运行应用的。目前只支持x86和x86-64两种。

首先必须开启Linux内核的profiling支持，若内核的该性能在使用时不被支持的话，请把内核降到2.6.11以下，开启CONFIG_PROFILNG的设定，然后再进行控制。接下来保持启动CONFIG_PROFILNG，同时可以使用“[Hack #89]使用oprofile获取详细系统profile”一节中所介绍的方法，这样会更好。值得注意的是，由于GUI的原因，GTK要降到+2.6.0以下，libglade要降到2.5.1以下。

完成上述操作后，就可以展开sysprof的存档，通过./configure;make;make in进行安装。然后加载作为modprobe sysprof-module等的模块module，安装就结束了。

使用方法

Sysprof的使用方法极为简单，想获取特定程序的profile时，启动sysprof点击开始后启动该profile，最好刚停止时点击Profile钮。下面是与“**[Hack #87]** 使用 gprof 搜索 profile”一节相同的例子，我们不妨来试试。

```
void slow() {
    int i;
    for (i = 0; i < 2000000; i++);
}

void f() {
    int i;
    for (i = 0; i < 1000; i++) slow();
}

void g() {
    int i;
    for (i = 0; i < 4000; i++) slow();
}

int main() {
   f();
   g();
}
```

没有必要添加特别的编译器选项以及调试符号信息，符号不完全拆卸也没问题。

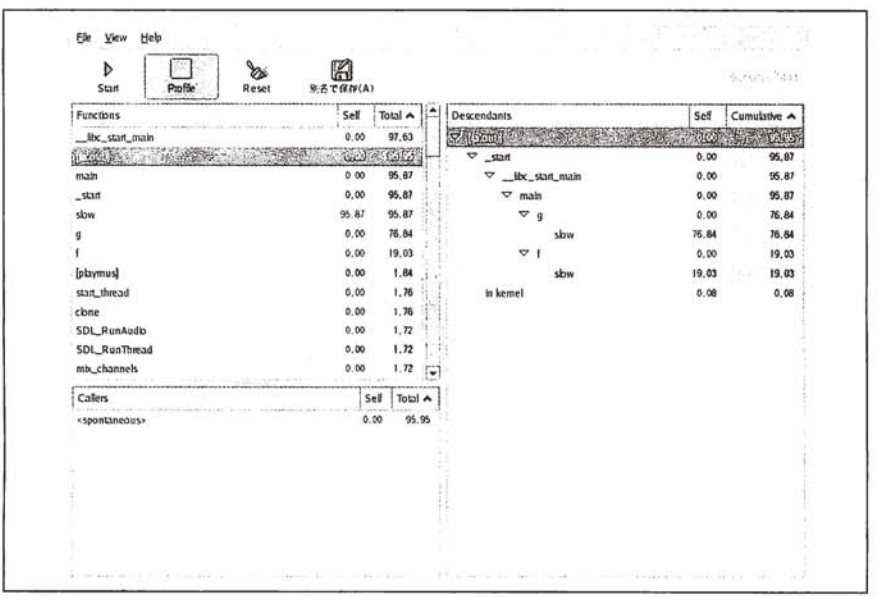


图 6-1：sysprof 运行画面

获得该范例的profile时，这里规定的函数f和g大致是2:8左右，而且此时系统整体的模式正好处于播放音乐的状态，所以也可以看出playmus软件正在进入目录。该模式可以保存在xml文件中。

sysprof是在运行中高频率的采集样本，并记录该时刻pc的位置，所以得出的结果不一定是完全正确的，但在搜索程序瓶颈时十分实用。

总结

本节介绍了使用sysprof获取系统整体profile的方法。如果使用sysprof，可以非常便捷的获取profile。

—— *Shinichiro Hamaji*

HACK #89 使用oprofile获取详细的系统profile

本小节介绍的是获取系统整体模式的软件 —— oprofile。

对于很多CPU，可能oprofile比sysprof更能详细的获取CPU上的信息。“**[Hack #88]** 使用sysprof简单搜索系统profile”一节中介绍的sysprof是功能有限但极易掌握操作的软件。相比之下本节的Hack所介绍的oprofile，虽然需要较为复杂的操作，但很多CPU可以运行，而且可以使用CPU的各种功能，获取各种系统profile。

安装

本书在创作时使用的oprofile为0.9.1版本。oprofile是在GPL2的基础上被发布的。

与sysprof相同，oprofile也是与内核模块和应用客户端相适应的运行软件。根据文档，Linux2.2中的x86、Linux2.4中添加的IA-64、Linux2.6中添加的Alpha、MIPS、ARM、x86-64、SPARC64、PPC64和限定的PA-RISC以及s390被作为范例。

作者将oprofile在Mobile Celeron 1.7Hz、Pentium 4 2.53Hz、Xeon 3.2Hz双处理器三种环境中做了试验，在Xeon中CPU资源全部可以使用，在Pentium 4中可以获取简单的信息，而在Mobile Celeron中则无法获取profile。据oprofile网站的消息，oprofile也有在笔记本电脑中无法运行的情况发生。本节的Hack是用Xeon的机器对运行状况进行确认。

通过对 sysprof 的 Hack 介绍见[Hack #88]，用内核的设定预先开启 CONFIG_PROFILING和CONFIG_OPREPORT，oprofile与平常一样通过 ./configure; make; make in stall 进行安装，完成后安装就结束了。

基本使用方法

Oprofile在获取 profile 前用 opcontrol 进行控制，浏览 profile 结果时则使用 opreport 或 opannotate。虽然也附带着使用 oprof_start Qt 的 GUIfront-end，但根据作者的试验，操作不是很容易，所以本节的 Hack 对于 CUI 的命令内容进行说明。

首先，最好完成基本设定。在最低限度以下，最好预先设定内核的位置（请注意内核移动时被压缩的镜像 vmlinuz 中禁止的事项）和记录的 call graph 的数目。

```
% opcontrol --vmlinux=/usr/src/linux-2.6.15.4/vmlinux --callgraph=20
```

这些设定被记录在 $HOME/.oprofile/daemonrc 中，所以一旦设定后则没有更改的必要。通过下文所示方法，可以启动 demon process oprofiled。

```
% opcontrol --start
```

运行 profile 时使用下列方法。

```
% opcontrol --dump
```

一旦profile的内核发生复位，则点击reset，一旦停止则点击stop，demon 完成时点击 shutdown。

与 gprof 和 sysprof 的 Hack（见[Hack #87]、[Hack #88]）相同，请获取下列代码的 bench mark。

```
void slow() {
    int i;
    for (i = 0; i < 2000000; i++);
}

void f() {
    int i;
    for (i = 0; i < 1000; i++) slow();
}

void g() {
    int i;
    for (i = 0; i < 4000; i++) slow();
}
```

```
int main() {
    f();
    g();
}
```

在oprofiled运行的情况下，尝试下列操作：

```
% gcc -O -g bench.c
% opcontrol --reset
% ./a.out
% opcontrol --dump
```

profile到此被保存在文件中。浏览使用了opreport的profile，虽然支持各种选项，但请看使用了-c的call graph。

```
% opreport -c

samples %        image name       app name         symbol name
-------------------------------------------------------------------------
  5       0.0051 a.out            a.out            main
  19409  19.9324 a.out            a.out            f
  77960  80.0624 a.out            a.out            g
  97374  53.1753 a.out            a.out            slow
  97374  100.000 a.out            a.out            slow [self]
-------------------------------------------------------------------------
  12937   7.0648 libruby.so.1.8.3 libruby.so.1.8.3 (no symbols)
  12937  100.000 libruby.so.1.8.3 libruby.so.1.8.3 (no symbols) [self]
```

与之前相同，可以看出f和g花费的时间比例大约为2:8，也可以看出由于使用了CPU提供的信息，与sysprof相比获得了更多的样例。oprofile与sysprof相同，是获取系统整体profile的软件，所以可以看到如libruby.so.1.8.3等和其他的进程信息。

请看下列使用了opannotate的源代码级的profile信息：在opannotate选项上加上-a，并输出汇编代码，加上-s，并输出汇编代码。源代码表示的调试器信息很重要，不论加-a还是-s，都会变得很简单。

```
% opannotate -a -s
... slow ................. ...
08048348 <slow>: /* slow total: 97374 5.6869 */
                :void slow() {
                : 8048348:      push  %ebp
                : 8048349:      mov   %esp,%ebp
                : 804834b:      mov   $0x1e8480,%eax
                :    int i;
                :    for (i = 0; i < 2000000; i++);
  97361 5.6861 : 8048350:      dec    %eax
      7 4.1e-04 : 8048351:      jne    8048350 <slow+0x8>
                :}
```

```
1 5.8e-05 :                8048353: leave
5 2.9e-04 :                8048354: ret
```

查出缓存失效

如果使用oprofile，不仅得到profile信息，还可以搜索出各类事件的发生次数。在oprofile --list-event中可以看到可搜索事件的list。为了搜索出缓存失效，最好搜索BSQ_CACHE_REFERENCE。下面是搜索事件增加的示例。

```
% opcontrol --event=BSQ_CACHE_REFERENCE:100000:0x100
% opcontrol --shutdown
% opcontrol --start
```

采样率为100000，0x10f称作单位标记，选择捕获的事件。若制定0x10f则应该可以挑出L2缓存失效。现在请搜索如下的简单样例。

```
#define NUM 10000
int main(){
        int a[NUM][NUM];
        int i, j, sum;
        for (i = 0; i < NUM; i++) {
                for (j = 0; j < NUM; j++) {
//                      sum += a[i][j];
                sum += a[j][i];
        }
    }
  return 0;
}
```

