O'REILLY®

Learning eBPF

Programming the Linux Kernel for Enhanced Observability, Networking, and Security

Compliments of

ISOVALENT

Liz Rice

eBPF 学习指南

为增强可观测性、网络和安全性编码 *Linux* 内核

2023.08.26

序

在云原生社区及相关技术领域，eBPF 已成为近年来最热点的话题之一。以 eBPF 为平台，已构建出了一代全新的强大工具和项目，涵盖网络、安全和可观测性等领域，并且还在不断地涌现出更多类似的产品。相比于其前身，这些工具和项目在性能和准确性方面表现得更好。eBPF 相关的会议，如 eBPF 峰会和云原生 eBPF 日吸引了成千上万的与会者和观众。截至目前为止，eBPF 的Slack社区也已经拥有超过 14,000 名成员。

为什么 eBPF 被选作众多基础设施/工具的构建基础？它是如何实现了性能上的提升？从性能追踪到网络流量加密等各种不同的环节中，eBPF 又如何发挥作用？

本书旨在通过让读者了解 eBPF 的工作原理，并介绍编写 eBPF 代码的方法来解答上述问题。

本书写给谁？

本书适用于对 eBPF 感到好奇并希望了解其工作原理的开发人员、系统管理员、运维人员和学生。本书将为那些希望自己学习探索并编写 eBPF 程序的人提供基础知识。由于 eBPF 为全新一代观测性和工具提供了优秀的平台，eBPF 领域在未来几年内很可能会出现就业机会。

即时你不一定要自己写 eBPF 代码，本书对你仍然有用。如果你从事运维、安全或任何涉及软件基础设施的工作，你可能已经或未来几年会遇到基于 eBPF 开发的工具。如果你了解这些工具的原理，就能更好地使用它们。例如，如果你了解事件如何触发 eBPF 程序，对于 eBPF 工具显示的性能指标，你能迅速且准确地理解它到底在测量什么。如果你是软件开发人员，可能也会接触到一些基于 eBPF 的工具，例如，你正在对应用程序进行性能调优，可以使用类似 Parca 的工具生成函数调用时间的火焰图。如果你正在评估安全工具，本书将帮助你理解 eBPF 的优势，并避免不成熟的使用方式以提升抵御攻击的效果。

即使你现在不使用 eBPF 工具，我也希望本书能让你对 Linux 的某些领域形成自己的理解，即使这些领域你可能之前都没有碰过。大多数开发人员都认为内核作为底层是自然而然的，因为他们使用的编程语言具有高级抽象，使他们能够专注于应用程序开发的工作，而这已经足够困难了！程序员使用调试器和性能分析工具来帮助自己高效地完成工作。了解调试器或性能工具的内部工作原理可能很有趣，但并非必要。然而，对很多人来说，深入了解更多是有趣且充实的。当然，大多数人使用 eBPF 工具是无需了解它们是如何构建

的。亚瑟 · 克拉克（Arthur C. Clarke）曾写道：“任何足够先进的技术都和魔法无异”，但就个人而言，我喜欢深入挖掘技术细节并弄清楚原理。

本书内容

eBPF 在飞速发展，这样，编写一本不会频繁更新的参考资料其实是相当困难的。然而，一些基本原理和原则不太可能发生重大变化，所以这些也是本书讨论的些内容。

第一章阐述了为什么 eBPF 如此强大，并解释了在操作系统内核中运行自定义 eBPF 程序的能力是如何实现众多令人兴奋的功能的。

第二章更加具体，通过一些 “Hello World” 示例代码介绍了 eBPF 程序和映射 (Map) 的概念。

第三章深入讨论了 eBPF 程序及其在内核中的运行方式，第四章探讨了用户空间应用程序与 eBPF 程序之间的接口。

近年来，eBPF 面临的一个重大挑战是跨内核版本的兼容性问题。第五章介绍了解决这个问题的方法：“编译一次，在任何地方运行”（CO-RE）。

验证过程是区分 eBPF 和内核模块最重要的特点之一。第六章将介绍 eBPF 验证器。

在第七章中，您将对多种类型的 eBPF 程序及其附加点 (Attachment point) 有所了解。其中许多附加点位于网络堆栈中，第八章将更详细地探讨 eBPF 在网络功能中的应用。第九章介绍了如何使用 eBPF 构建安全工具。

如果您想编写与 eBPF 程序交互的用户空间程序，有许多库和框架可供选择。第十章概述了各种编程语言提供的工具。

最后，在第十一章中，我将展望 eBPF 未来发展，并告诉读者一些可能在 eBPF 领域出现的新动态。

基本要求

这本书假设您熟悉 Linux 上的基本 Shell 命令和使用编译器编译源码。书中包含一些简单的 Makefile 示例，因此，您至少也该对如何使用 make 有最基本的了解。

书中有很多 Python、C 和 Go 的代码示例。要理解这些示例，并不需要深入了解这些编程语言，但如果您熟悉这些代码，那么将从本书中获益更大。最后，还假设您熟悉指针的概念，指针用于标识内存位置。

示例代码和练习

书中有许多代码示例。如果您想亲自尝试这些例子，可以在 https://github.com/lizrice/learning-ebpf 找到相应的 GitHub 代码仓库，并按照安装和运行说明进行操作。

我在大多数章节的末尾提供了练习，帮助您通过这些示例或编写自己的程序来探索 eBPF 编程。

由于 eBPF 不断演进，能使用的功能取决于您所运行的 Linux 内核版本。在较早版本中存在诸多限制可能在新版本中已被取消或放宽。Iovisor 项目提供了关于不同 BPF 功能添加的内核版本的描述，在本书中，我尽力标注了特定功能添加的时间。本书的示例是使用 5.15 版本的 Linux 内核进行测试的，而在撰写本文时，一些流行的 Linux 发行版尚未支持该新内核版本。如果您在本书出版后不久就读到了本书，可能会发现一些功能在您所用的 Linux 内核上无法工作。

eBPF 只能用于 Linux 吗?

eBPF 最初是为 Linux 开发的。实际上，没有任何特殊原因阻止将其应用于其他操作系统，事实上，微软一直在为 Windows 开发 eBPF 实现。我在第 11 章中简单地讨论了下这点，但在本书的其余部分，我只专注于 Linux 上的实现，并且所有的示例都基于 Linux 系统。

排版约定

本书使用以下排版约定：

- 斜体

用于表示新术语、URL、邮件地址、文件名和文件扩展名。

- 等宽字体

用于代码，以及在段落中用于引用程序元素（如变量或函数名、数据库、数据类型、环境变量、语句和关键字）。

- **等宽粗体**

用于显示用户应该按照字面意义输入的命令或其他文本。

- 等宽斜体

用于显示应由用户提供的值或由上下文确定的值替换的文本。

白灰色框表示一个提示或建议！

橘黄色框表示一般的注意事项！

霏红色框表示警告或注意事项！

使用代码示例

你可以在 https://github.com/lizrice/learning-ebpf 下载补充材料（代码示例、练习等）。

如果有技术问题或使用代码时遇到问题，请发送邮件到 bookquestions@oreilly.com。

本书的目的是帮助你完成工作。一般来说，如果提供了示例代码，您可以在自己的程序和文档中使用。除非复制了代码的重要部分，否则无需联系我们以获得许可。例如，编写一个使用本书多个代码片段的程序不需要许可。销售或分发 O'Reilly 图书中的示例代码需要许可。引用本书并引用示例代码回答问题不需要许可。将本书中的大量示例代码合并到产品文档中需要许可。

我们当然感谢你在作品中的致谢，但通常并不要求署名。署名通常包括书名、作者、出版商和 ISBN。例如："Learning eBPF by Liz Rice (O' Reilly)。Copyright 2023 Vertical Shift Ltd., 978-1-098-13512-6。"

如果你认为对示例代码的使用超出了合理范围或上述许可，请随时通过 permissions@oreilly.com 与我们联系。

O' Reilly 在线学习

在过去的 40 多年里，O'Reilly 一直在提供技术和商业培训、知识和洞见以帮助企业取得成功。

我们专家和创新者群体通过图书、文章和在线学习平台分享了他们的见解和专业知识。O'Reilly 的在线学习平台为您提供按需访问的实时培训课程、深入学习路径、交互式编码环境以及来自 O'Reilly 和其他 200 多家出版商的大量文本和视频资源。了解更多信息，请访问 https://oreilly.com。

与我们联系

请将有关本书的评论和问题发送至出版商：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938（美国或加拿大）

707-829-0515（国际或当地）

707-829-0104（传真）

我们为本书设有一个网页，其中列出了勘误、示例和其他信息。您可以访问此页面：https://oreil.ly/learning-eBPF。

发送电子邮件至 bookquestions@oreilly.com 以提出评论或询问有关本书的技术问题。

有关我们的图书和课程的新闻和信息，请访问 https://oreilly.com。

在 LinkedIn 上找到我们：https://linkedin.com/company/oreilly-media。

在 Twitter 上关注我们：https://twitter.com/oreillymedia。

在 YouTube 上观看我们：https://youtube.com/oreillymedia。

致谢

我要感谢为撰写本书做出巨大贡献的许多人：

• 我的技术审稿人 Timo Beckers、Jess Males、Quentin Monnet、Kevin Sheldrake 和 Celeste Stinger 提供了详细而有建设性的反馈意见和改进示例的好点子，对此我非常感激。

• 我站在那些建立、推广和维护 eBPF 的巨人的肩上，其中包括 Daniel Borkmann、Thomas Graf、Brendan Gregg、Andrii Nakryiko、Alexei Starovoitov 以及无数其他为社区做出贡献的人，他们不仅贡献了代码，还提供了演讲和博客文章。

• 感谢我在 Isovalent 的才华横溢而可爱的同事们，其中许多人是 eBPF 和内核专家，我从他们身上学到很多。

• 还要感谢 O'Reilly 团队，特别是我的编辑 Rita Fernando，在写作过程中给予了我大力支持，以及在计划方面的帮助，使得本书能按时完成；还有 John Devins，他鼓励我首次撰写这本书。

• Phil Pearl 不仅对内容给予了有益的反馈，还确保我的吃饭和休息。我将永远感激他的支持和鼓励。

我还要感谢多年来花时间对我的工作给予鼓励性评论的所有人，无论是在活动中还是在社交媒体上。知道我写或录制的东西帮助他人掌握了技术、概念或激发人们建立或书写自己作品的欲望，是一种极大的鼓励，谢谢你们！

目录

第一章　什么是 eBPF

eBPF 是一项革命性的内核技术，允许开发人员编写自定义代码，并能够动态加载到内核中，从而改变内核的行为方式（如果对内核的概念不太了解，不用担心，在本章中我很快会介绍）。

这项技术使得新一代高性能的网络、可观测性和安全性工具成为可能。正如你会看到的，如果要在应用程序里使用基于 eBPF 的工具，你不需要修改或重新配置应用程序，这都得益于 eBPF 在内核中发挥的作用。

使用 eBPF，你可以实现以下功能：

- 对系统的几乎任何方面进行性能追踪
- 高性能的网络编程，并内置可观测性功能
- 检测和防止恶意活动

下面从伯克利数据包过滤器（Berkeley Packet Filter）开始简要回顾下 eBPF 的历史。

1.1　eBPF 的起源：伯克利数据包过滤器

今天称之为"eBPF" 的技术源于 1993 年，由劳伦斯伯克利国家实验室的 Steven McCanne 和 Van Jacobson 撰写的一篇论文 [1] 中介绍的 BSD Packet Filter。这篇论文讨论了一个伪机器，可以运行过滤器，这些过滤器是用来确定是否接受或拒绝网络数据包的程序。这些程序使用了 BPF 指令集来编写，这是一组类似汇编语言的通用 32 位指令集。以下是直接摘自那篇论文的一个示例代码：

```
ldh   [12]
jeq   #ETHERTYPE IP, L1, L2
L1:   ret      #TRUE
L2:   ret      #0
```