这里是二次元排列的内容全部吻合的代码，但因为使用a[j][i]访问，访问的顺序变成a、a+1000、a+2000…、a+1、a+1001…，所以比起被注释的a[j][i]也更容易发生缓存失效。

```
% opcontrol --reset
% repeat 100./a.out
% opcontrol --dump
% opreport -c
samples %       image name      app name        symbol name
-------------------------------------------------------------------
   231   100.000 a.out           a.out           main
   231   94.6721 a.out           a.out           cache_miss
   231   100.000 a.out           a.out           cache_miss [self]
```

尝试运行了100回左右，总共发生了1028次缓存失效。下面是交换注释部分的运行结果。

```
7       100.000 a.out          a.out              main
7       0.5255 a.out          a.out              cache_miss
7       100.000 a.out          a.out              cache_miss [self]
..
```

缓存失效变得非常少，仅仅出现了7次，可以说达到了预想的效果。

总结

本节的Hack介绍了获取系统profile的软件——oprofile。oprofile与很多CPU相适应，比sysprof获取的CPU信息更为详细。

——*Shinichiro Hamaji*

使用GDB操作运行进程

本节的Hack介绍了一些充分运用拥有GDB的ptrace(2)系统调用功能的操作运行进程的示例。

GDB可以作为ptrace(2)系统调用便捷的前端界面使用。特别是具有可以调用程序中的目标进程中的函数这一强大功能，如果与ptrace的attach功能结合使用，可以很简单地对运行中的进程进行干涉。

活用示例

介绍几个简单的活用示例。以下是x86的Debian GNU/Linux系统的试验。另外还有使用6.4之前的版本难以顺利运行的示例。

```
% ps x | grep firefox
 3616 ?        Rl     19:40 /usr/lib/firefox/firefox-bin -a firefox
% gdb -q -p 3616
(gdb) p chdir("/")
[Switching to Thread -1221168480 (LWP 3616)]
$1 = 0
(gdb) detach
Detaching from program: /usr/lib/firefox/firefox-bin, process 3616
(gdb)
```

这个例子读出使用gdb -p，attach后的目标进程中的chdir(2)系统调用。正因为在mound point下，故称作demon和X客户端的后台进程，在设备忙时，无法umount的情况下也可以使用。

文件的操作也比较简单，请试着做如下操作：首先启动cat，再启动其他的shell中的GDB，再操作cat进程。

```
% gdb -q -p $(pidof cat)
(gdb) p write(1, "hoge", 4)
$1 = 4
(gdb) p open("/etc/passwd", 0) .. 0 .. O_RDONLY ....
$2 = 3
(gdb) p dup2(3, 0)
$3 = 0
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

如果输出了hoge字符串和/etc/passwd文件的内容，cat进程就成功了。使用配套且耐用的套接口，开通新的TCP连接也是有可能的。

读出execlp(2)、可以在中途下达别的指令。

```
(gdb) p execlp("ls", "ls", "/", 0)

Program exited normally.
The program being debugged stopped while in a function called from GDB.
When the function (execlp) is done executing, GDB will silently
stop (instead of continuing to evaluate the expression containing
the function call).
(gdb)
```

"ls/"应该已经运行。最后的信息是对控制没有返回到GDB假定的场所进行警告。其他的诸如目标进程连接libdl的程序，可以调用dlopen(3)，连接共享目标，然后使用。但是libdl未被连接的情况下，无法使用dlopen(3)，由于对GDB没有连接功能，可以用**[Hack #86]**中介绍的具有连接功能的程序livepatch来代替使用。

应用示例：外部命令cd

在围绕Unix提出的问题中，有问题提出cd为什么不是外部命令而是gel的内置命令，其原因是无法改变其他进程的当前目录。下面通过本节Hack的应用示例，通过外部命令对cd进行实际安装。

实际安装过程中若组合使用GDB和脚本语言会更为简便。

```
#!/usr/bin/ruby
# Usage:
#   ext-cd directory

require 'tempfile'

dir = ARGV.shift
```

```
t = Tempfile.new("ext-cd.gdb")
t.puts <<"End"
attach #{Process.ppid}
call chdir(#{dir.dump})
End
t.close

# shut up gdb.
STDIN.reopen("/dev/null")
STDOUT.reopen("/dev/null")
STDERR.reopen("/dev/null")

system("gdb", "-batch", "-n", "-x", t.path)
```

运行结果如下例：

```
$ /bin/pwd
/tmp
$ ./ext-cd /
$ /bin/pwd
/
```

另外要补充的是，Solaris中有/bin/cd，但该命令在运行ext-cd等程序时不使用，单个cd进程中只需使用chdir。由于全部这些命令是否可以用exe(2)，则由POSIX决定，其理由是在决定POSIX时的争论中，作为怎样看起来都无效的外部命令，争论没有总结出哪一个含有有效的Case，结果变成全部命令都准备作为外部命令。

总结

本节的Hack对充分运用拥有GDB的ptrace(2)系统的功能，以及一些对运行中进程进行操作的示例做了介绍。如果使用GDB，诸如Ruby等的脚本语言可以简单地编程。

—— *Takeshi Yaegashi, Akira Tanaka*

使用硬件调试的功能

对拥有x86的硬件调试器的支持功能进行说明

在硬件中拥有处理器里的调试器支持功能，例如准备x86架构中8个的调试寄存器（DR0～DR7），运用本节的Hack中Linux进程来说明它的活用方法。

x86 的调试寄存器

对于 x86 的调试器功能在《IA-32 Intel; Architecture Software Developer's Manual,Volume 3B》的《CHAPTER 18 Debugging and performance Monitoring》中有完整的说明。

简单来说，如果有对用 DRO～DR3 的 4 个寄存器指定的 hear 地址所代表的存储空间，进行进程访问，INT1 的调试器发生了中断。剩余的 4 个寄存器需要预约，规定控制使用调试的支持功能。

Linux/x86 中的调试器可以设定每个进程的值，已经发生 INT 1 的进程中，可以发送 SIGTRAP。利用这些可以实现 GDB 的 hard ware watch point。

改写自身的调试寄存器

因为内核态不允许访问调试寄存器，程序为了改写调试寄存器必须使用 ptrace 系统调用。如果遇到了内核页面文件 asm-i386/user.h，则可以定义 struct user 中 int u_debugreg[8]的成员。

GDB 在读写目标进程时，虽然也可以调试寄存器，但在读写进程自身的调试寄存器时，进程无法对自身进行 ptrace 操作。所以，使用 fork，对子进程进行 ptrace。

以下是调试寄存器中调整值函数的实际安装示例：

```
#include <asm/user.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void
set_debugregs(unsigned long *v)
{
   if (!fork()) {
        int i, *p = ((struct user *)0)->u_debugreg;
        pid_t ppid = getppid();
        ptrace(PTRACE_ATTACH, ppid, NULL, NULL);
        waitpid(ppid, NULL, 0);
        for (i = 0; i < 8; i++, p++, v++) {
             if (i == 4 || i == 5)
                  continue;
             if (ptrace(PTRACE_POKEUSER, ppid, p, *v) < 0)
```

```
                fprintf(stderr,
                    "ptrace failed: dr%d = %08lx\n", i, *v);
            }
            ptrace(PTRACE_DETACH, ppid, NULL, NULL);
            exit(0);
        }
        wait(NULL);
    }
```

可以比较简单的书写像这样的调整值函数，但想逆向读出函数值时，为了得到已经运行的子进程的ptrace(PTRACE_PEEKUSER)的结果，一些进程间的通信也是必要的。

调试寄存器的活用示例

试着使用以下的set_debugger()函数，用简单的测试程序进行试验。

```
#include <signal.h>
#include <asm/ucontext.h>

int tmp, data0, data1;
void func(void) {}

static unsigned long regs[] = {
        (unsigned long)&data0,
        (unsigned long)&data1,
        (unsigned long)func,
        0, /* unused */
        0, /* reserved */
        0, /* reserved */
        0, /* cant read */
        0x00fd013f, /* Trap conditions
                    DR0: write, DR1: read/write, DR2: exec, DR3:
unused */
};

#define TRY(x) do { fputs("Trying " #x "\n", stderr); x; } while (0)

static void
trap_handler(int n, siginfo_t *si, struct ucontext *uc)
{
      fprintf(stderr, " Trapped at 0x%08lx\n", uc->uc_mcontext.eip);
      /* When we hit DR2, disable breakpoints to avoid infinite loop */
      if (uc->uc_mcontext.eip == regs[2]) {
              regs[7] = 0;
              set_debugregs(regs);
      }
}

int
main(void)
{
```

```
        struct sigaction sa = {
                .sa_sigaction = (void *)trap_handler,
                .sa_flags = SA_RESTART | SA_SIGINFO,
        };

        sigemptyset(&sa.sa_mask);
        sigaction(SIGTRAP, &sa, NULL);
        set_debugregs(regs);