这段小小的代码过滤掉了非 Internet 协议（IP）数据包。该过滤器的输入是一个以太网数据包，第一条指令（ldh）从该数据包中加载从第 12 个字节开始的 2 字节值。在下一条指令（jeq）中，该值与表示 IP 数据包的值进行比较。如果匹配，执行跳转到标记为 L1 的指令，并且通过返回一个非零值（这里标识为 #TRUE）来接受该数据包。如果不匹配，则该数据包不是 IP 数据包，通过返回 0 来拒绝。

你可以想象（或者参考论文以找到更多示例），还可以编写更复杂的过滤器程序，根据数据包的其他指标来做决策。重要的是，过滤器的作者可以编写自定义程序在内核中执行，这就是 eBPF 的核心所在。

BPF 最初代表"Berkeley Packet Filter"，于 1997 年首次引入到版本为 2.1.75 的 Linux 内核中，在 tcpdump 工具中作为一种高效捕获拟跟踪数据包的方式。

到 2012 年，内核版本 3.5 中又引入了 seccomp-bpf，这使得可以使用 BPF 程序决定是否允许用户空间应用程序进行系统调用，我将在第 10 章中详细探讨这一点。这个引入使得最初只能用于数据包过滤的 BPF 逐渐演变为了如今的通用平台。也是从那时起，名称中的"packet filter" 这个词就不那么确切了！

1.2 从 BPF 到 eBPF

从 2014 年开始，BPF 演变成了"extended BPF" 或"eBPF"，这始于内核版本 3.18。这个演变涉及几大重要变化：

- BPF 指令集进行了全面改进，以在 64 位机器上更高效，并且解释器完全重写。
- 引入了 eBPF 映射（maps），这是一种数据结构，可以被 BPF 程序和用户空间应用程序访问，允许它们之间共享信息。你将在第 2 章学习关于映射的内容。
- 添加了 bpf() 系统调用，以便用户空间程序可以与内核中的 eBPF 程序进行交互。你将在第 4 章中了解这个系统调用。
- 添加了几个 BPF 帮助函数。你将在第 2 章看到一些示例，并在第 6 章中了解更多细节。
- 添加了 eBPF 验证器，以确保 eBPF 程序的安全运行。这在第 6 章中会进行讨论。

这些变化为 eBPF 奠定了基础，但 eBPF 项目的开发并没有放缓节奏，此那时起，eBPF 又逐渐发展出了许多重要的特性。

1.3 eBPF 演进历程

自 2005 年起，Linux 内核中存在一个名为kprobes（内核探测点）的特性，允许在内核代码的几乎任何指令上设置钩子。开发人员可以编写内核模块，将函数附加到 kprobes 上进行调试或性能测量。

2015 年，又实现了将 eBPF 程序附加到 kprobes 的能力，这是 Linux 数据追踪方式发生革命性转折的起点。与此同时，内核的网络堆栈中也开始添加钩子，允许 eBPF 程序负责更多的网络功能。我们将在第 8 章中看到更多相关内容。

到了 2016 年，基于 eBPF 的工具已经开始在生产系统中使用。Brendan Gregg在 Netflix 的追踪工作在基础设施和运维领域广为人知，他说eBPF 给 Linux 带来了" 超能力"。同年，Cilium 项目宣布成立，成为第一个在容器环境中使用 eBPF 替换数据路径的网络项目。

在随后的一年，Facebook（现为 Meta）将Katran项目开源。Katran 是一个第四层负载均衡器，满足了 Facebook 对高可扩展性和高速解决方案的需求。自 2017 年以来，每个

发送到 Facebook.com 的数据包都经过eBPF/XDP。对我个人而言，这一年在得克萨斯州奥斯汀的 DockerCon 上看到了 Thomas Graf 关于 eBPF 和Cilium项目的演讲，点燃了我对该项技术所带来的可能性的憧憬。

2018 年，eBPF 成为 Linux 内核中的一个独立子系统，由 Isovalent 的Daniel Borkmann和 Meta 的Alexei Starovoitov担任维护人员（后来又加入了 Meta 的Andrii Nakryiko）。同年，引入了 BPF 类型格式（BTF），使得 eBPF 程序更具可移植性。我将在第 5 章中详细探讨这个主题。

2020 年引入了 LSM BPF，允许将 eBPF 程序附加到 Linux 安全模块（LSM）内核接口上。这表明已经确定了 eBPF 的第三个重要用途：除网络和可观测性外，eBPF 又成为了安全工具的理想平台。

多年来，由 300 多名内核开发人员和众多贡献者为相关用户空间工具（如 bpftool，我将在第 3 章中介绍）、编译器和编程语言库做出的努力，eBPF 的功能大幅增强。eBPF 程序曾被限制最多执行 4,096 条指令，但现在这个限制已经增长到了 1 百万，并且通过对尾调用和函数调用的支持（你将在第 2 章和第 3 章中看到），该限制已经变得无关紧要。

要深入了解 eBPF 的历史，没有比一直从事该项目的维护人员更合适的了。Alexei Starovoitov 从 BPF 在软件定义网络（SDN）中的起源开始对 BPF 的历史进行了一场引人入胜的演讲。在这次演讲中，他讨论了早期 eBPF 补丁被接受到内核的策略，并揭示了 eBPF 的正式诞生日是 2014 年 9 月 26 日，这一天标志着第一批涵盖了验证器、BPF 系统调用和映射的补丁被接受。Daniel Borkmann 也讨论了 BPF 的历史以及其演变出的网络和追踪功能。我强烈推荐他的演讲《eBPF 和 Kubernetes：微服务扩展的帮手》，其中充满了有趣的信息。

1.4　命名困难

eBPF 的应用远远超出了数据包过滤的范畴，以至于这个首字母缩写现在基本上没有实际意义，它已经成为一个独立的术语。由于当今广泛使用的 Linux 内核都支持“扩展”部分，eBPF 和 BPF 这两个术语在很大程度上可以互换使用。在内核源代码和 eBPF 编程中，常用的术语是 BPF。例如，正如我们将在第 4 章中看到的，用于与 eBPF 交互的系统调用是 bpf()，帮助函数以 bpf_ 开头，不同类型的 (e)BPF 程序使用以 BPF_PROG_TYPE 开头的名称进行标识。在内核社区外，名称“eBPF”似乎已经被广泛接受，例如在社区网站ebpf.io和基金会eBPF Foundation的名称中。

1.5　Linux 内核

要理解 eBPF，你需要牢固掌握 Linux 中内核和用户空间之间的区别。我在报告《什么是 eBPF?》[2] 中涵盖了这一点，并且已将其中的一些内容放到了接下来的几段中。

Linux 内核是应用程序及其运行的硬件之间的软件层。应用程序在一个称为用户空间的非特权层中运行，无法直接访问硬件。相反，应用程序使用系统调用（syscall）接口发出

请求，请求内核代其执行操作。硬件访问涉及读写文件、发送或接收网络流量，甚至只是访问内存。内核还负责协调并发进程，维持多个应用程序同时运行。如图 1-1。

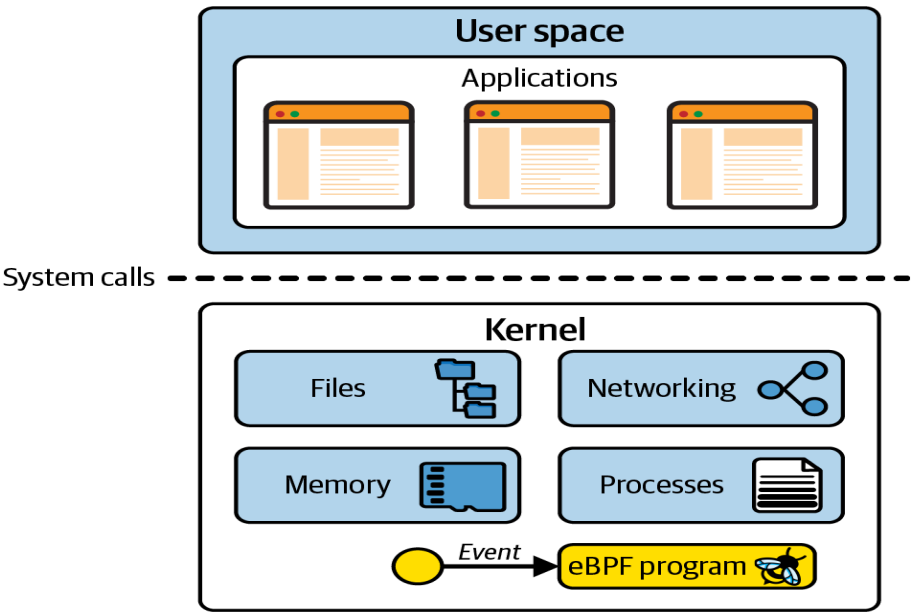

图 1.1: 用户空间中的应用程序使用系统调用接口向内核发出请求

应用程序开发人员通常不直接使用系统调用接口，因为编程语言为我们提供了更高级的抽象和标准库，这些库提供了更容易编程的接口。因此，很多人对内核在程序运行时所做的工作并不了解。如果您想了解内核被调用的频率，可以使用 strace 工具来展示应用程序执行的所有系统调用。

使用 cat 命令将单词 “hello” 显示到屏幕上涉及到了 100 多个系统调用:

```
$ strace -c echo "hello"
hello
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
24.62    0.001693          56        30        12 openat
17.49    0.001203          60        20           mmap
15.92    0.001095          57        19           newfstatat
15.66    0.001077          53        20           close
10.35    0.000712         712         1           execve
3.04     0.000209          52         4           mprotect
2.52     0.000173          57         3           read
2.33     0.000160          53         3           brk
2.09     0.000144          48         3           munmap
1.11     0.000076          76         1           write
```

```
  0.96    0.000066          66        1        1 faccessat
  0.76    0.000052          52        1          getrandom
  0.68    0.000047          47        1          rseq
  0.65    0.000045          45        1          set_robust_list
  0.63    0.000043          43        1          prlimit64
  0.61    0.000042          42        1          set_tid_address
  0.58    0.000040          40        1          futex
------ --------- ---------- ------- ------- ----------------
100.00  0.006877          61      111       13 total
```

由于应用程序在很大程度上依赖于内核，如果能够观察应用程序与内核的交互，就能了解程序的行为。通过 eBPF，我们可以向内核中添加工具来获取这些交互信息。

例如，如果能够截获打开文件的系统调用，就能够准确地查看应用程序访问了哪些文件。但是，该如何进行截获呢？考虑一下，如果通过修改内核，在每次调用系统调用时添加新的代码以创建某种输出，会涉及哪些工作？

1.6　向内核添加新功能

Linux 内核非常复杂，在撰写本文时，其代码量约为3000 万行。对于任何代码库的更改，都需要对现有代码有一定的了解，所以除非你已经是一名内核开发者，否则这可能是一个巨大的挑战。

此外，如果要将你的更改贡献到上游，你将面临的挑战不仅仅是技术方面的。Linux 是一个通用的操作系统，用于各种环境和情况。这意味着，如果你希望自己的更改成为官方 Linux 发布的一部分，仅仅编写可运行的代码是不够的。代码还必须被社区（尤其是 Linux 的创造者和主要开发者 Linus Torvalds）接受，认为这个更改对所有人有益。实际上，这并非是铁定的事-提交的内核补丁只有三分之一被接受 [3]。

假设你已经找到了一个截获打开文件系统调用的好方法，经过几个月的讨论和艰苦的开发，更改被内核接受了。太棒了！但是，该方法在其他人机器上可用之前还需要等多长时间？

Linux 内核每两到三个月会发布一个新版本，但即使一个更改已经进入其中一个版本，它离大多数人的生产环境都还有一段时间。这是因为大多数人并不直接使用 Linux 内核，而是使用的诸如 Debian、Red Hat、Alpine 和 Ubuntu 等 Linux 发行版，这些发行版负责将 Linux 内核的某个版本与其他组件打包在一起。你可能会发现，你喜欢的发行版使用的可能是几年前的内核版本。