        TRY(tmp = data0);
        TRY(tmp = data1);
        TRY(data0 = 1);
        TRY(data1 = 1);
        TRY(func());

        return 0;
}
```

编译运行后，结果如下：

```
% gcc -g -Wall -O2 debugregs.c
% ./a.out
Trying tmp = data0
Trying tmp = data1
  Trapped at 0x080487de
Trying data0 = 1
  Trapped at 0x08048818
Trying data1 = 1
  Trapped at 0x08048844
Trying func()
  Trapped at 0x080486b0
```

如果使用x86的调试寄存器，可以发现对于一般的以页为单位的内存保护机制来说，不可能被发现细微的内存访问也能被感知到。

假如在GDB中运行该程序，在SIGTRAP发生时，GDB的控制可以转移。与在“[Hack #92]C程序中断点的设定”一节中介绍的技术相同，它可以在设定程序的监视点时使用。

小结

本小节对拥有x86的硬件调试器的支持功能进行了说明。虽然在极少的记忆中可监视最大为4字，但熟练使用后对某些操作都很有效。

参考文献

- 《IA-32 Intel;Architecture Software Developer's Manual,Volume 3B》的《CHAPTER 18 Debugging and performance Monitoring》(*http://www.intel.com/design/Pentinm4/manuals/253669.htm*)

- 《32幅图解pit microcomputer8084的使用方法》的第十三章《debugger support》

——*Takeshi Yaegashi*

C程序中break point的设定可以用断点这个说法

介绍调试C程序时断点的设定。

调试C程序时GDB等的调试是有效的。通常，断点是从调试器中设定的，但本节的Hack介绍的是调试器对象的C程序中的断点的设定。

对Linux进行的#include操作，只要在任意的部分插入如下内容就可以了。

```
raise(SIGTRAP);
```

使用raise()函数发出SIGTRAP信号。或者若x86限定的话也可以插入下列内容。

```
__asm__ __volatile__("int3");
```

这里为了让SIGTRAP signal发生嵌入int3(Oxcc)命令。这种方法的使用与调用raise()不同，由于没有调用函数，调用栈不会散乱。GDB也是在break point的软件试设定时，在相关部分写入int3，所以做法相似（在使用GDB时，写入int3部分的源代码需要保存）。

若组成上述代码，在GDB上运行已经编译过的二进制文件，则应该中断相关部分中的处理，将控制权移交GDB。

在调试器上检索是否被运行

在调试器上运行程序时，若应用SIGTRAP被调试器处理这一性质，则可以在调试器上运行程序搜索。下列程序在调试器上运行是用being debugged表示，反之则用not being debugged表示。

```
#include <stdio.h>
#include <signal.h>
int being_debugged = 1;
void signal_handler(int signum) {
    being_debugged = 0;
}
```

```
int main() {
    signal(SIGTRAP, signal_handler);
    __asm__ __volatile__("int3");
    if (being_debugged) {
        printf("being debugged\n");
    } else {
        printf("not being debugged\n");
    }
    return(0);
}
```

运行结果如下

```
% gcc test.c
% ./a.out
not being debugged
% gdb ./a.out
... .. ...
(gdb) run
Starting program: /tmp/a.out
Program received signal SIGTRAP, Trace/breakpoint trap.
0x080483a9 in main ()
(gdb) c
Continuing.
being debugged
```

这是用于阻止有恶意的程序在调试器上运行的技巧，详细信息请参照《安全防范》。

总结

虽然想要进行上述操作的情况不是很多，但是在运行时特定的timing中中断处理然后用调试器发现该状态，以及在检测宏中很难用调试器设定断点的位置时，运用这个却是很方便的。

参考文献

- 《安全防范》（Cyrus Peikari、Anton Chuvakin 著，西原启辅校译，伊藤真浩、岸信之、进藤成纯翻译，O'Reilly Japan）。

—— *Satoru Takabayashi*

第7章

其他的Hack

Hack #93～100

本章将介绍没有纳入前面六章的各种各样的hack，最后，作为文献导读将介绍成为今后binary hack的指导书籍以及网址。

Boehm GC的结构

Boehm GC是用来把GC导入C/C++语言的内存管理库。

Boehm GC是能把GC导入C/C++语言的有力的内存管理库。适应于许多的UNIX,Windows，OS/2,也支持red操作用于W3M和gcj等，可谓是实用范围很广的库。

Boehm GC的使用方法

Boehm GC可以从开发者Boehm的网址（*www.hpl.hp.com/personal/Hans_Boehm/gc/*）得到源代码。写本书时的最新版本是6.6。

基本

使用Boehm GC，必须有gc.h，还要在主要进程中调用初始化函数GC_INIT。

```
#include"gc.h"

int main(intargc,char**argv){
  GC_INITC);
```

而后只需把C程序中的malloc/free换成Boehm GC提供的malloc即可。Boehm GC管理的可使用内存空间一满则会自动启动GC，未被使用的垃圾对象也将被回收。

之前的函数	置换的函数
malloc	GC_malloc
calloc	GC__calloc
realloc	GC_realloc
free	GC_free

GC_malloc_atomic作为GC_malloc的高速版使用，这是整数数组等内部不含指针的对象。GC_free是通常不做任何处理的函数，即使删除也无所谓。

Finalizer

被GC_malloc分配的空间，可注册被回收前的timing调用的回调函数。这类的函数被称为finalizer，可以以finalizer形式注册的是符合以下原型的函数。

```
typedef void (*GC_finalizetion_proc)(GC_PTR obj,GC_PTR client_data);
```

必须使用GC_register_finalizer注册，可以在object obj附上finalizer也可以接受登录时用client_data转移的资料。一个对象中可分配的只有一个，过去登录的东西由old_fn和old_client_data转移。

```
void GC_regiter_finalizer(GC_PTR obj,
                          GC_finalization_proc fn,GC_PTR client_data,
                          GC_finalization_proc*old fn,GC_PTR*old_client_data);
```

一旦用NULL调用GC_register finalizer和fn，即使正在运行中也能解除finalizer，另外使用finalize有几项注意事项。

- finalizer被调用后，对象会被强制回收，与Java的finalizer没有区别（对象的复活），在finalizer中，若保存对象的指针可以进行侵入，则有很大的危险性。
- 被finalizer登陆的对象也有亲子关系的情况，从父对象开始调用finalizer。因此，被finalizer注册的对象群一旦有循环引用。则会显示警告信息，finalizer也不能调用。要想进行拥有可以制作循环参数的对象的finalizer注册请使用GC_register_finalizer_ignore_self、GC_register_finalizer_no_order。
- 在多线程程序中，引起GC的线程可进行finalizer的运行，因此为了做到使用任何线程调用finalizer都不出现问题，必须安装。

在多线程上使用

在多线程环境下使用Boehm GC，必须遵守各个平台的限制。Linux中则有以下的限制。

- 所有的模块的最开头都要加上gc.h，且必须指定用-DGC_LINUX_THREADS与-D_REENTRANT进行编译。这是因为pthread库函数与Boehm GC的rapper函数互换。
- 在GC中平行使用dlopen系的函数，操作不保险。
- 使用线程本地存储后则有误收非垃圾对象的危险。

Boehm GC的构成

Boehm GC使用了Mark-Sweep算法。GC_malloc一旦在新对象的防护上失败，GC则会启动并进行以下处理。

1. 被GC root引用的Boehm GC的对象，被看作进行标记。GC root是指由全局变量、局部变量、malloc确定了的内存等，还有包含除Boehm管理外的所有内存。
2. 做上标记的对象中也包含指针，来回循环对这个对象进行标记，最后残留下的没被标记的对象，可以用于判断没被引用过的对象。
3. 占有的内存空间与释放列表相连，内存回收到此告一段落。

但是，实际进行以上处理需要多种Hack协助。

收集GC root

首先，GC root资料分散在寄存器、堆栈、data段、堆等。寄存器内的数据，是可以模拟setjmp运行的引导代码，因此可以将内容写入内存。

堆栈将从栈底（栈指针的初期位置）到现在的栈指针加到GCroot上。现在的栈指针可以从本地变量的地址推测出来。

进程下载共享库的情况中内存空间中的text段、data段则会混乱。为了只把data部分存入GCroot里，共享库的映射信息是必需的。在Bottom GC中**[Hack #65]**确认下载的共享库用一样的方法调查进程内的内存。例如在Linux中，给予dl_iterate_phdr(3)里以下这样的回调函数。

```
#include <stddef.h> /* for offsetof macro */
#include <link.h>
#include <elf.h>

int GC_register_dynlib_callback(struct dl_phdr_info* info,
size_t size, void* ptr) {
  int i;
  const ElfW(Phdr)* p = info->dlpi_phdr;

  for (i = 0; i < (int)(info->dlpi_phnum); ((i++),(p++)))
    switch (p->p_type) {
      case PT_LOAD: {
        char * start;
        if (!(p->p_flags & PF_W)) break;
        start = ((char *)(p->p_vaddr)) + info->dlpi_addr;
        /*将[start, start + p->p_memsz)内存区域附加到
          root segment上*/
      } break;
    }

  return 0;
}
```

malloc(3)所占的内存和mmap/munmap(2)确保的内存也将变成GCroot，后者所占的内存有存在未被分配的页的危险。这与“**[Hack #81]**使用SIGSEGV确认地址的有效性”一样，通过补充SIGSEGV来把握内存页的状态。

保守的GC

借助操作系统的帮助可以确定存放数据的地方，若是C/C++的程序的话，不能区分某些数据到底是小数型还是整数型。Boehm GC以字为单位来探索内存区域，类似指针的bit列也都被看成指针。这样，不能正确指定对象的指针的GC叫做保守GC。相比能正确区分哪是指针哪是数据的GC叫做“Precise GC”、“Type-accurateGC”、“Exact GC”等。

保守GC与Precise GC相比有两个缺点，一个是由于保守GC把bit列大约写成指针。Precise GC的一部分可能不被回收，内存效率会有稍许变差，另一个是用Precise GC的话可以移动任意空内存区。但是保守GC不可能实现。由此，程序一发展内存空区的分开化就会成为问题。

只是用C/C++语言来进行Precise GC，最低程度编译器、链接器的支持是必不可少的。现存库的链接也将不能实行，为了好好利用以前的资源，Boehm GC采用保守GC。

线程停止、用无重新启动机能的OS使线程停止

操作多线程时，调用GC_malloc，意识到堆内存枯竭的线程将运行GC。在GC中运行GC的线程以外一旦运行，能改换对象的内容的marking将不能正常运行。因此必须使其他的所有线程都停止。

然而，Pthread库的线程的停止、重新启动机能都无限定，也有把pthread_suspend_np等当成独立处理系的API来定义的OS。但在Linux等中线程停止、重新启动的功能本来就不存在。因此，Boehm GC实际配有使用信号来模拟线程停止、重新启动的功能。在Linux中，一般不使用SIGPWR、SIGXCPU信号而使用这个。