比如，很多企业用户使用 Red Hat Enterprise Linux（RHEL）。在撰写本文时，当前版本是 RHEL 8.5，发布日期为 2021 年 11 月，它使用的是 Linux 4.18 版本。这个内核发布于 2018 年 8 月。

如图 1-2 中的漫画所示，将新功能从构思阶段引入到 Linux 内核再到生产环境需要几

年的时间。

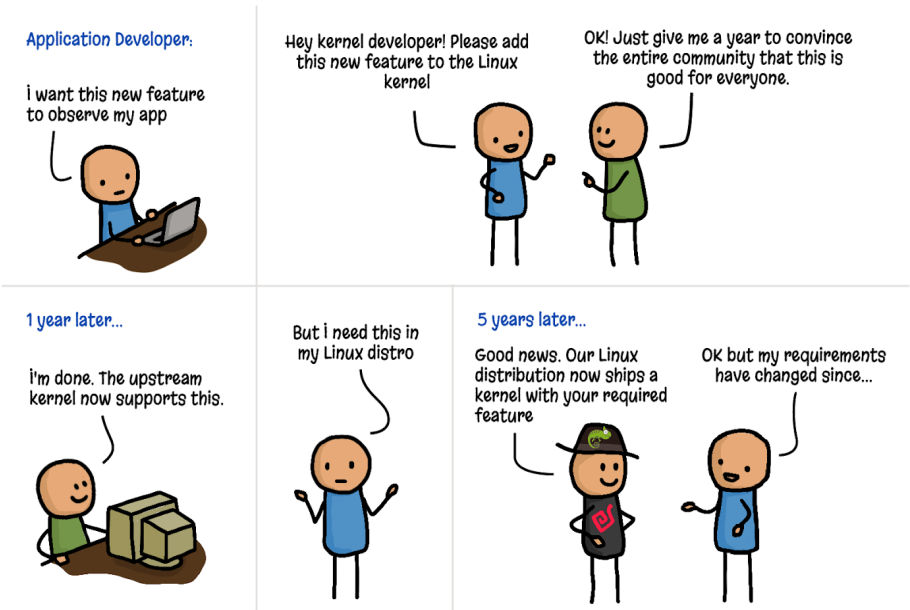

图 1.2: 给内核添加新功能（由 Vadim Shchekoldin 和 Isovalent 绘制）

1.7 内核模块

如果你不想等几年才将更改纳入内核，这里还有另一种选择。Linux 内核设计时考虑到了引入内核模块，这些模块可以按需加载和卸载。如果你想改变或扩展内核行为，编写一个模块当然是一种可行的方法。内核模块可以分发给其他人使用，独立于官方的 Linux 内核发布，因此不需要被接受到上游代码库中。

这里最大的挑战是，仍然需要内核编程。长期以来，用户对使用内核模块一直非常谨慎，一个很明显的原因是：如果内核代码崩溃，会导致机器上的所有内容都崩溃，用户如何确信一个内核模块是安全运行的呢?

”安全运行”不仅仅意味着不崩溃-用户还想知道一个内核模块在安全方面是安全的。它是否包含攻击者可以利用的漏洞？我们是否相信模块的作者不会在其中放入恶意代码?因为内核是特权代码，它可以访问机器上的所有内容，包括所有数据，所以内核中的恶意代码会引起严重的问题。这种担心同样出现在内核模块。

内核的安全性是 Linux 发行版需要很长时间才能整合新版本的一个重要原因。如果其他人在各种情况下运行某个内核版本已经数月甚至数年了，如果有问题，应该都暴露了。发行版维护者可以相对有信心地确保他们向用户/客户提供的内核是经过加固的，也就是说它是安全运行的。

eBPF 提供了一种非常不同的安全方法：eBPF 验证器，它确保只有在安全的情况下加载 eBPF 程序-它不会导致机器崩溃或陷入死循环，并且不允许数据泄露。我将在第 6 章中更详细地讨论验证过程。

1.8　动态加载 eBPF 程序

eBPF 程序可以动态地加载到内核中，并随时移除。一旦 eBPF 程序被附加到一个事件上，无论是什么引发了该事件，程序都会被触发。例如，如果你将一个 eBPF 程序附加到打开文件的系统调用上，那么无论哪个进程打开文件，该程序都会被触发。无论该进程在程序加载时是否已经运行，都没有关系。这种方式，比升级内核然后重启机器要好得多。

这种方式使得使用 eBPF 的可观察性或安全工具拥有巨大的优势-它能立即获得机器上发生的所有事件可见性。在容器环境中，这包括对容器内所有正在运行的进程以及主机机器上的进程的可见性。我将在本章后面详细讨论这对云原生部署的影响。

此外，正如图 1-3 所示，可以通过 eBPF 快速创建新的内核功能，而无需要求其他 Linux 用户接受相同的更改。

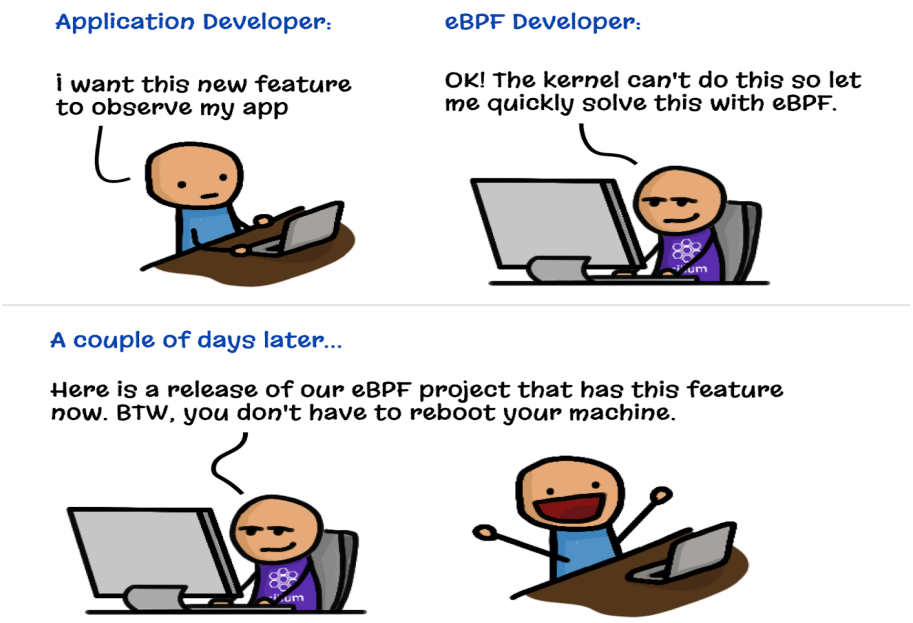

图 1.3: 使用 eBPF 给内核添加功能（由 Vadim Shchekoldin 和 Isovalent 绘制）

1.9　高性能 eBPF 程序

eBPF 程序是一种非常高效的添加工具到内核的方式。一旦被加载并进行即时编译（你将在第 3 章中了解到），该程序将作为本地机器指令在 CPU 上运行。此外，ePBF 程序不需要在处理事件时进行内核和用户空间之间的切换（这是一种昂贵的操作），因此也不会造成这部分开销。

2018 年的一篇论文 [4] 描述了 eXpress Data Path（XDP），其中包含了 eBPF 在网络中实现性能改进的一些示例。例如，在 XDP 中实现路由与常规 Linux 内核实现相比，“性能提升了 2.5 倍”，而在负载均衡方面，“XDP 相对于 IPVS 提供了 4.3 倍的性能增益”。

对于性能跟踪和安全观测，eBPF 的另一个优势是可以在将事件发送到用户空间之前，

在内核中对事件进行过滤，从而避免不必要的开销。毕竟，最初的 BPF 就是用来过滤特定的网络数据包的。如今，eBPF 程序可以收集系统中各种事件的信息，并使用复杂的、定制的过滤器将用户关心的特定子集信息发送到用户空间。

1.10 云原生 eBPF

现在，众多企业不是直接在服务器上程序。相反，他们都是采用云原生部署：采用容器、诸如 Kubernetes 或 ECS 之类的编排器，或者像 Lambda、云函数、Fargate 等无服务器方法。这些方法都使用自动化来选择每个工作负载该在哪个服务器上运行。在无服务器部署模式中，我们甚至不知道每个工作负载运行在哪个服务器上。

然而，服务器仍然参与其中，每个服务器（无论是虚拟机还是裸机）都运行一个内核。在容器中运行应用程序时，如果它们在同一（虚拟）机器上运行，则共享同一个内核。在 Kubernetes 环境中，这意味着在给定节点上的所有 Pod 中的所有容器都在使用同一个内核。用 eBPF 程序对该内核进行功能扩展后，所有在该节点上的工作负载都对这些 eBPF 程序可见，如图 1-4 所示。

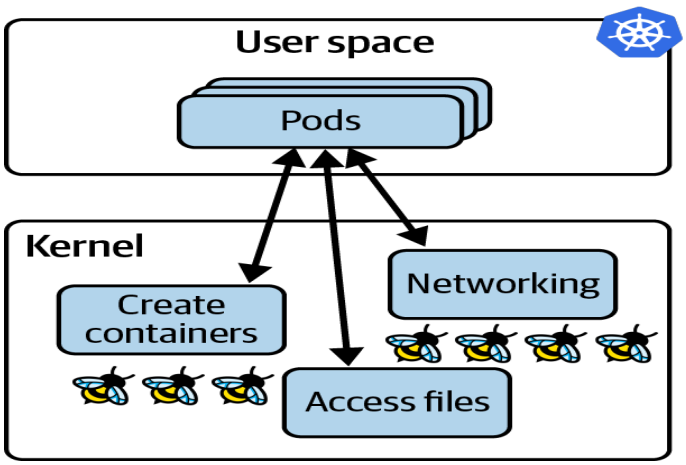

图 1.4: 内核中的 eBPF 程序可以看到在 Kubernetes 节点上运行的所有程序

在云原生计算中，eBPF 基于其对节点上所有进程的可见性以及动态加载程序的能力，为我们带来了真正的超能力：

- 无需修改应用程序，甚至不需要改变配置，就可以使用 eBPF 工具。
- 一旦程序被加载到内核并与事件关联，eBPF 程序就可以立即开始观测现有的应用程序进程。

与此相比，sidecar(边车模式) 用于将日志记录、跟踪、安全和服务网格等功能添加到 Kubernetes 应用程序中。在该模式下，各种功能作为一个容器被“注入”到每个应用程序 Pod 中。这个过程涉及修改定义应用程序 Pod 的 YAML 配置文件，添加 sidecar 容器的定义。这种方法比将这些功能添加到应用程序的源代码中更方便（在 sidecar 模式出现之前，

就需要这样做；例如，在应用程序中包含日志记录库，并在代码中适当的位置进行调用）。然而，sidecar 模式也有不足之处：

- 为了添加 sidecar 容器，需要重启应用程序 Pod。
- 需要修改应用程序的 YAML。通常这是自动化的过程，但如果出现问题，sidecar 容器就无法添加，这意味着 Pod 不会被添加上这些功能。例如，可以通过注释来指示一个部署，让入口控制器将 sidecar 容器的 YAML 添加到该部署的 Pod 规范中。但如果部署没有正确标记，sidecar 容器就无法添加。
- 当一个 Pod 中有多个容器时，它们可能在不同的时间达到就绪状态，其顺序可能无法预测。sidecar 的注入可能会显著延长 Pod 的启动时间，甚至导致竞态条件或其他不稳定情况。例如，Open Service Mesh 的文档描述了应用程序容器在 Envoy 代理容器就绪之前必须能够处理所有流量丢弃的情况。
- 如果将服务网格等网络功能作为 sidecar 实现，那么所有与应用程序容器之间的流量都必须经过内核中的网络堆栈才能到达网络代理容器，这会增加流量延迟，如图 1-5 所示。我将在第 9 章讨论如何使用 eBPF 提高网络效率。

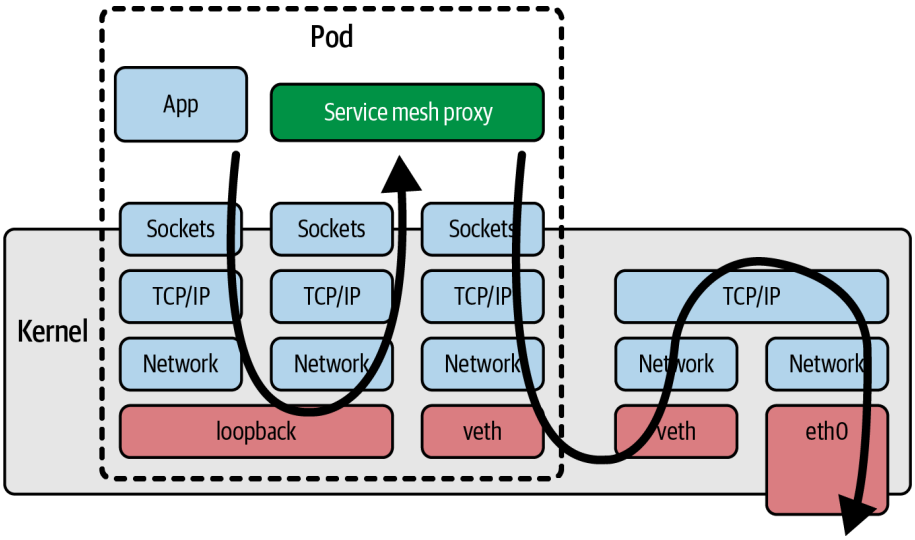

图 1.5: 使用服务网格代理 sidecar 容器的网络数据包路径

所有这些问题都是基于 sidecar 模式固有的。幸运的是，现在有了 eBPF，我们有了一种可以避免这些问题的新模型。此外，由于基于 eBPF 的工具可以看到（虚拟）机器上发生的所有情况，恶意行为就更难绕过。例如，如果攻击者成功在你的主机上部署了一个加密货币挖矿程序，他们不会将各种基于 sidecar 的观测功能也加上。如果你依赖于基于 sidecar 的安全工具来阻止应用程序进行意外的网络连接，只要没有注入 sidecar，那么就无法发现挖矿程序连接到矿池的情况。相比之下，基于 eBPF 实现的网络安全可以监控主机上的所有流量，因此可以轻松阻止这种加密货币挖矿行为。关于出于安全原因丢弃网络数据包的能力，我们将在第 8 章回到这个问题。

1.11 总结

希望本章让你对 eBPF 作为一个平台的强大之处有了一定了解。它允许改变内核的行为，提供构建定制工具或自定义策略的灵活性。基于 eBPF 的工具可以观察内核中的任何事件，因此可以观察到在（虚拟）机器上运行的所有应用程序，无论它们是否容器化。eBPF 程序还可以动态部署，允许实时改变行为。

到目前为止，我在概念层面上讨论了 eBPF。在下一章，我将更详细地探索基于 eBPF 的应用程序的组成部分。

第二章　eBPF 的 “Hello World”

在上一章中，我讨论了为什么 eBPF 如此强大，但如果你对 eBPF 程序的实际意义还没有明确的理解，那也没有关系。在本章中，我将用一个简单的 “Hello World” 示例让你更好地理解这点。

阅读过程中，你会了解到有好几种编写 eBPF 程序的库和框架。作为热身，我将首先向你展示一个最好理解的方法：BCC Python 框架。它提供了一种非常简单的方式来编写 eBPF 程序。我将在第 5 章中提到这个框架我并不推荐用于生产环境，特别是那些打算给其他用户使用程序。当然，学习原理嘛，这是第一步，用这个来学习其实非常适合。

如果你想自己尝试这段代码，可以在 https://github.com/lizrice/learning-ebpf 的 chapter2 目录中找到它。你可以在 https://github.com/iovisor/bcc 找到 BCC 项目，安装 BCC 的说明在 https://github.com/iovisor/bcc/blob/master/INSTALL.md。

2.1　BCC 中的 “Hello World”

下面是使用 BCC 的 Python 库编写的 eBPF “Hello World” 程序 hello.py 的源代码：

```
1  #!/usr/bin/python
2  from bcc import BPF
3
4  # This is the eBPF program
5  program = r"""
6  int hello(void *ctx) {
7      bpf_trace_printk("Hello World!");
8      return 0;
9  }
10 """
11
12 # Load the eBPF program
13 b = BPF(text=program)
14
15 # Attach the eBPF program to a tracepoint
```

```
16 syscall = b.get_syscall_fnname("execve")
17 b.attach_kprobe(event=syscall, fn_name="hello")
18
19 # Read and print the trace events
20 b.trace_print()
```

这段代码由两部分组成：eBPF 程序本身将在内核中运行，而一些用户空间代码将 eBPF 程序加载到内核并读取其生成的信息。如图 2-1 所示，hello.py 是这个应用程序的用户空间部分，而 hello() 才是在内核中运行的 eBPF 程序。

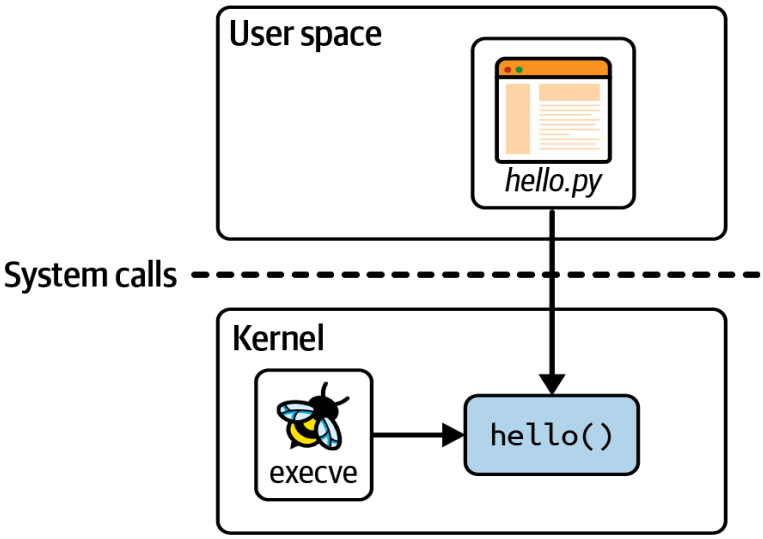

图 2.1: ”Hello World” 的用户空间和内核空间

让我们逐行解读这些代码。

第一行表示这是 Python 代码，可以运行它的程序是 Python 解释器（/usr/bin/python）。eBPF 程序本身是用 C 代码编写的，代码如下：

```
1 int hello(void *ctx) {
2     bpf_trace_printk("Hello World!");
3     return 0;
4 }
```

这段 eBPF 程序的作用是用辅助函数 bpf_trace_printk() 来打印一条消息。辅助函数是 “扩展” BPF 相对于其 “经典” BPF 的特性，是一组可以由 eBPF 程序调用与系统进行交互的函数，我将在第 5 章中进一步讨论它们。目前，你只需将它视为为了输出一行文本。

eBPF 程序在 Python 代码中被定义为一个名为 program 的字符串。在执行之前，需要对这个 C 程序进行编译，但 BCC 会为你代劳。（在下一章中，你将学习如何自己编译 eBPF 程序。）你只需要在创建 BPF 对象时将该字符串作为参数传递，如下一行代码所示：

```
b = BPF(text=program)
```

eBPF 程序需要附加到一个内核事件上。在这个例子中，我选择了附加到系统调用 execve 上，该系统调用用于执行程序。当这台机器上有新程序启动时，都会调用 execve()，从而触发 eBPF 程序。虽然”execve()” 是 Linux 中的标准接口名称，但在内核中它的函数名取决于芯片架构，BCC 只是为我们提供了一种统一方便的函数名称：

```
syscall = b.get_syscall_fnname("execve")
```

syscall 表示要附加的内核函数的名称。使用 kprobe 就可以将 hello 函数附加到该事件上，如下所示：

```
b.attach_kprobe(event=syscall, fn_name="hello")
```

此时，eBPF 程序已加载到内核并附加到一个事件上，因此当机器上启动新的程序时，该程序将被触发。Python 代码里只需读取内核输出的跟踪信息并将其显示在屏幕上就可以了：

```
b.trace_print()
```

trace_print() 函数将无限循环（可用 Ctrl+C 停止），打印出所有跟踪信息。

图 2-2 说明了这段代码的过程。Python 程序编译 C 代码，将其加载到内核中，并将其附加到 execve 系统调用的 kprobe 上。每当这台（虚拟）机器上的应用程序调用 execve() 时，都会触发 eBPF 的 hello() 程序，该程序将一行跟踪信息写入特定的伪文件中（本章稍后将介绍该伪文件的位置）。Python 程序从伪文件中读取跟踪消息并将其显示。

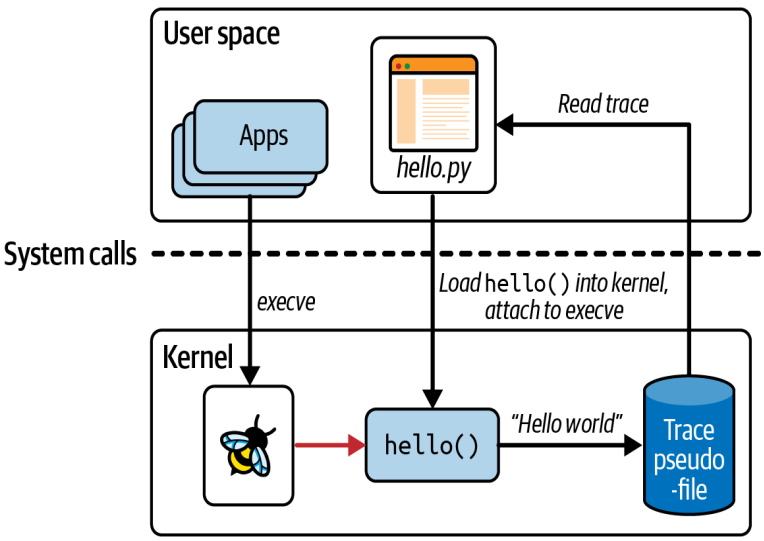

图 2.2: ”Hello World” 的运行示例

2.2 运行 “Hello World”

运行这个程序时，根据你所使用的（虚拟）机器上正在发生的情况，你会立即看到跟踪信息，因为其他进程正在使用 execve 系统调用执行程序。如果没看到任何内容，请打开第二个终端并随意执行些命令，你会看到 “Hello World” 生成的跟踪信息：

```
\$ hello.py
b'  bash-5412   [001] .... 90432.904952: 0: bpf_trace_printk:
    Hello World'
```

由于 eBPF 非常强大，使用它需要特殊权限。特殊权限会自动分配给特权 (root) 用户，因此最简单的方式就是以特权用户身份运行 eBPF 程序（用 sudo 命令）。为清晰起见，本书示例中，我不会包含 sudo 命令。只要你看到 “Operation not permitted” 错误，首先要检查的是你是否是特权用户。

CAP_BPF 权限在内核版本 5.8 中引入，并且它提供了足够的权限来执行某些 eBPF 操作，例如创建某些类型的 maps。然而，你可能还需要额外的特权：

1. 加载跟踪程序需要 CAP_PERFMON 和 CAP_BPF 权限。
2. 加载网络程序需要 CAP_NET_ADMIN 和 CAP_BPF 权限。

在 Milan Landaverde 的博客文章《Introduction to CAP_BPF》中有更多详细信息。

一旦 hello eBPF 程序加载并附加到一个内核事件上，它就会被现有程序生成的事件触发。也就是说：

- eBPF 程序可动态改变系统的行为，无需重启机器或重启现有进程。只要 eBPF 代码被附加到一个事件上，它会立即生效。
- 无需对其他程序进行更改，它们的活动都可以被 eBPF 看到。无论在机器上的哪个终端上运行可执行文件，只要是用 execve() 系统调用，它就会被触发输出跟踪信息。同样，如果你有一个运行可执行文件的脚本，那也会触发 hello eBPF 程序。你无需更改终端的 shell、脚本或正在运行的可执行文件的任何内容。

trace 输出显示触发 hello eBPF 程序运行的事件的一些上下文信息。在本节开头显示的示例输出中，执行 execve 系统调用的进程 ID 5412，并且在运行 bash 命令。对于跟踪消息，这些上下文信息是作为内核跟踪基础设施的一部分添加的（这与 eBPF 无关）。但是，正如你稍后将在本章中了解的那样，还可以在 eBPF 程序中检索此类上下文信息。