```
#define SIG_THR_SUSPEND SIGPWR
#define SIG_THR_RESTART SIGXCPU

__thread int count; /*suspend的多重记录*/
sigset_t suspend_handler_mask;

void init() {
  sigfillset(&suspend_handler_mask);
  /*等待中也可以接到信号*/
  sigdelset(&suspend_handler_mask, SIG_THR_SUSPEND);
  sigdelset(&suspend_handler_mask, SIG_THR_RESTART);
  sigdelset(&suspend_handler_mask, SIGINT);
  sigdelset(&suspend_handler_mask, SIGQUIT);
  sigdelset(&suspend_handler_mask, SIGABRT);
  sigdelset(&suspend_handler_mask, SIGTERM);
}
```

各个线程作为SIG_THR_SUSPEND、SIG_THR_RESTART信号的处理器先使thread_suspender登陆。信号处理期间用sigsuspend(2)待机。

```
void thread_suspender_handler(int sig, siginfo_t* sig_info, void*
sig_data) {

  if (sig == SIG_THR_SUSPEND) {
    count++;
    if (count == 0) {
      do {
        sigsuspend(&suspend_handler_mask);
      } while(count > 0);
    }
  } else if (sig == SIG_THR_RESTART) {
    count = 0;
  }
}
```

之后GC根据其他的线程用pthread_kill(3)传达要求。

```
pthread_kill(target_thread, SIG_THR_SUSPEND);
pthread_kill(target_thread, SIG_THR_RESTART);
```

总结

以上介绍了 Boehm GC 的使用方法和简单的原理，在 C/C++ 程序中，特别是多线程环境的 GC 很难处理。也正因此 Boehm GC 的内部安装是 Hack 的一道“坎”。深刻依存平台内存管理器的 Hack，有兴趣的各位请一定参照 Boehm GC 的源代码。

参考文献

- “Richard Jones's Garbage Collection Page”（*http://www.ukc.ac.uk/computer_science/ Html/Jones/gc.html*）可以说是 GC 教科书的决定版的一页。

—— *Minoru Nakamura*

请注意处理器的存储器顺序

在本 Hack 中，说明了内存访问的顺序的内存排序，内存排序的程序缺陷属于最难排查的一部分。

内存排序，或者说内存一致兼容性是处理器访问的规则。现代的处理器大部分的提高都为非性能，不同于指令的排列顺序，用多处理器系统编写并行、异步程序，进行驱动程序的开发时，请想到存储器顺序。

以下这段代码是两个线程间保护临界区的 lock 程序，线程各自有 0 和 1 的编号，是想进入临界区的。

首先把 `lock_array[me]` 换成 1（第 8 行），然后确认另外一个线程的 `lock_array[other]`（第 9 行），其他的线程尝试同时进入。因为出现了 1 将产生竞争。因为 lock 失败，自己的 `lock_array[me]` 清除为 0 得重新再做。

这段程序看似顺利在运行，但根据处理器的种类有两个线程同时进入临界区的危险。

```
1 volatile int lock_array[2];
2
3 void routine(int me /* 0 or 1 */) {
4   int other = 1 - me;
5
```

```
 6 start_point:
 7   /* lock */
 8   lock_array[me] = 1;
 9   if (lock_array[other] != 0) {
10      lock_array[me] = 0;
11      /* wait a moment */
12      goto start_point;
13   }
14
15   /* critical section */
16
17   /* unlock */
18   lock_array[me] = 0;
19 }
```

原因在于第8行与第9行的运行顺序，根据volatile修饰符、lock_array[me]的储存命令→lock_array[other]的读取命令的顺序保证其机器语言化。

处理器的内部构造有时会破坏程序的顺序转换存储器访问。存储器访问将变成知道了对方的状态后改变自己的状态的程序而无法保证运行，如此会引起意想不到的情况，这就是顺序问题的难处。

处理器的存储器访问顺序

内存顺序有同一地址的访问顺序和不同地址的访问顺序。

对于同一地址的存储器访问，大部分的处理器可以保证照程序运行，但是若是IA-64不保护顺序，并运行下面的代码，即便出现a = 2、b = 1的结果也被认同。

```
Thread1       Thread2
==========    ==========
1: *p = 1;
2:            *p = 2;
3: a = *p;
4: b = *p;
```

另一方面不同的存储器地址的存储器访问的顺序对现在用的大多处理器都成问题。

存储器访问，分为Write After Read（WAR）、Read After Write（RAW）、Read After Read（RAR）、Write After Write（WAW）4种类型。例如：WAR是存储命令发出后再发读取命令这一类型，所有的类型中保证顺序的内存访问顺序称为“Strong Ordering”（也叫Sequential Ordering）或是

Program Ordering。相反，所有类型中认可顺序变更的内存访问顺序叫做“Weak Ordering”，串有各种各样的内存 Relaxed Memory Ordering 。如表 7-1，主要的内存访问顺序以及采用其他处理器进行综合（保证顺序的为○，有逆转可能性的记为×，只是平常不使用，特殊模式和部分命令有效的内存访问顺序除外。）

表 7-1：内存访问顺序的种类

种类	RAR	WAR	WAW	RAW	使用的处理器
Strong Ordering	○	○	○	○	i386、PA-RISC
Total Store Ordering	○	○	○	×	IBM 370、SPARC (normal)、i486、Pentium
Partial Store Ordering	○	○	×	×	SPARC（PSOmode）
Speculative Processor Ordering	×	×	○	×	Pentium Pro、Pentium 4
Weak Ordering	×	×	×	×	IA-64、Alpha、POWER、SPARC（RMOmode）

memory barrier 命令

引起内存储器访问的○逆转是由于进程与缓存之间的叫做存储缓冲的机构。存储缓冲是处理器中的一种缓存。可让占几个命令的分量的 store 命令在处理器中待机。其结果得出以下效果。

- 在相同地址运行两次存储命令，最开始的存储命令将被取消，写入缓存。
- 在相同的地址上，存储命令→载入命令相连接，后发的载入命令跳过从存储缓冲区得到数据的缓存的读取。

因为有存储缓冲区减少了对缓存的访问，高速化虽有可能，但对不同地址的存储器访问速度会变慢，因此进程预备好了被叫做内存屏障命令或是 fence 命令的存储器访问顺序化的专用命令。

内存屏障命令的形式及其效果根据架构而各异，但 x86 中，有表 7-2 这样的命令。

表7-2：x86的memory barrier命令

命令	使用的处理器	概要	顺序化的依赖
sfence	Pentium III	store命令的连续化	WAW
lfence	Pentium 4	下载命令的连续化	RAR
mfence	Pentium 4	下载命令与store命令的连续化	4种形式全部

在GCC中用以下的inline汇编程序的宏可以在编码中加入memory barrier命令。

```
#define membar() asm volatile("mfence":::"memory")
```

前面的样本程序由于第8行与第9行之间夹有这个membar错误，将不出现表7-3中总结主要的处理器架构的memory barrier命令。

表7-3：各种各样架构的memory barrier命令

架构	命令	备注
x86	sfence、lfence、mfence	
IA-64	Mf	a
PowerPC & POWER	sync、lwsync、eieio、isync	
SPARC	stbar、membar	
PA-RISC	Sync	b
Alpha	mb、wmb	
MIPS	Sync	c

a. IA-64是存在于其他地方，立足于语义存储器基础上独特的memory barrier被广泛使用。

b. PA-RISC是基本的存储器访问，属strong ordering，但cache限制命令等的顺序性将变弱。

c. 就作者调查的范围，好像MIPS用架构水平并未确定内存排序是否使用内存边界，根据处理器芯片和系统构成的不同而不同。

原子命令产生的atomic命令效果

大多的处理器在发出Compare-And-Swap等atomic命令时，存储器访问有上升的副作用。若是x86附有lock扩展的原子操作，定义为等待直到先行命令完成存储缓冲清空。所以比较memory barrier更要强有力的进行atomic命令效果。

只是，原子命令产生的效果并非所有的架构都能实现。在 IA-64、Alpha、POWER 中，除原子操作对象的地址以外的存储器访问，允许越过原子命令来改变顺序。

另外，就给予系统的影响也有必要注意，内存屏障命令虽是处理器中能与原子命令具有运行顺序的命令，但原子命令是为锁定系统总线、存储器总线，并影响其他处理器的重要命令。因此，只需内存屏障命令就能完成的情况，使用内存命令的话效率更高。

程序语言和 API 支持

若用 C/C++ 语言就能使用内存屏障和原子操作固然是好，但 2006 年到现在除了紧紧依靠系统外还是别无他法。想确定内存屏障标准的尝试比如Java用语言方式来定义内存的尝试都在向前发展。让我们期待今后工作的展开。

另外，在操作系统和系统库提供的 API 中，有保证存储器访问的 atomic 命令效果的方法。例如：`pthread_mutex_lock` 同 barrier 命令一样，可以原子命令在 API 调用前的存储器访问，但由于成本高，不能代替内存边界使用，但可以考虑在调用 API 前进行内存排序。

总结

以上就决定存储器访问的顺序的内存排序进行了说明。内存排序的问题，即使编写多线程程序的人士也一直没有觉察到的。但是就著作者的经验，内存排序的程序缺陷属于最难调查的一部分。让我们小心不要掉入意想不到的陷阱。

参考文献

1. “Linux Journal, Memory Ordering in Modern Microprocessors”（Part I：*http://www. linuxjournal.com/article/8211*、Part II：*http://www.linuxjournal.com/article/8212*）
2. “The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, 4.10 Memory Synchronization”（*http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap04.html#tag_04_10*）
3. “Threads and memory model for C++..Hans Boehm”（*http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/*）

4. "Java Community Process, JSR-000133 Java Memory Model and Thread Specification Revision" (*http://jcp.org/aboutJava/communityprocess/final/jsr133/index.html*)

—— *Minoru Nakamura*

对 Portable Coroutine Library（PCL）进行轻量的并行处理

本章 Hack 将介绍用 PCL，基于 C 语言的 co-routine 的使用方法和 PCL 的原理进行说明。

本章Hack将介绍用PCL、C 语言使用~的方法以及本章Hack中将进行简单的 Coroutine 的 ~~ 的解说，以及用 C 语言等语言实现 Portable Coroutine Library 的解说。

co-routine 是什么

co-routine 也叫做 micro-thread、fiber。由于 fiber 和 thread 是相对应的名字，作者认为这是一个很新颖的命名。我认为用与 thread 对比的、"非时间段非协调型的 thread"、"本身可认为或说必须进行替换 thread" 等说明会比较易懂。从与 sug routine 的对比来考虑的话，sug routine 正如词缀 sug 所表示的，调用方为主并在这里运行。而 co-routine 正如词缀 co 所表示，调用方被调用后互相协调的运行。