您可能想知道 Python 代码是如何知道从哪里读取跟踪输出的。答案并不复杂——内核中的 bpf_trace_printk() 辅助函数始终将输出发送到预定义的伪管道文件：/sys/kernel/debug/tracing/trace_pipe。您可以使用 cat 命令查看其内容 (需要 root 权限)。

对于简单的 “Hello World” 示例或调试，单个跟踪管道位置就足够了。输出的格式单调，仅支持字符串输出，因此对于传递结构化信息来说并非有用。最重要的是，在（虚拟）机器上只有这么一个位置。如果同时运行多个 eBPF 程序，它们都会将输出写入同一个跟踪管道，这对用户来说非常混乱。

有一种更好的方法可以从 eBPF 程序中获取清晰的信息：eBPF Map。

2.3 BPF Map

Map(映射) 是一种数据结构，可以从 eBPF 程序和用户空间访问。映射是将 eBPF 与 BPF 区分开来的重要特点之一。(你可能认为它们通常被称为 “eBPF maps”，但经常看到的是 “BPF maps”。通常情况下，这两个术语可以互换使用。)

映射可用于在多个 eBPF 程序之间共享数据，或在用户空间应用程序和运行在内核中的 eBPF 代码之间进行通信。常见用途包括:

- 用户空间将配置信息写入映射，以供 eBPF 程序检索
- eBPF 程序存储状态，以便稍后由另一个 eBPF 程序（或同一程序的未来运行）检索
- eBPF 程序将结果或指标写入映射，供将呈现结果的用户空间应用程序检索

Linux 的 uapi/linux/bpf.h文件中定义了各种类型的 BPF 映射，并在内核文档中描述了关于它们的信息。总体而言，映射都是键值存储，并且在本章中你将看到散列表、性能和环形缓冲区以及 eBPF 程序数组的映射示例。

某些映射类型被定义为数组，其键类型始终为 4 字节的索引；其他映射则是散列表，其键可以使用任意数据类型。

有一些映射类型针对特定类型的操作进行了优化，例如先入先出队列、先入后出栈、最近最少使用数据存储、最长前缀匹配和 Bloom 过滤器（一种概率数据结构，旨在快速确定元素是否存在)。

某些 eBPF 映射类型保存有关特定对象的信息。例如，sockmap 和 devmap 保存有关套接字和网络设备的信息，并由与网络相关的 eBPF 程序用于重定向流量。程序数组映射存储一组索引的 eBPF 程序，并用于实现尾调用，其中一个程序可以调用另一个程序。甚至还有一种映射类型用于支持存储有关映射的信息。

某些映射类型针对不同的 CPU 版本也有不同变体，也就是说，内核为每个 CPU 核心的版本使用不同的内存块。对于非特立 CPU 的映射，可能会让人担心并发问题，因为多个 CPU 核心可能同时访问同一个映射。内核版本自 5.1 起，添加了映射的自旋锁支持。(第 5 章中继续讨论此主题)

下一个示例（GitHub 仓库中的 chapter2/hello-map.py）展示了使用散列表映射进行一些基本操作。它还演示了 BCC 提供的方便抽象，使得使用映射非常简单。

2.3.1 哈希表 Map

与前一个例子类似，这个 eBPF 程序将附加到 execve 系统调用的入口点。它使用键值对填充一个散列表，其中键是用户 ID，值是该用户 ID 下运行的进程调用 execve 的次数。实际上，这个示例将显示每个不同用户运行程序的次数。

首先，来看一下 eBPF 程序本身的 C 代码:

```
1 BPF_HASH(counter_table);                    // 1
2
3 int hello(void *ctx) {
```

```
4       u64 uid;
5       u64 counter = 0;
6       u64 *p;
7
8       uid = bpf_get_current_uid_gid() & 0xFFFFFFFF; // 2
9       p = counter_table.lookup(&uid);               // 3
10      if (p != 0) {                                 // 4
11          counter = *p;
12      }
13
14      counter++;                                    // 5
15      counter_table.update(&uid, &counter);         // 6
16      return 0;
17  }
```

1. BPF_HASH() 是一个 BCC 宏，用于定义一个散列表映射。

2. bpf_get_current_uid_gid() 是一个辅助函数，用于获取触发此 kprobe 事件的进程所运行的用户 ID。用户 ID 存储在返回的 64 位值的最低 32 位中。（最高 32 位存储组 ID，但该部分已屏蔽掉。）

3. 在散列表中查找与用户 ID 匹配的键的条目。它返回指向散列表中相应值的指针。

4. 如果存在该用户 ID 的条目，则将计数器变量设置为散列表中的当前值（由 p 指向）。如果散列表中没有该用户 ID 的条目，指针将为 0，计数器值将保持为 0。

5. 无论当前计数器值是什么，都会将其递增一。

6. 使用计数器值更新哈希表中的用户 ID。

让我们更仔细地看下访问散列表的代码：

```
p = counter_table.lookup(&uid);
```

还有后面的代码行：

```
counter_table.update(&uid, &counter);
```

如果你认出 “这不是合法的 C 代码!”，那么你确实是对的。C 语言不支持在结构体上定义方法。这是一个很好的例子，展示了 BCC 的 C 语言版本是一个非常类似 C 语言的语言，将代码发送给编译器之前，BCC 会对其进行重写。BCC 提供了一些方便的快捷方式和宏，将它们转换为 “正确的” C 代码。

就像上一个示例一样，C 代码被定义为一个名为 program 的字符串。该程序被编译、加载到内核中，并与 execve kprobe 关联，方式与前一个 “Hello World” 示例完全相同：

```
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
```

```
b.attach_kprobe(event=syscall, fn_name="hello")
```

这次需要在 Python 代码中做一些额外的工作来读取散列表中的信息：

```
while True:                                    // 1
    sleep(2)
    s = ""
    for k,v in b["counter_table"].items(): // 2
        s += f"ID {k.value}: {v.value}\t"
    print(s)
```

1. 这部分代码无限循环，每两秒查找输出并显示一次。

2. BCC 自动创建一个 Python 对象来表示散列表，这段代码循环遍历所有值并将其打印到屏幕上。

运行这个示例时你需要在第二个终端中运行些命令。以下是我在另一个终端中运行命令时所获得的输出，右侧用注释标注了另一个终端中运行的命令：

```
Terminal 1                  Terminal 2
\$ ./hello-map.py
                            [直到我运行某些命令之前都是空行]
ID 501: 1                   ls
ID 501: 1
ID 501: 2                   ls
ID 501: 3       ID 0: 1     sudo ls
ID 501: 4       ID 0: 1     ls
ID 501: 4       ID 0: 1
ID 501: 5       ID 0: 2     sudo ls
```

该示例每两秒生成一行输出，无论是否发生了任何事件。在输出的末尾，散列表包含了两个条目：

- key=501, value=5
- key=0, value=2

在第二个终端中，用户 ID 为 501。在该用户 ID 下，运行 ls 命令会增加 execve 计数器的值。运行 sudo ls 命令，会产生两次 execve 调用：一次是以用户 ID 501 执行 sudo 命令，另一次是以根用户 ID 0 执行 ls 命令。

在这个示例中，我用了一个散列表来将数据从 eBPF 程序传递到用户空间。（在这里，我也可以使用数组类型的映射，因为键是整数；散列表允许使用任意类型作为键。）散列表在数据自然以键值对形式存在时非常方便，但用户空间的代码必须定期轮询该表。Linux 内核已经支持了用于将数据从内核发送到用户空间的perf 子系统，而 eBPF 则包括对 perf 缓冲区及其后继改进者 BPF 环形缓冲区的支持。

2.3.2 性能和环形缓冲区 Map

本节中将描述一个稍微复杂一些的“Hello World”版本，它用了 BCC 的 BPF_PERF_OUTPUT 功能，可以将数据按用户选择的结构写入 perf 环形缓冲区映射中。

如果内核版本为 5.8 或更高，现在一般推荐使用称为 “BPF 环形缓冲区” 的新结构，而不是 BPF perf 缓冲区。Andrii Nakryiko 在他的博文BPF 环形缓冲区中讨论了它们的区别。第 4 章中你会看到 BCC 的 BPF_RINGBUF_OUTPUT 示例。

环形缓冲区绝不是 eBPF 特有的，但我还是要解释一下，以防有读者之前没有遇到过。你可以将环形缓冲区视为逻辑上组织成环形的一块内存，具有单独的 “写入” 和 “读取” 指针。某些任意长度的数据被写入到写指针所在的位置，数据的长度信息则包含在该数据的头部。写指针移动到数据的末尾之后的位置以准备进行下一次写入。

同样地，对于读操作，数据从读指针所在的位置读取，使用头部确定要读的数据量。读指针沿着与写指针相同的方向移动，指向下一个可用的数据片段。图 2-3 展示了一个具有三个不同长度的项可供读取的环形缓冲区的示例。

如果读指针追上写指针，意味着没有数据可读了。如果写操作导致写指针越过读指针，数据不会被写入，而是会增加一个丢弃计数器。读操作包括丢弃计数器，以指示自上次成功读取以来是否丢失数据。

如果读和写操作以完全相同的速率进行且始终两种操作处理相同数量的数据，则至少从理论上讲，你使用刚好能保存该数据大小的环形缓冲区就够了。大多数应用程序中，读和写之间的时间会有一定的变化，因此缓冲区大小需要调整以适应这种情况。

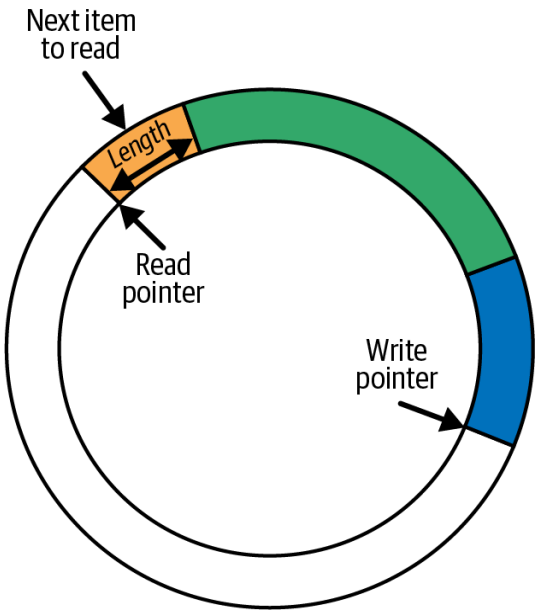

图 2.3: 一个环形缓冲区

您可以在 Learning eBPF GitHub仓库的 chapter2/hello-buffer.py 文件中找到此示例的代码。和本章最开始讲的第一个 “Hello World” 类似，这个版本在每次调用 execve() 时

都将字符串 “Hello World” 打印到屏幕上，它还会查找每个 execve() 调用相关的进程 ID 和命令名称，以产生类似于第一个示例的输出。这为我们提供了展示更多 BPF 辅助函数的机会。

下面是要加载到内核中的 eBPF 程序：

```
BPF_PERF_OUTPUT(output);

struct data_t {
    int pid;
    int uid;
    char command[16];
    char message[12];
};

int hello(void *ctx) {
    struct data_t data = {};
    char message[12] = "Hello World";

    data.pid = bpf_get_current_pid_tgid() >> 32;
    data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

    bpf_get_current_comm(&data.command, sizeof(data.command));
    bpf_probe_read_kernel(&data.message,
        sizeof(data.message), message);

    output.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
```