```
coroutine() {
  print("1\n")
  yield
  print("3\n")
}

main() {
  coroutine()
  print("2\n")
  yield
  print("4\n")
}
```

这种情况下，1、2、3、4 可以按顺序输出。上面所记的编码，跟由 yield 把处理转移到其他的 co-routine 是很具特点的，与线程不同，co-routine 不

同时运行。因此无需进行同期处理等，非常简单。比线程还方便的地方很多。作者经常把它用于游戏的运行处理部分等。

Portable Coroutine Library (PCL)

要实现 co-routine、stack（推理以及当作 context 使用的寄存器必须退避），只要是语言系统自身不支持就不能使用。但是，若使用吸收结构差异的 PCL，不能使用 co-routine 的语言也可使用 co-routine。

PCL 可以从 *http://xmailserver.org/libpcl.html* 得到。写作本书时的最新版是 1.6，作为同种类型的库，PCL 相对来说很大，但 GNU Pth 也很有名。另外，C++ 中，也有 Boost Coroutine（*https://boost_consulting.com:8443/trac/soc/wiki/coroutine*）。

把刚才的例子用 PCL 重新写成样本如下所示。

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

#define CO_STACK_SIZE (32 * 1024)

void spawn(void *arg) {
    printf("%d\n", 1);
    co_resume();
    printf("%d\n", 3);
}

int main() {
    coroutine_t co;

    co = co_create(&spawn, NULL, NULL, CO_STACK_SIZE);
    co_call(co);
    printf("%d\n", 2);
    co_call(co);
    printf("%d\n", 4);

    return 0;
}
```

上面所记的运行也是 1、2、3、4 的顺序，如您所见用 `co_call` 来运行 `co_resume`。用 `co_resume` 可从 coroutine 回归。`co_create` 的第 2 个参数，在调用第一个参数的函数时可作为参数使用，但在此没被使用。

PCL 的安装

大部分的PCL 的编码非常短，一个pcl.c大致有500行，可以完成基本的操作。只有把堆栈保存在动态确定空间里和调整寄存器 setjmp/longjmp 中。它的大部分安装在*http://www.gnu.org/software/pth/rse-pmt.ps*的论文里有说明。

首先，只需储存最近的叫做 ucontext(3)的Unix 环境，就可以利用它。这恰恰就是用来进行 coroutine 的机构，用 makecontext 生成 context，使用 getcontext 和 swapcontext 可进行 context 的退出和变更。

在没有它的环境中，context 的移动由 setjmp/longjmp 进行，用有些方法再造一个 stack 来保存 context。

第2个方法是使用sigstack(2)/sigaltstack(2)这种系统调用。这种方法在上述论文中有详细的叙述，但是用 sigstack/sigaltstack 把 sigaltstack 向系统通知后，一旦访问信号句柄就会 setjmp，制作一个 context，返回的话 stack 又将回到原位。这就是它的大概过程，这以后利用 setjmp/longjmp 可实现循环。

最后，虽然过时但实用的方法是动态保存 stack，并复制、保存每个架构及OS的必要空间，进入jmp_buf构造体的内部安装。自己重新输入指定stack的寄存器和保持指令、程序运行位置的寄存器的空间。

这种方法实用但有环境依存，被 ifdef 强制分开。增加 PCL 的对应环境，最后的方法是增加 ifdef，例如笔者在 mingw32 环境中想使用 PCL，从支持Io这个coroutine的脚本语言的Scheduler.c拷贝粘贴适宜场所来使用。加入笔者做成的 PCL-1.6 的补丁（patch），在 *http://shinh.skr.jp/binary/pcl_ioarch_1_6.patch* 中。

总结

在本 Hack 中，介绍了用PCL.C 语言来使用coroutine的方法和PCL 的原理。

—— *Shinichiro Hamaji*

计算 CPU 的 clock 数

读出 CPU clock，可测定用于细微处操作的时间。

最近的大部分处理器中都有 CPU 的 clock 以及类似于它的 clock 的计算。它的值可通过软件获得。在 lock 等非常小的处理速度测定上有用。另外，也可用于了解随机数的种类以及输入后所在的时间。并且，通过调查一秒钟内有多少 clock 就可以测定频率。

各种各样的处理器

以下的处理器中有读出 clock 的功能。

进程	命令	字节数	参考
Pentium	`rdtsc`	64	EDX 里容纳上位 32 位、EAX 里容纳下位 32 位
Itanium	`mov R = ar44`	64	R 里容纳 64 位
32bit PowerPC	`mftbu H、mftb L`	64	H 里容纳上位 32 位、L 里容下位 32 位 [a] [b]
64bit PowerPC	`mftb R`	64	R 里容纳 64 位
UltraSPARC	`rd %tick, R`	64	R 里容纳 64 位 [c]
HP PA-RISC	`2.0 mfctl %cr16, R`	64	R 里容纳 64 位
HP PA-RISC 1.0	`mfctl %cr16, R`	32	R 里容纳 32 位
Alpha	`rpcc R`	32	R 的下位 32 位里容纳 32 位 [d]
S/390	`stck ADDRESS`	64	在指定的地址里容纳 8 位
MIPS	`mfc0 R, $9`	32	[e]
SH64	`getcon cr62, R`		[e]

a. 在 32 位 PowerPC 中，2 个命令是必不可少的。由于不能一次性读出 64 位的数值。因此按 `mftbu`、`mftb`、`mftbu` 的顺序读出。用 2 个 `mftbu` 来确定读出的高级 32 位并未变化。

b. 用 PowerPC 得到的 clock 并非 CPU 的 clock，是总线 clock 的 1/4。

c. 用 UltraSPARC 使用 64 位寄存器必须要有 OS 依存的值。

d. Alpha 的寄存器有 64 位，高级 32 位中 Linux 则是高级 32 位加上低级 64 位的话，也看成为 process 单位的 clock 数值这类的修正值。

e. 未经实验确认。

安装

在此，作为例子，在 Pentium AMD64，32 位的 Power PC 进行安装。

Pentium

Pentium 以及 Pentium 互换的处理器可如下安装。

```
unsigned long long clockcount_pentium(void)
{
  unsigned long long ret;
  __asm__ volatile ("rdtsc" : "=A" (ret));
  return ret;
}
```

在这里，rdtsc 是 EDX:EAX/EDX 里有上位 32 位，EAX 里有下位 32 位。64 位值的表示方法里容纳 64 位。但是 EDX:EAX 是表现 Pentium 中 64 位值的标准方法。因此，无论**[Hack #23]**用 GCC 使用 inline assember，还是所述的 constraint，表示 EDX:EAX 可指定 "A"，在这使用它可以从变量中取出数值。

AMD64

AMD64的命令set由于Pentium的上位互换，可使用rdtsc进行如下安装。

```
unsigned long long clockcount_amd64(void)
{
  unsigned int eax, edx;
  __asm__ volatile ("rdtsc" : "=a" (eax), "=d" (edx));
  return eax | (unsigned long long)edx << 32;
}
```

在这里，用 rdtsc 从 EDX:EAX 取出的 clock，先使用 EAX、EDX 指定 "a" 和 "d"，然后取出每个变量，并用 C 重新构成 64 位值。

另外，这种安装不仅是 AMD64，Pentium 也可使用。相反，Pentium 项显示的安装 AMD64 却不能用。这是由于 "A" 这个指定在 AMD64 中显示 RAX 不能取出 EDX:EAX。

32 bit PowerPC

32 bit PowerPC 可如下所示。

```
unsigned long long clockcount_powerpc32(void)
{
  unsigned int h1, h2, l;
```

```
  do {
    __asm__ volatile ("mftbu %0" : "=r" (h1));
    __asm__ volatile ("mftb %0" : "=r" (l));
    __asm__ volatile ("mftbu %0" : "=r" (h2));
  } while (__builtin_expect(h1 != h2, 0));
  return (unsigned long long)h1 << 32 | l;
}
```

在这里，用mftbu可得出上位32位，用mftb则可得出下位。但是，在最初的mftbu和mftb中间，clock前进可能出现下位32位溢出，上位32位增大。因此，须再运行一次mftbu，若高级32位变化则再从头开始做。另外，由于要求重做的可能性非常小，用__builtin_expect，若知gcc h1 != h2可高效率变成0并帮助进行分歧预测。

测定通过频率和boot的通过时间

使用上面所说的clockcount_pentium()，用以下的程序就可实现通过clock的频率和boot的通过时间。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>

int main()
{
  int ret;
  struct timeval tv1, tv2;
  unsigned long long c1, c2;
  double t, speed, uptime;

  c1 = clockcount_pentium();
  ret = gettimeofday(&tv1, NULL);
  if (ret == -1) { perror("gettimeofday"); exit(1); }
  for (;;) {
    sleep(1);
    c2 = clockcount_pentium();
    ret = gettimeofday(&tv2, NULL);
    if (ret == -1) { perror("gettimeofday"); exit(1); }
    t = tv2.tv_sec - tv1.tv_sec + (tv2.tv_usec - tv1.tv_usec) * 1e-6;
    speed = (c2 - c1) / t;
    uptime = c2 / speed;
    printf("0x%llx %lf[Hz] %lf[sec]\n", c2, speed, uptime);
  }
}
```

假设这个程序的运行结果会如下。

```
0xa0ed80a7f9f7 1261795267.448678[Hz] 140230.297854[sec]
0xa0edcdb8978f 1274756551.617955[Hz] 138805.498897[sec]
```

```
0xa0ee1bb2474b 1284146489.383961[Hz] 137791.543871[sec]
0xa0ee66e312cf 1277204231.704936[Hz] 138541.499022[sec]
...
```

从这个结果中，可以知道频率为1.2～1.3GHz，从boot开始该过程大约需要38小时的时间。

注意

周期

计数器为32位的时候，假设频率为1GHz，计数器将会发生大约4s溢出。因此，虽然可以测定微处理时间，但是却不能够知道从boot开始的相对较长的时间。

SMP

多核处理器的每个处理器都有计算器。这种情况下各计算器并非完全同步，即使在起始状态设定的同时，也有慢慢变得偏离的可能。

clock的变化

最近的处理器为了节约电力，常降低操作频率。例如：Intel的SpeedStep、AMD的PowerNow!等。这种情况，在本Hack中读出的clock并不一定与实际的速度能够相比。

总结

通过读出处理器的clock，可测定用于细小操作上的时间。

——*Akira Tanaka*

HACK #97 浮点数的bit列表现

在计算机中，根据IEEE754的规格形式，以bit列形式来表现实际的近似值。

在本Hack中，说明了计算机中的实数的近似值是如何表现的。

浮点数

用计算机处理实数时，通常来用浮点数近似表现的方法。关于浮点数，有

IEEE754这一标准，近年几乎所有的处理器都以这一标准为立。IEEE754标准有如何用bit列来表示数值，还规定大致的方向控制，例外、溢出时的操作等规定。

图7-1：x86 中浮点数的表现形式

在IEEE754中，作为数值的表现形式规定用32位表现的单精度数、64位的双精度数以及以两者为基础的增加bit计数的扩展精度、扩展双精度。在C语言中，float型是单精度，double型是双精度，long double型是扩展精度。

大多数的处理器都支持单精度和双精度，部分处理器也支持比64位还长（例如128位）的扩张双精度。例如x86处理器的扩张双精度是80位。相反，像大多数的GPU一样，以单精度就已足够，而不支持双精度以上的处理器也不为少。

浮点小数由符号、exponent、significand、mantissa三部分组成。综合此三部分，单精度则为32位，双精度则为64位，符号、exponent以及significand、mantissa各个部分的bit各用s、e、f指代，其bit列基本上表示为以下的值。

```
单精度：(-1)^s 2^e-127 1.f
倍精度：(-1)^s 2^e-1023 1.f
```

在这里"."即小数点，例如：

```
1 01111111 11000000000000000000000 （二进制）
```

这个单精度数如下表示为−1.75。

```
(-1)^1 2^127-127 1.11 = -1.75
```

请注意小数1.11是二进制。指数部的bit全部为0，另外，所有的地方都有特殊的规定。无限大、NaN以及非正规数等等。另外列出别的种类的数值。

缩写浮点数

确认上面所记的bit列是-1.75。下面，假设int型和float型的大小都为32位。这个假设，在大部分32位/64位的Unix OS及Windows中也成立。

```
unsigned int i;
float f;

i = 0xbfe00000u; /* -1.75 */
printf("%x\n", i);

f = *((float *)&i);
printf("%g\n", f);
```

运行这个编码要如下所示，能确定前面所示的bit列（0xbfe00000）表示-1.75。

```
bfe00000
-1.75
```

在Java中，像这样用多个不同的类型来处理同一的内存空间是不可行的。但是，却有用来进行从bit列到浮点数的变换的Float#intBitsToFloat、Double#longBitsToDouble方法。

总结

在计算机中，根据IEEE754标准，以bit列形式来表达实数的近似值。

—— *Kazuyuki Shudo*

x86的浮点数运算命令的特殊性

在Hack中，将说明导出与其他处理器不同的运算结果的x86FPU的特殊性，也介绍了它的回避方法。

在x86处理器的浮点数运算器中，尽管也是IEEE754标准，有若干和其他处理器不同的地方。也因此会出现四则运算的结果与其他处理器不同的现象。

特殊点

x86 处理器的浮点数（以下 FP）运算命令有几大特殊点。近来，FP 寄存器也和整数寄存器一样。一般都是广泛使用的寄存器并排架构式。但自从 x86 的 FP 寄存器换成 stack，stack top 起着特殊的作用。

x86 的特殊性并不仅如此。尽管遵守 IEEE754 标准 FP 寄存器上的数值表现与其他的大多数处理器不同。因此会出现运算结果与其他的处理器不同的现象。

例如：下列程序的运行结果，与其他的 IEEE754 标准处理器就不同。

```
#include <stdio.h>

double dmul(double a, double b) {
  return a * b;
}

int main(int argc, char **argv) {
  unsigned long long int i, j;
  double f, g;

#ifdef __i386__
  {
    unsigned short cw;
    asm("fnstcw %0" : "=m" (cw));
    cw &= ~0x0300u;
    cw |= 0x0200u; /* double precision (64 bit) */
    asm("fldcw %0" : : "m" (cw));
  }
#endif

  i = 0x0008008000000000ull; /* 1.11281e-308 */
  j = 0x3ff0000000000001ull; /* 1.0 + alpha */

  printf("0x%016llx * 0x%016llx\n", i, j); /* uses i & j */

  f = *((double *)&i);
  g = *((double *)&j);

  printf("%g (0x%016llx) * %g (0x%016llx) =\n", f, f, g, g);

  f = dmul(f, g);

  printf("%g (0x%016llx)\n", f, f);
}
```

在 SPARC 等其他处理器中，运行结果如下：

```
0x0008008000000000 * 0x3ff0000000000001
1.11281e-308 (0x0008008000000000) * 1 (0x3ff0000000000001) =
1.11281e-308 (0x0008008000000001)
```

但在x86中则如下：

```
0x0008008000000000 * 0x3ff0000000000001
1.11281e-308 (0x0008008000000000) * 1 (0x3ff0000000000001) =
1.11281e-308 (0x0008008000000000)
```

运算结果末尾的1位，在其他的处理器中是1，而在x86中会成0，请注意这一点。

x86的浮点数寄存器

x86支持由IEEE754规定的单精度、双精度，扩展双精度的表现形式。各个的长度为32位、64位、80位（参照[Hack #97]浮点数的bit列表现）。由浮点数运算器控制寄存器的值，可设定小数部分的精确度。前面记载的程序的inline assembler编码。通过运行fldcw命令来设定其精确度。

从这开始就是x86独特的地方，精度的设定的影响只在于小数部分。指数部在寄存器上通常是15位。即使设定为单精度或双精度也不会变为8位或11位，而是保持常态F的15位（图7-2）。

在运算结果的绝对值变得大到无法用8、11位的指数部分表现时，在其他的处理器上的结果将变得无限大。但是，在x86的15位指数部分不会出现这样膨胀。

一旦运行以下的程序，在SPARC等其他的处理器中会出现膨胀，运算结果会变得无限大，但在x86中将不会发生。

```
#include <stdio.h>

int main(int argc, char **argv) {
```

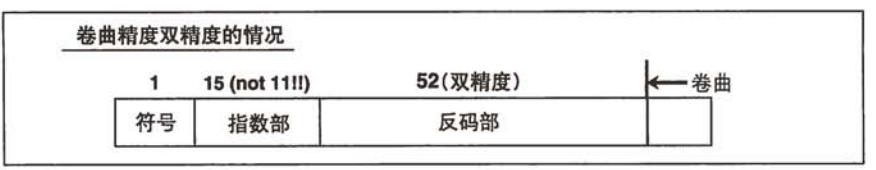


图7-2：x86操作的特殊性

```
  unsigned long long int i, j, k;
  double f, g, h;

#ifdef __i386__
{
  unsigned short cw;
```

```
    asm("fnstcw %0" : "=m" (cw));
    cw &= ~0x0300u;
    cw |= 0x0200u; /* double precision (64 bit) */
    asm("fldcw %0" : : "m" (cw));
  }
#endif

  i = 0x7fe0000000000000ull; /* 1.0 x 2^1023 */
  k = j = i;
  printf("%016llx, %016llx, %016llx\n", i, j, k); /* uses i, j and k */

  f = *((double *)&i);
  g = *((double *)&j);
  h = *((double *)&k);

  printf("%g + %g - %g = ", f, g, h);

  f += g;
  f -= h;

  printf("%g\n", f);
}
```

在SPARC中的运行结果如下：结果"Inf"也就是说变得无限大。在x86中，这种膨胀不会发生，不是"Inf"而是"8.98847e+307"。