1. BCC 定义了宏 BPF_PERF_OUTPUT，用于创建映射（map），该映射用于从内核传递消息到用户空间。这里将映射称为 output。

2. 每次运行 hello() 函数时，代码写入数据结构的内容。以下是该结构的定义，包含进程 ID、当前运行命令的名称和一个文本消息字段。

3. data 是一个局部变量，用于保存要提交的数据结构，而 message 保存着”Hello World” 字符串。

4. bpf_get_current_pid_tgid() 是一个辅助函数，用于获取触发此 eBPF 程序运行的进程 ID。它返回一个 64 位值，其中进程 ID 位于高 32 位。

5. bpf_get_current_uid_gid() 是前面示例中的辅助函数，用于获取用户 ID。

6. 类似地，bpf_get_current_comm() 也是一个辅助函数，用于获取执行 execve 系统

调用的进程中正在运行的可执行文件（或” 命令”）的名称。这是一个字符串，而不是像进程和用户 ID 那样的数值，在 C 语言中不能简单地使用 “=” 赋值字符串。你必须将字符串写入的地址 &data.command 作为辅助函数的参数进行传递。

7. 对此示例，每次消息都是”Hello World”。bpf_probe_read_kernel() 将其复制到数据结构的正确位置。

8. 此时，数据结构已填充了进程 ID、命令名称和消息。output.perf_submit() 函数调用则将数据放入映射中。

与第一个”Hello World” 示例一样，这个 C 程序在 Python 代码中被赋值给名为 program 的字符串。以下是 Python 代码的其余部分：

```
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")

def print_event(cpu, data, size):
    data = b["output"].event(data)
    print(f"{data.pid} {data.uid} {data.command.decode()} " +
        f"{data.message.decode()}")

b["output"].open_perf_buffer(print_event)
while True:
    b.perf_buffer_poll()
```

1. 编译 C 代码、加载到内核并将其附加到系统调用事件的代码行与您之前看到的”Hello World” 版本相同。

2. print_event 是一个回调函数，用于将一行数据输出到屏幕上。BCC 完成了一些繁重的工作，因此我可以简单地将映射表示为 b[”output”]，并使用 b[”output”].event() 从中获取数据。

3. b[”output”].open_perf_buffer() 打开了性能环形缓冲区。该函数以 print_event 作为参数，定义了每当有数据可从缓冲区中读取时要使用的回调函数。

4. 程序无限循环轮询性能环形缓冲区。只要里面有数据就会调用 print_event 函数。

运行此代码将得到与第一个”Hello World” 程序非常类似的输出：

```
\$ sudo ./hello-buffer.py
11654 node Hello World
11655 sh Hello World
...
```

与之前一样，你可能要打开一个新终端并运行命令以触发输出。这个例子与第一个”Hello World” 例子间的最大区别在于，现在数据通过一个名为 output 的环形缓冲区映射传递，该

映射是由程序自己创建，如图 2-4 所示。

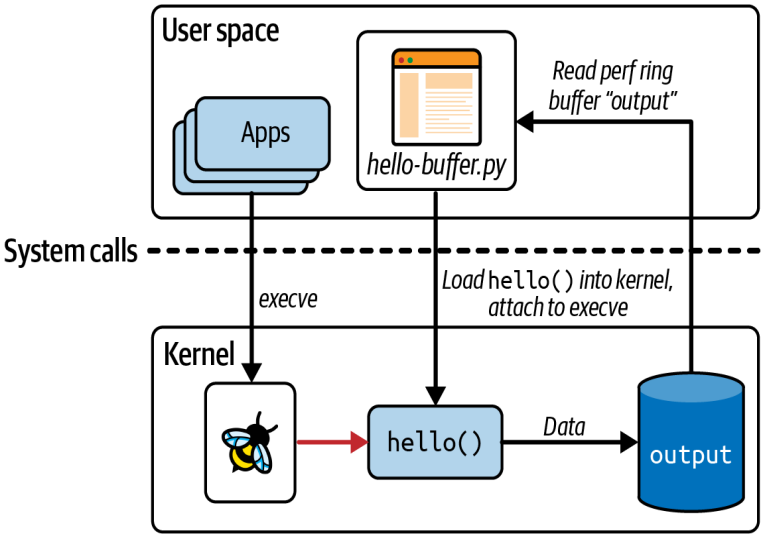

图 2.4: 使用 perf 环形缓冲区将数据从内核传递到用户空间

可以使用 cat /sys/ kernel/debug/tracing/trace_pipe 来验证信息是否发送到跟踪管道。

除了演示环形缓冲区映射的使用之外，此例子还展示了一些 eBPF 辅助函数，用于获取触发 eBPF 程序运行事件的上下文信息。在这里，你看到了获取用户 ID、进程 ID 和当前命令名称的辅助函数。正如第 7 章中会提到的那样，可用的上下文信息和可用于检索它的有效辅助函数集取决于程序的类型和触发它的事件。

上下文信息可用是 eBPF 代码有价值的原因之一。每当事件发生时，eBPF 程序可以报告事件发生了，还可以提供有关触发事件的详细信息。它还具有高性能，因为所有这些信息在内核中收集，无需切换到用户空间，进行任何上下文同步。

本书中还有更多的例子，其中 eBPF 辅助函数用于收集其他上下文信息，以及使用 eBPF 程序更改上下文信息甚至完全阻止事件发生。

2.3.3 函数调用

eBPF 程序可以调用内核提供的辅助函数，但是如果你要将代码拆分为函数呢？一般来说，在软件开发中，将常见代码提取到一个函数中，就可以在其他地方调用该函数，而不是反复写相同的代码，这被认为是一种好的规范。但在早期，eBPF 程序不允许调用除辅助函数外的其他函数。为了解决这个问题，程序员向编译器指示将函数 “始终内联”：

```
static __always_inline void my_function(void *ctx, int val)
```

通常情况下，源代码中的函数会导致编译器生成跳转指令，导致执行跳转到组成被调用函数的一组指令（然后在该函数完成时再次跳回）。如图 2-5 的左侧。右侧展示了内联函

数的情况：没有跳转指令；相反，在调用函数内直接生出该函数的指令副本。

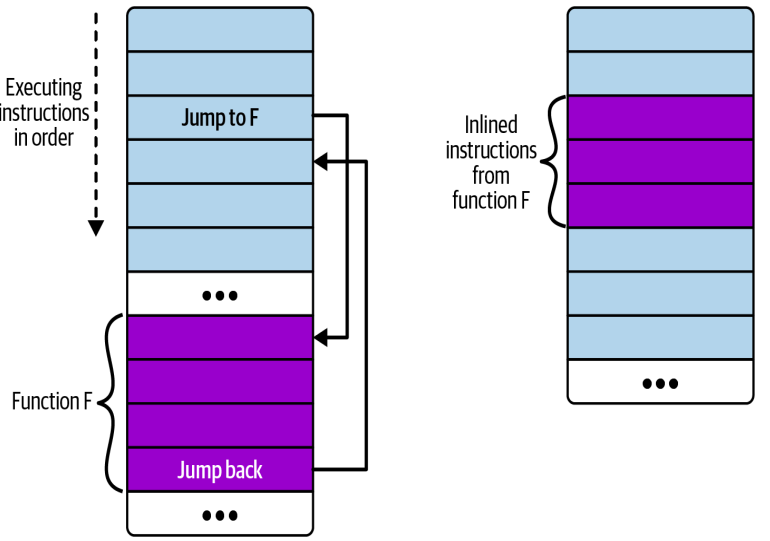

图 2.5: 非内联和内联函数指令的内存布局

如果该函数在多处使用，那么在编译后的可执行文件中会有该函数的多个副本。(有时编译器可能会选择内联函数进行优化，这也是为什么可能无法将 kprobe 附加到某些内核函数的原因之一。第 7 章中将再讨论这个问题。)

从 Linux 内核 4.16 和 LLVM 6.0 开始，解除了需要内联函数的限制，以便 eBPF 程序员可以更自然地编写函数调用。然而，这个名为 “BPF to BPF function calls” 或 “BPF subprograms” 的特性目前还没有被 BCC 框架支持，所以下一章中再来讨论。(当然，如果内联的话，可以继续在 BCC 中使用函数。)

在 eBPF 中，还有另一种将复杂功能分解为小任务的机制：尾调用（tail calls)。

2.3.4 尾调用

正如ebpf.io所描述的，“尾调用可以调用和执行另一个 eBPF 程序，并替换执行上下文，类似于常规进程中的 execve() 系统调用的操作方式。” 换句话说，在尾调用完成后，执行不会返回给调用者。

尾调用并不是 eBPF 编程中独有的。尾调用的目的是避免在函数递归调用时一遍遍地向堆栈添加帧，这最终可能导致堆栈溢出错误。如果可以组织代码在调用递归函数后执行最后一个操作，那么与调用函数相关联的堆栈帧实际上并没有增长。尾调用允许在不增加堆栈的情况下调用一系列函数。在 eBPF 中，这特别有用，因为堆栈限制为 512 字节。

尾调用使用 bpf_tail_call() 辅助函数执行，其签名如下所示

```
long bpf_tail_call(void *ctx, struct bpf_map *prog_array_map,
    u32 index)
```

三个参数具有含义如下：

- ctx 允许将上下文从调用的 eBPF 程序传递给被调用的程序。
- prog_array_map 是一个类型为 BPF_MAP_TYPE_PROG_ARRAY 的 eBPF 映射，其中保存了用于标识 eBPF 程序的一组文件描述符。
- index 指示应调用该组 eBPF 程序中的哪一个。

这个辅助函数有点不一样，因为执行成功了它不会返回。当前运行的 eBPF 程序在堆栈上被调用的程序所取代。辅助函数可能会失败，例如，如果映射中不存在指定的程序，则原调用程序会继续执行。

用户空间代码必须将所有的 eBPF 程序加载到内核中，并设置程序数组映射(prog_array_map)。

来看一个简单的使用 BCC 编写的 Python 示例；可在 GitHub仓库的 chapter2/hello-tail.py 中找到代码。eBPF 程序附加到一个追踪点，该追踪点是所有系统调用的公共入口点。该程序使用尾调用来为特定的系统调用操作码跟踪特定的消息。如果给定操作码没有尾调用，则该程序将输出一个通用消息。

如果你正在使用 BCC 框架，可以使用稍微简化的形式进行尾调用：

```
prog_array_map.call(ctx, index)
```

将代码传递给编译器之前，BCC 会将上述行重写为：

```
bpf_tail_call(ctx, prog_array_map, index)
```

以下是 eBPF 程序及其尾调用的代码：

```
1  BPF_PROG_ARRAY(syscall, 300);                              // 1
2
3  int hello(struct bpf_raw_tracepoint_args *ctx) {           // 2
4      int opcode = ctx->args[1];                             // 3
5      syscall.call(ctx, opcode);                             // 4
6      bpf_trace_printk("Another syscall: %d", opcode); // 5
7      return 0;
8  }
9
10 int hello_execve(void *ctx) {                              // 6
11     bpf_trace_printk("Executing a program");
12     return 0;
13 }
14
15 int hello_timer(struct bpf_raw_tracepoint_args *ctx) { // 7
16     if (ctx->args[1] == 222) {
17         bpf_trace_printk("Creating a timer");
18     } else if (ctx->args[1] == 226) {
```

```
19          bpf_trace_printk("Deleting a timer");
20      } else {
21          bpf_trace_printk("Some other timer operation");
22      }
23      return 0;
24  }
25
26  int ignore_opcode(void *ctx) {                          // 8
27      return 0;
28  }
```

1. BCC 提供了 BPF_PROG_ARRAY 宏,便于定义 BPF_MAP_TYPE_PROG_ARRAY 类型的映射。我将这个映射命名为 syscall，允许最多 300 个条目，对于本示例来说足够了。

2. 在用户空间代码中,将这个 eBPF 程序附加到 sys_enter 原始跟踪点,任何系统调用都会触发该跟踪点。附加到原始跟踪点的 eBPF 程序接收的上下文采用 bpf_raw_tracepoint_args 结构的形式。

3. 对于 sys_enter 而言，原始跟踪点参数包括用于标识正在进行的系统调用的操作码。

4. 在这里，对与操作码匹配的程序数组中的条目进行尾调用。将源码传递给编译器之前，BCC 将重写这行代码，以调用 bpf_tail_call() 辅助函数。

5. 如尾调用成功，这行代码不会执行，不会输出操作码值。我用这行来提供没有映射中程序条目的操作码的跟踪。

6. hello_exec() 是一个要加载到系统调用程序数组映射中的程序，在操作码指示为 execve() 系统调用时作为尾调用执行。它将生成一行跟踪信息，告诉用户正在执行新程序。

7. hello_timer() 是另一个要加载到系统调用程序数组中的程序。在这种情况下，它会被多个程序数组条目引用。

8. ignore_opcode() 是一个空的尾调用程序，不执行任何操作。我会在不希望生成任何跟踪信息的系统调用中使用。

现在来看看加载和管理这组 eBPF 程序的用户空间代码:

```
b = BPF(text=program)
b.attach_raw_tracepoint(tp="sys_enter", fn_name="hello") // 1

ignore_fn = b.load_func("ignore_opcode", BPF.RAW_TRACEPOINT) // 2
exec_fn = b.load_func("hello_exec", BPF.RAW_TRACEPOINT)
timer_fn = b.load_func("hello_timer", BPF.RAW_TRACEPOINT)

prog_array = b.get_table("syscall")                      // 3
prog_array[ct.c_int(59)] = ct.c_int(exec_fn.fd)
prog_array[ct.c_int(222)] = ct.c_int(timer_fn.fd)
```

```
prog_array[ct.c_int(223)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(224)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(225)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(226)] = ct.c_int(timer_fn.fd)

# Ignore some syscalls that come up a lot                    // 4
prog_array[ct.c_int(21)] = ct.c_int(ignore_fn.fd)
prog_array[ct.c_int(22)] = ct.c_int(ignore_fn.fd)
prog_array[ct.c_int(25)] = ct.c_int(ignore_fn.fd)
...
b.trace_print()                                              // 5
```

1. 用户空间代码将主要的 eBPF 程序附加到 sys_enter 跟踪点，而不是像之前一样附加到 kprobe 上。

2. 这些对 b.load_func() 的调用会返回每个尾调用程序的文件描述符。请注意，尾调用程序需要与其父程序具有相同的程序类型，这里是 BPF.RAW_TRACEPOINT。此外，需要指出的是，每个尾调用程序本身都是一个独立的 eBPF 程序。

3. 用户空间代码在 syscall 映射表中创建条目。对每个可能的操作码，映射表并不需要完全填充；如果某个特定的操作码没有条目，那么意味着不会执行任何尾调用。此外，可以有多个条目指向同一个 eBPF 程序。这种情况下，我希望对一组与计时器相关的系统调用执行 hello_timer() 尾调用。

4. 一些频繁运行的系统调用会使跟踪信息输出变得非常杂乱，以至于无法阅读。我使用了 ignore_opcode() 尾调用来处理这些系统调用。

5. 将跟踪信息输出打印到屏幕上，直到用户终止程序为止。

运行这个程序会生成每个在虚拟机上运行的系统调用的跟踪输出，除非操作码是将其链接到 ignore_opcode() 尾调用的条目。以下是在另一个终端上运行 ls 命令时的一些输出（为可读性省略了些细节）：

```
./hello-tail.py
b'  hello-tail.py-2767  ... Another syscall: 62'
b'  hello-tail.py-2767  ... Another syscall: 62'
...
b'  bash-2626           ... Executing a program'
b'  bash-2626           ... Another syscall: 220'
...
b'  <...>-2774          ... Creating a timer'
b'  <...>-2774          ... Another syscall: 48'
b'  <...>-2774          ... Deleting a timer'
...
```

```
b'  ls-2774             ... Another syscall: 61'
b'  ls-2774             ... Another syscall: 61'
...
```

具体执行的系统调用并不重要，但可以看到不同的尾调用被调用了，并生成了跟踪消息。在尾调用程序映射中没有条目的操作码，会输出默认消息”Another syscall”。

请查看 Paul Chaignon 关于不同内核版本中 BPF尾调用成本的文章。

自内核版本 4.2 起，eBPF 支持尾调用，但长期以来，它们与 BPF 间函数调用不兼容。这个限制在内核 5.10 中解除了。尾调用最多可以链接 33 个调用，加上每个 eBPF 程序的 100 万条指令的限制，意味着现在的 eBPF 程序员在内核中编写非常复杂的代码时有很大的灵活性。

2.4 总结

通过一些具体的 eBPF 程序例子，希望本章能帮助您巩固对内核中 eBPF 代码通过事件触发的心智模型。你还看到了用 BPF 映射将数据从内核传递到用户空间的例子。

使用 BCC 框架隐藏了构建程序、加载到内核并附加到事件的许多细节。在下一章中，我将向你展示编写 “Hello World” 的不同方法，并更深入探讨这些隐藏细节。

2.5 练习

以下是一些可选的练习，如果你想进一步探索”Hello World” 的示例，请尝试（或考虑）以下内容：

1. 修改 hello-buffer.py 的 eBPF 程序，根据进程 ID 的奇偶性输出不同的跟踪消息。

2. 修改 hello-map.py，使其能够触发多个系统调用。例如，openat() 通常用于打开文件，write() 用于向文件写入数据。你可以将 hello eBPF 程序附加到多个系统调用的 kprobe 上。然后尝试为不同的系统调用编写修改版本的 hello eBPF 程序，演示从多个不同的程序访问同一个 map 的能力。

3. hello-tail.py 是一个附加到 sys_enter raw tracepoint 的程序，该 tracepoint 在任何系统调用发生时触发。将 hello-map.py 更改为通过附加到相同的 sys_enter raw tracepoint 显示每个用户 ID 执行的系统调用总数。下面是我在进行修改后得到的输出：

```
$ ./hello-map.py
ID 104: 6   ID 0: 225
ID 104: 6   ID 101: 34   ID 100: 45   ID 0: 332   ID 501: 19
ID 104: 6   ID 101: 34   ID 100: 45   ID 0: 368   ID 501: 38
ID 104: 6   ID 101: 34   ID 100: 45   ID 0: 533   ID 501: 57
```

4. 使用 BCC 提供的 RAW_TRACEPOINT_PROBE 宏简化对原 tracepoint 的附加。要在 hello-tail.py 中使用它，步骤如下：

- 将 hello() 函数的定义替换为 RAW_TRACEPOINT_PROBE(sys_enter)。
- 从 Python 代码中删除显式的 b.attach_raw_tracepoint() 调用。

你会看到 BCC 自动创建附加项，并且程序的运行结果完全相同，这是 BCC 提供的许多便利宏的示例。

5. 进一步修改 hello_map.py，使哈希表中的键标识特定的系统调用（而不是特定的用户），输出结果显示系统中该系统调用发生的次数。

第三章　eBPF 程序剖析

在前一章中，我们看到了使用 BCC 框架写的 eBPF “Hello World” 程序。本章中则有一个完全用 C 语言写的 “Hello World” 程序，以便读者可以看到 BCC 幕后处理的细节。

本章还将向读者展示 eBPF 程序从源代码到最终执行的各个阶段，如图 3-1 所示。

图 3.1: C/Rust 源代码编译成 eBPF 字节码，字节码被即时编译（JIT）或解释成机器码

eBPF 程序是一组 eBPF 字节码指令，可以直接用这种字节码编写 eBPF 代码，就像用汇编语言编程一样。但高级编程语言是更容易处理的，至少在写代码时时，我可以说绝大多数 eBPF 代码都是用 C 写的，然后再编译成 eBPF 字节码。

从概念上讲，这些字节码是在内核中的 eBPF 虚拟机中运行。

3.1　eBPF 虚拟机

eBPF 虚拟机，就像其他虚拟机一样，是一种软件实现。它接受以 eBPF 字节码指令形式表示的程序，并且这些指令必须转换为 CPU 上运行的机器指令。

在早期的 eBPF 实现中，字节码指令在内核中进行解释执行，也就是说，每次运行 eBPF 程序时，内核会检查指令并将其转换为机器码，然后执行。出于性能和避免 eBPF 解释器中的 Spectre 漏洞的考虑，解释执行已在很大程度上被即时编译（JIT）所取代。编译意味着将字节码转换为机器指令只会发生一次，即在程序加载到内核的时候。

eBPF 字节码由一组指令组成，这些指令作用于（虚拟的）eBPF 寄存器。eBPF 指令集和寄存器模型设计目的与常见的 CPU 体系结构类似，使得从字节码到机器码的编译或解释步骤很简单。

3.1.1　eBPF 寄存器

eBPF 虚拟机使用 10 个通用寄存器，编号从 0 到 9。此外，寄存器 10 被用作堆栈帧指针（只能读，不能写）。在执行 BPF 程序时，各种值存储在这些寄存器中以跟踪状态。

重要的是要理解，eBPF 虚拟机中的这些寄存器是通过软件定义实现的。你可以在 Linux 内核源代码的include/uapi/linux/bpf.h头文件中查看，BPF_REG_0 到 BPF_REG_10 都是枚举。

eBPF 程序的上下文参数在执行开始前会加载到寄存器 1 中，函数的返回值则存储于寄存器 0 中。

eBPF 代码调用函数之前会将函数的参数存储在寄存器 1 到 5 中（如参数少于五个，则不会使用所有的寄存器）。

3.1.2 eBPF 指令

同样，linux/bpf.h头文件定义了一个称为 bpf_insn 的结构体，用于定义一个 BPF 指令：

```
struct bpf_insn {
    __u8 code;          /* opcode */            // 1
    __u8 dst_reg:4;     /* dest register */     // 2
    __u8 src_reg:4;     /* source register */
    __s16 off;          /* signed offset */     // 3
    __s32 imm;          /* signed immediate constant */
};
```

1. 每个指令都有一个操作码，定义了指令要执行的操作，例如将一个值加到寄存器中，或跳转到程序中的另一个指令。Iovisor 项目的“非官方 eBPF 规范”中列出了有效指令的列表。

2. 不同的操作可能涉及两个寄存器。

3. 根据操作的不同，可能会有一个偏移值和/或一个立即整数值

这个 bpf_insn 结构体的长度为 8 个字节，有时指令也可能会超过 8 个字节。如要将一个寄存器设置为 64 位值，无法将该值的 64 位与操作码和寄存器信息一起存到结构体中。这种情况下，指令需采宽指令编码，总长度为 16 字节。

加载到内核中后，eBPF 程序的字节码由一系列这样的 bpf_insn 结构体表示。验证器对这些信息执行多项检查，以确保代码可以安全运行。在第 6 章中将详细介绍验证过程。

大多数不同的操作码都可以归类为以下几类：

- 将一个值加载到寄存器中（可以是立即值，也可以是从内存或其他寄存器中读取的值）
- 将寄存器中的值存储到内存中
- 执行算术运算，例如将一个值加到寄存器的值上
- 如果满足特定条件，则跳转到另一个指令

对于 eBPF 架构的概述，我推荐阅读 Cilium 项目文档中包含的BPF 和 XDP 参考指南。如果想了解更多细节，内核文档对 eBPF 指令和编码进行了相当清晰的描述。

下面用另一个简单的 eBPF 程序示例来学习它从 C 代码到 eBPF 字节码再到机器码指令的过程。

如果想自己构建和运行这段代码，可以在github.com/lizrice/learning-ebpf找到该代码以及设置运行环境的说明，本章的代码位于 chapter3 目录中。本章的示例使用了一个名为 libbpf 的库来编写 C 代码，第 5 章中会更多地谈及这个库。

3.2 针对网络接口的 eBPF “Hello World”

上一章中的示例中，通过系统调用 kprobe 触发了”Hello World” 的跟踪输出；而这一次，我将展示一个网络数据包到达时触发的 eBPF 程序，它会写入一行跟踪信息。