```
7fe0000000000000, 7fe0000000000000, 7fe0000000000000
8.98847e+307 + 8.98847e+307 - 8.98847e+307 = Inf
```

对策

在x86得出和其他处理器完全一样的运算结果可能吗？可能。但用一般的方法是行不通的。首先，可考虑把每次运算的结果存入存储器的方法。必要的话可再次下载到寄存器。指数部分通常是15位。存入存储器的话对应精度的位数若是单精度则为8位，是双精度则为11位。

但是，这种方法也并不完美。运算时存入存储器时的两次引起小数部分的异常(用其他的处理器进行)，有时会出现与一次估算得到的结果不同的值。也就是说，发生溢出作为双精度虽可作为非正规的表达值，但指数部存在于15位和余数这两部分。

所以在寄存器中，可以标准数形式表达。这种情况下，存入存储器时会转换为非标准数，在这将发生四舍五入。最近列举的乘法程序例，恰恰如此是四舍五入发生两次的例子。为了防止两次四舍五入的发生，不仅在存储器存储时，运算时也必须适当控制溢出。这通过在运算前，先乘上操作数相应的常量，运算后乘以常量的相反数就能实现。

只有这样做，才可以在X86中也能得出与其他的处理器得出完全一样的结果。

SSE2命令

在pentium 4以后的X86的处理器中，有Streaming SIMD Extensions 2（SSE2）这样的SIMD运算命令集。SSE2虽是完成SIMD运算的命令集，但也是包括占有两个操作数的一般的浮点数运算。数值的表现形式是IEEE754的单精度和双精度。则运算、平方根、四舍五入方向的控制等，除余数外支持大部分的IEEE754的要求。

实际上，使用SSE2命令来代替SSE2之前的传统的FP运算命令，可以得到和其他处理器一样的运算结果。在SSE2中，指数部分并不一定为15位。

Java语言的strictfp

在Java语言中有`strictfp`这样的修饰。它可以安装在类方法界面里。在其中可以得到一样的运算结果。x86以外的IEEE754标准处理器情况类似，也就是说，用x86上的Java虚拟机SSE2命令可否进行FP运算。或者可用前面所述的复杂的方法。

总结

说明了导出与其他的处理器不同的运算结果的x86FPU的特殊性及其回避方法。

—— *Kazuyuki Shudo*

HACK #99 用结果无限大和NaN化运算来生成信号

作为在运算结果无限大和非数值情况下生成信号的方法。将介绍使用C99规格的header fenv.h的方法和x86处理器本身拥有的方法。

本章Hack，将介绍在浮点数运算结果无限大和非数值时生成信号的方法。

IEEE754的规定

浮点数运算规格IEEE754（参照**[Hack #97]**浮点数的bit列表现），只要没有特别规定，即使运算结果变得无限大和非数值化，都不要中断程序的运行，

这是规定。因此，即使是非自主地变成无限大和NaN，程序还会使用其结果继续处理工作。

在这里，将叙述运算结果变得无限大或NaN时的检测方法，介绍在glibc中的方法和X86使用的方法。

IEEE754规定的例外和SIGFPE

运算结果变得无限大或NaN，会出现相对应的例外，浮点数运算器中的例外flag会被设定。

这些例外，是否会引起处理器异常，还要看FPU的设定。处理器的例外在Unix OS中，会生成SIGFPE这样的信号，只要没特别设定处理器的例外，也就不生成信号。这由IEEE754本身规定。

例如，一旦运行以下的0除法计算，运算结果是正数的无限大。用print(3)来表示结果则表示为Inf。

```
double a = 1.0 / 0.0 ;
```

若任其如此，自然地，运算结果变得无限大或NaN，即会变成非常棘手的事情。就这样任处理器进展下去，要确定哪儿会出现问题，哪个编码是问题产生的原因等就会变得很难。

因此，以下介绍运算时生成信号的方法。若信号生成，用debug运行程序，可确定无限大和NaN的原因。

用glibc的方法

C99标准中，有定义浮点数演算器（FPU）的四舍五入方式和处理例外的fenv.h这样的头文件。fenv.h宣告了几个处理FPU中的例外flag。处理器的例外，也就是控制信号生成的方法。

glibc作为C99标准的独自的扩展，提供控制生成信号的函数。以下的编码针对某些种类的异常，设定FPU使其生成信号。如第一行通过定义_GNU_SOURCE，而使得glibc的独自扩展变得有效。

```
#define _GNU_SOURCE
#include <fenv.h>

int excepts = fegetexcept() ;
```

```
excepts | = FE_INVALID | FE_DIVBZERO | FE_OVERFLOW
feenableexcept(excepts);
```

由此，前面提到的0除数除法运算中将生成SIGFPE。只要不捕捉这个信号，程序就会以以下的异常状态结束。

```
(运行)
Floating exception
```

可指定例外的种类如下：

宏	条件
FE_INVALID	非法操作
FE_DIVBYZERO	0式除法运算
FE_OVERFLOW	溢出
FE_UNDERFLOW	under flow
FE_INEXACT	精度错误
FE_ALL_EXCEPT	被支持的所有例外的理论和

x86依赖的方法

在x86中，通过操作FPU中的控制字寄存器，就浮点数运算的各种例外，可使处理器的异常发生。只要没有特别设定，所有的异常都可抑制。以下的inline汇编编码去除一部分的掩码。

```
unsigned short cw;
asm("fnstcw %0" : "=m" (cw));
cw &= ~0xd;
asm("fldcw %0" : : "m" (cw));
```

在这第二行，在变量cw中下载控制字寄存器，第三行除去mask，第四行在寄存器里返回值。若是Linux，fpu_control.h header.#include可如下书写：

```
fpu_control_t cw;
_FPU_GETCW(cw);
cw &= ~(_FPU_MASK_IM | _FPU_MASK_ZM | _FPU_MASK_OM);
_FPU_SETCW(cw);
```

由于这个设定，通过0除法运算可使SIGFPE发生，为了这种类别的设定变更，各个OS中准备了以下的头文件。

Linux fpu_control.h

FreeBSD	floatingpoint.h
Solaris	ieeefp.h

在x86的FPU控制字寄存器里，作为mask有以下6位，这些对应各个例外的发生条件。在这中间，只有非正规数例外，在IEEE754中没被规定。

宏	值	条件
IM	0x1	非法操作
DM	0x2	非正规数
ZM	0x4	0式除法运算
OM	0x8	溢出
UM	0x10	under flow
PM	0x20	精度错误

前面记载的程序中，除去了0xd相对应的mask，也就是说，清除了IM、ZM、OM位。

总结

作为在运算结果变得无限大和非数值情况下生成信号的方法。介绍了C99标准头文件fenv.h的使用方法和x86处理器本身固有的方法。

—— *Kazuyuki Shudo*

文献介绍

本章将介绍供Binary Hack参考的文献。

书籍

Write Great Code

要编写Great Code正如认识硬件，编写软件这个副标题所示。在《Write Great Code》中，CPU、Cache内存器、存储器、周边设备等硬件的话题中分为许多的版面。里面有许多要掌握本书必需的基础知识。（Randall Hyde著、トップスタジオ译，鸠鹞文敏、后藤正德，松本雪宏审译，每日交流）

详解Unix编程

关于Unix的系统调用、库函数等一个个都有详细解说的重要的书籍。平常

无意识地使用的系统调用，若仔细阅读本书的说明，应该能发现各种各样的注意事项和意想不到的使用方法，关于历史背景的记述也很丰富，标准规格的解释非常严密也是一大特色，是在Unix中进行编程必备的书。（W. Richard Stevens著，大木敦雄译，Karl Pearson Etude）。

计算机的构成和设计

从作者的名字（パターソンとヘネシー）中可以知道《パタヘネ》这个名著。从计算机的基本概念命令集、算术运算性能评价、处理器和计算机系统的架构，都做了解说。通过它也可了解硬件和软件之间的相互关系。不仅能从软件后，从硬件方看也能得到理解如何运行的必须的知识。（上/下卷、David. A、John. L. 著，成田光彰译，日经BP社）

Linkers&Loaders

关于链接器和加载较为稀有的专业书籍。不仅是链接器和加载的结构，还可得到与系统编程有关的知识。书的前半部分就链接器相关的OS的结构，例如假想内存和文件映射的关系，以及映射的输入时复制的结构等内容和历史性原因一样做了说明，后半部分详细介绍了链接器的结构、位置独立编码的实现方法和动态链接的结构。这本书推荐给想详细了解本书多次出现的链接器，下载工具的使用方法的各位。（John R.Levine著，神原一矢审译，ポジティブエッジ译，オーム社）

Debug的理论和安装

正如其名，是就debug的理论知识和安装的说明书，不仅限于特定的debug内容。各种OS的debug 、Java的debug等囊括各种各样与debug相关的知识。debug的安装与运行环境的OS之间有很密切的关系，但break point、stepping等基本的概念通行于任何debug。在本书中，就实现这些debug功能而准备的API和处理器的功能做了详细的说明。（Jonathan B. Rosenberg著，吉川邦夫译，ASCII）

黑客的兴趣所在

说到Binary可举二进制，说到二进制则想到bit，这是一本包括bit单位的操作，细微到各种各样的操作都有介绍的好书。（ヘンリー・S・ウォーレン、ジュニア著，亀泽彻、铃木贡、赤池英夫、葛毅、藤波顺久、玉井浩译，SIB · Access）

安全防护

处理信息安全技术的书，在第3章“Linux逆向工程”中，有许多知识可供阅读本书时参考。例如nm、gdb、lsof、ltrace、 objdump等基本的工具的使用方法的说明。逆向assemble结果的阅读方法的说明，利用ptrace(2)的简单工具的开发例子。GNU BFD（特别是libopcodes.a）的活用例子都有写到。原书是日语版，同时封面是大力士的图片。(Cyrus Peikari、Anton Chuvakin著，西原启辅审译，依藤真浩、岸信之、进藤成纯译，O'Reilly · Japan)

JIS X 3010:2003 程序语言 C

最新的C语言标准ISO/IEC 9899:1999，通称:C99，可以说是学习Hack C编译器时必备的书 。在这本标准中，包含许多难以理解的微妙部分都是用C语言记载的。例如：在C语言中的整数和浮点数的精度的运算规则在Web上找不到归纳完整的说明，但只要阅读这个规格就能理解。本书的**[Hack #43]**用-ftrapv检测出整数运算的overflow，**[Hack #47]**使bitmask常数无符号化，**[Hack #99]**由于结果变得无限大或NaN的运算生成信号等在成对Hack的更深入的理解上有很大帮助。从本财团法人日本标准协会WebStore (*http://www.web store.jsa.or.jp/*) 谁都可以购买。

Internet

How To Write Shared Libraries

是与Linux的共享库（动态共享对象）相关的决定性文档。从把ELF二进制导入Linux的历史经过开始，共享库的优点、不足之处、动态链接的组成、性能的改善方法等，浓缩成了多种话题。可以说搜罗了运行共享库的Hack时要掌握的必要信息。作者Ulrich Drepper的网站（*http://people.redhat.com/~drepper/*）中登载了与Linux的thread库和SELinux都有关的重要信息，因而这是Binary Hacker非常关心的网站。(*http://people.redhat.com/~drepper/dsohowto.pdf*)

The Single UNIX Specification, Version 3

Unix的规范，由以下4个部分（volume）组成。

- Base Definitions (XBD)：术语定义、文件、文件认可等的Unix的基本概念解说，以及面向C语言头文件的定义等。

- System Interfaces (XSH)：与信号、socket、线程等的interface有关的全部说明，以及每个函数和系统调用的定义等。
- Shell and Utilities (XCU)：ls、cat等各个命令的定义，以及shell script的语法定义等。
- Rationale (XRAT)：在以上3个部分中没有被好好归纳的部分以及制定标准经过的说明等。

The Single Unix Specification是对Unix的深入学习非常有用的文档，特别是XSH的API一览表，经常会有意外的发现。在Linux distribution部分，收录了作为man的XSH和XCU的内容。尝试着寻找节0p（帧头）、1p（命令）、3p（函数及系统调用）的man。另外，在mozdev.org中，分配着为了从Firefox的检索bar中检索该规范的插件，从mozdev.org的首页开始用“Single Unix Specification”开始检索。(*http:// www.unix.org/single_unix_specification/*)

arbitrary unix stuff

登载各种Unix的#!的安装和各种shell的内部组成、echo命令的运作等与Unix相关的通用信息的网站，也是面向对Unix的各种功能有兴趣者的网站。(*http://www.in-ulm.de/~mascheck/various/*)

GCC的说明书

关于GCC的启动选项和C/C++的扩展功能的文档，看一遍启动选项和__attribute__扩展部分应该没有什么损失，在线汇编程序的说明书也在这里。

- 附属于gcc-2.95.3的Texinfo形式说明书的日文翻译。(*http:// www.sra.co.jp/wingnut/gcc/*)
- 英文的最新版 (*http://gcc.gnu.org/onlinedocs/gcc/*)
- 关于命令列选项

  GCC Command Options (*http://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html*)
- 关于__attribute__.

  Attribute Syntax (*http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html*)

C++ Attributes（*http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html.*）

Type Attributes（*http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html*）

Function Attributes（*http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html*）

Variable Attributes（*http://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html*）

- 关于在线汇编程序

  Assembler Instructions with C Expression Operands（*http://gcc.gnu.org/onlinedocs/ gcc/Extended-Asm.html.*）

  Constraints for asm Operands（*http://gcc.gnu.org/onlinedocs/gcc/Constraints.html*）

- 其他

  Pragmas Accepted by GCC.（*http://gcc.gnu.org/onlinedocs/gcc/Pragmas.html.*）

GNU 工具链接 info

是在本书中经常出现的固定编号软件的说明书。要注意Web上的说明书并不一定是最新的。

- info libc（*http://www.gnu.org/software/libc/manual/html_node/index.html*）
- info binutils（*http://www.gnu.org/software/binutils/manual/html_node/binutils_toc.html*）
- info gdb（*http://www.gnu.org/software/gdb/documentation/*）
- info bfd（*http://www.gnu.org/software/binutils/manual/bfd-2.9.1/html_node/bfd_toc.html*）

GCC Wiki

集中了与GCC相关的信息的Wiki，登载了多与GCC开发相关的信息和与GCC最新功能的话题。（*http://gcc.gnu.org/wiki/*）

comp.lang.c Frequently Asked Questions

用新闻组comp.lang.c生成的与C语言相关的FAQ，被称为“C FAQ”。恰当地指出了初、中级读者在C语言中经常容易出错的几点。也有与此相关的书籍。(*http://www.c-faq.com/*)

微处理器结构手册

IA-32（x86）、EM64T、IA-64（IPF）

- 《Inter日语技术资料的下载页》(*http://www.intel.com/jp/developer/download/index.htm*)

  汇集了Intel的说明书、application note\data sheet的日文翻译的网页。在开发软件中必不可少的“IA-32 Intel Architecture Software Developer's Manuals”、“Itanium Architecture Software Developer'sManauls”，可以下载最优化手法的日文说明，也有面向最新英文版文档的链接。

AMD x86、AMD64

- 《AMD Developer Central - Documentation》(*http://developer.amd.com/documentation.aspx*)

  正宗AMD64的命令集规范《AMD64 Architecture Programmer's Manual》以及面向Athlon64和Opteron的最优化向导。

Alpha

- 《Alpha technical documentation library》(*http://h18002.www1.hp.com/alphaserver/technology/chip-docs.html*)

  介绍了Alpha 21x64系列命令集的《Alpha Architecture Handbook》、《Compiler Writer's Guide for the Alpha 21264》以及各个主要的规范可以下载。

- 《Tru64 UNIX Version 4.0F Online Documentation》(*http://h30097.www3.hp.com/docs/pub_page/V40F_DOCS.HTM*)。有关于“Digital UNIX Assembly Language Programmer's Guide”这个Alpha的ABI的重要资料。

ARM

- 《ARM：manual download》*http://www.jp.arm.com/document/manual/*

可下载各种日文说明书的页面（免费登录）。命令集的解说包含在《RealView 代码生成 v2.0 asembler guard》中。

MIPS

- 《MIPS IV Instruction Set》(*http://techpubs.sgi.com/library/tpl/cgi-bin/download. cgi?coll=hdwr&db=bks&docnumber=007-2597-001*)

  介绍了命令集。

- 《MIPSpro N32 ABI Handbook》(*http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=0650&db=bks&docnumber=007-2816-005*)

  IRIX 的 ABI 的解说。

- 《MIPSpro Assembly Language Programmer's Guide》(*http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=0650&db=bks&docnumber=007-2418-006*)

  关于 assembler 的编程方法的解说。

- 《MIPS Technologies, Inc》(*http://www.mips.com/*、日文：*http://www.mips.jp/*)

  MIPS32、MIPS64 的规范。

PA-RISC

- 《PA-RISC 1.1 architecture and instruction set reference manual》(*http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,958,00.html*)

- 《PA-RISC 2.0 Architecture》(*http://h21007.www2.hp.com/dspp/tech/tech_TechDocument DetailPage_IDX/1,1701,2533,00.html*)

PowerPC

- 《IBM Microelectronics - PowerPC》(*http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC*)

  可下载命令集规范以及 IBM 的 PowerPC 进程的用户说明书。

- 《Freescale Semiconductor - PowerPC Processors》(*http://www.freescale.com/powerpc*)

在Apple公司的Macintosh上提供了PowerPC的半导体部门拆分出来的独立公司的网页。可从各项文档中下载每个进程的参考说明书等。

SH

- 《Renesas Technology 文献》*http://japan.renesas.com/fmwk.jsp?cnt=Documentation.jsp&fp=/products/mpumcu/superh_family/&title=%E3%83%89%E3%82%AD%E3%83%A5%E3%83%A1%E3%83%B3%E3%83%88&lid=2#*

  软件说明书

  命令集的解说以及各种日文的说明书可以下载。

SPARC

- 《SPARC International, Inc. - Standards》(*http://www.sparc.org/standards.html*)

  SPARC标准团体的网页。命令集《The SPARC Architecture Manual V8 & V9.》和ABI规范可以下载。

- 《UltraSPARC Processors Document》(*http://www.sun.com/processors/documentation.html*)

  Sun Microsystems 的 UltraSPARC 进程的 user guard 可以下载。

- 《OpenSPARC.net》(*http://opensparc.sunsource.net*)

  UltraSPARC T1 的主要的进程代码的规范被开源的OpenSPARC Project的网页。UltraSPARC的构造和规范书籍已假想化功能的规范都可以下载。

S/370

- 《ESA/390 Principles of Operation.》(*http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9AR006/CCONTENTS*)

- 《z/Architecture Principles of Operation》(*http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9ZR000/CCONTENTS*)

  《Principles of Operation》(PoO)是IBM System/360(S/360的结构)说明书。虽然是产品说明书,但可以作为结构的教科书来广泛使用。S/360随着S/370、ESA/390、ZSeries的进步也相应改写了PoO。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引