数据包处理是 eBPF 的一项基本功能。现在了解每个数据包到达网络接口时都会触发的 eBPF 程序的基本思想对后续学习可能会有帮助。该程序可以检查甚至修改数据包的内容，并对内核在处理数据包时做出决策（或判段）。eBPF 的判断可以告诉内核是继续按常规方式处理数据包、丢弃它或将其重定向到其他位置。

在我这里展示的例子中，程序不对数据包做任何处理，它只在收到数据包时将”Hello World” 和一个计数器写入跟踪管道。

示例程序位于 chapter3/hello.bpf.c。通常约定将 eBPF 程序放在以 bpf.c 结尾的文件名中，以区分它们与可能存在于同一目录中的用户空间 C 代码。以下是完整的程序：

```
#include <linux/bpf.h>                  // 1
#include <bpf/bpf_helpers.h>

int counter = 0;                        // 2

SEC("xdp")                              // 3
int hello(void *ctx) {                  // 4
    bpf_printk("Hello World %d", counter);
    counter++;
    return XDP_PASS;
}

char LICENSE[] SEC("license") = "Dual BSD/GPL"; // 5
```

1. 些头文件。以防您不熟悉 C 编程，每个程序都必须包含程序使用到的任何结构或函数的定义 (头文件)。从这些头文件的名称就可以猜出它们与 BPF 相关。

2. 展示 eBPF 程序如何使用全局变量。程序每次运行时，计数器都会递增。

3. 宏 SEC() 定义了一个名为 xdp 的段 (section)，可以在编译出的目标文件中看到它。第 5 章中将详细介绍段的用法，现在你可以简单地将其视为定义了一个 eXpress Data Path（XDP）类型的 eBPF 程序。

4. eBPF 程序。在 eBPF 中，程序名就是函数名，因此此程序称为 hello。它使用了一个辅助函数 bpf_printk 来写入一串文本，递增全局变量 counter，然后返回值 XDP_PASS，这是告诉内核应该将该数据包作为正常数据包进行处理。

5. 最后还有另一个 SEC() 宏，用于定义许可证字符串，这是 eBPF 程序的一项要求。内核中的一些 BPF 辅助函数被定义为” 仅 GPL”。如果你想使用其中的任何函数，那么你的整个 BPF 代码都必须声明为具有 GPL 兼容的许可证。验证器（第 6 章中讨论）将会检查声明的许可证是否与程序使用的函数兼容。某些 eBPF 程序类型，包括使用 BPF LSM（第 9 章中讨论）的程序，也需要与 GPL 兼容。

也许你想知道为什么上一章使用了 bpf_trace_printk()，而这里使用的是 bpf_printk()。一句话来解释就是：BCC 框架里这个功能称为 bpf_trace_printk()，而 libbpf 库里称为 bpf_printk()，这两个函数其实都是对内核函数 bpf_trace_printk() 的封装。Andrii Nakryiko 在他的博客上写了一篇很好的文章来解释这个问题。

这个程序是附加到网络接口 XDP 事件的钩子程序。可以将 XDP 事件视为在（物理或虚拟）网络接口上接收到数据包瞬间触发的事件。

有些网卡支持卸载 XDP 程序，这样就可以在网卡上执行这些程序。这意味着每个到达的数据包都可以在网卡上进行处理，而不必通过 CPU 处理。XDP 程序可以检查甚至修改每个数据包，因此可以以高性能方式进行 DDoS 保护、防火墙或负载均衡等操作。

看过了 C 语言写的 eBPF 程序，下一步来学习下如何将其编译成内核可以执行的目标文件。

3.3 编译 eBPF 目标文件

eBPF 代码需要编译成 eBPF 虚拟机可以理解的机器指令，即 eBPF 字节码。LLVM 项目的 Clang 编译器可以完成这一任务，只需要指定 -target bpf 参数。以下是一个 Makefile，用于编译 eBPF 程序：

```
hello.bpf.o: %.o: %.c
    clang \
        -target bpf \
        -I/usr/include/$(shell uname -m)-linux-gnu \
        -g \
        -O2 -c $< -o $@
```

这将把 hello.bpf.c 源代码编译成一个名为 hello.bpf.o 的目标文件。这里 -g 标志是可选的，它用于生成调试信息，以便在检查目标文件时可以查看源代码与字节码的对应关系。让我们来检查下这个目标文件，以更好地理解其中包含的 eBPF 代码。

3.4 检查 eBPF 目标文件

常用的 file 命令行工具可以用来确定文件的内容：

```
$ file hello.bpf.o
hello.bpf.o: ELF 64-bit LSB relocatable, eBPF, version 1 (
   SYSV), with debug_info, not stripped
```

这里显示它是一个 ELF（可执行和可链接格式）文件，包含 eBPF 代码，针对 LSB（最低有效位）架构的 64 位平台。如果在编译步骤中使用了 -g 标志，还会包含调试信息。

你可以使用 llvm-objdump 进一步检查该对象以查看其中的 eBPF 指令：

```
$ llvm-objdump -S hello.bpf.o
```

即使不熟悉反汇编，该命令的输出也不难理解。

```
hello.bpf.o:    file format elf64-bpf   // 1

Disassembly of section xdp:             // 2

0000000000000000 <hello>:               // 3
; bpf_printk("Hello World %d", counter"); // 4
   0:   18 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r6 = 0 ll
   2:   61 63 00 00 00 00 00 00 r3 = *(u32 *)(r6 + 0)
   3:   18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 ll
   5:   b7 02 00 00 0f 00 00 00 r2 = 15
   6:   85 00 00 00 06 00 00 00 call 6
; counter++;                            // 5
   7:   61 61 00 00 00 00 00 00 r1 = *(u32 *)(r6 + 0)
   8:   07 01 00 00 01 00 00 00 r1 += 1
   9:   63 16 00 00 00 00 00 00 *(u32 *)(r6 + 0) = r1
; return XDP_PASS;                      // 6
   10:  b7 00 00 00 02 00 00 00 r0 = 2
   11:  95 00 00 00 00 00 00 00 exit
```

1. 第一行确认了 hello.bpf.o 是一个 64 位的 ELF 文件，包含 eBPF 代码（有的工具显示 BPF，而其他工具显示 eBPF，这个没有特定原因，这些术语现在几乎可以互换使用）。

2. 接下来是标记为 xdp 的代码段反汇编，与 C 代码中的 SEC() 定义对应。

3. 这里是一个名为 hello 的函数。

4. 五行 eBPF 字节码指令，与源码中的 bpf_printk(”Hello World

5. 三行 eBPF 字节码指令，用于递增 counter 变量。

6. 另外两行字节码对应源码中的 return XDP_PASS。

没必要理解每行字节码与源代码的精确对应关系，除非你特别想这样干。编译器会负责生成字节码，你无需考虑这些对应关系！现在，来详细地检查一下输出，以便你对这个反汇编是如何与 eBPF 指令和寄存器间的联系有一定了解。

在每行字节码的左侧，都可以看到该指令相对于内存中 hello 的偏移量。正如前面所述，eBPF 指令通常为 8 字节长，而在 64 位平台上，每个内存位置可以容纳 8 个字节，因此偏移量通常逐指令递增。然而，此程序中的第一条指令恰好是需要 16 个字节的宽指令编码，以将寄存器 6 设置为 64 位值 0，所以偏移量是 2。之后是另一条 16 字节的指令，将寄存器 1 设置为 64 位值 0。剩下的指令每行都是 8 字节，因此每行偏移量递增一。

每行的第一个字节是告诉内核要执行操作的操作码，在指令行的右侧是该指令的解释。截至撰写本文时，Iovisor 项目提供了关于 eBPF 操作码的最完整文档，官方 Linux内核文档也正迎头赶上，而 eBPF 基金会正在编写与特定操作系统无关的标准文档。

让我们看看偏移量为 5 的指令，它看起来像这样：

```
5:  b7 02 00 00 0f 00 00 00 r2 = 15
```

这个指令的操作码是 0xb7，根据文档，对应的伪代码是 dst = imm，可以理解为 “将目标设置为立即值”。目标由第二个字节定义，0x02 表示 “寄存器 2”。这里的 “立即值”（或文字值）是 0x0f，转换成十进制是 15。因此，这个指令是告诉内核 “将寄存器 2 设置为值 15”。这与指令右侧的输出相对应：r2 = 15。

偏移量为 10 的指令类似：

```
10: b7 00 00 00 02 00 00 00 r0 = 2
```

这行指令的操作码也是 0xb7，这次它将寄存器 0 的值设置为 2。当 eBPF 程序运行结束时，寄存器 0 保存返回码，而 XDP_PASS 的值就是 2，这与源码中始终返回 XDP_PASS 的部分对应上了。

现在你知道 hello.bpf.o 是一个字节码形式的 eBPF 程序，那么下一步就是将其加载到内核中。

3.5 将程序加载到内核

在这个例子中，我们将使用一个叫 bpftool 的工具。你也可以以编程的方式加载程序，在本书的后面章节中将会看到相关示例。

一些 Linux 发行版提供了包含 bpftool 的软件包，或者可以从源代码编译。可以在Quentin Monnet 的博客上找到有关安装或构建该工具的更多信息，以及Cilium 网站上的附加文档和用法。

以下是使用 bpftool 将程序加载到内核中的例子。请注意，需要以 root 身份（或使用 sudo）获取 bpftool 所需的 BPF 权限。

```
$ bpftool prog load hello.bpf.o /sys/fs/bpf/hello
```

这行命令将从编译的目标文件中加载 eBPF 程序，并将其“固定”到/sys/fs/bpf/hello。命令没有输出响应就表示加载成功。你可以使用 ls 命令确认程序是否已就位：

```
$ ls /sys/fs/bpf
hello
```

eBPF 程序已成功加载，让我们使用 bpftool 工具来了解程序及其在内核中的状态。

3.6 检查已加载的程序

bpftool 工具可以列出加载到内核中的所有程序。你可能会看到输出中有几个预先存在的 eBPF 程序，但为了清晰起见，这里我只列出“Hello World”示例相关的行：

```
$ bpftool prog list
...
540: xdp name hello tag d35b94b4c0c10efb gpl
        loaded_at 2022-08-02T17:39:47+0000 uid 0
        xlated 96B jited 148B memlock 4096B map_ids 165,166
        btf_id 254
```

该程序的 ID 为 540。这个标识是加载程序时分配的一个数字。知道了 ID，就可以显示该程序的更多信息。首先，将输出以 JSON 格式显示，以便更清楚地看字段名称和值：

```
$ bpftool prog show id 540 --pretty
{
    "id": 540,
    "type": "xdp",
    "name": "hello",
    "tag": "d35b94b4c0c10efb",
    "gpl_compatible": true,
    "loaded_at": 1659461987,
    "uid": 0,
    "bytes_xlated": 96,
    "jited": true,
    "bytes_jited": 148,
    "bytes_memlock": 4096,
    "map_ids": [165,166],
    "btf_id": 254
}
```

根据字段名称，很多内容都很好理解：

- 程序的 ID 是 540。
- type 字段表明程序可以使用 XDP 事件附加到网络接口。还有其他类型的 BPF 程序可以附加到不同类型的事件上。
- 程序名称是 hello，这是源码中的函数名。
- tag 是该程序的另一个标识符，我很快会介绍。
- 该程序采用了与 GPL 兼容的许可证。
- 时间戳显示了程序加载的时间。
- 用户 ID 为 0（即 root）加载了该程序。
- 该程序包含了 96 字节已翻译的 eBPF 字节码。
- 该程序已进行 JIT 编译，编译结果是 148 字节的机器码。
- bytes_memlock 字段表明该程序保留了 4096 字节的内存，不会被分页出去。
- 该程序引用了 ID 为 165 和 166 的 BPF 映射。这可能看起来令人困惑，因为源码中没有明显的引用。在本章后面，你将了解到如何在 eBPF 程序中使用映射语义来处理全局数据。
- 在第 5 章中，你将了解 BTF 的相关知识，但现在只需知道 btf_id 表示该程序有一块 BTF 信息。只有在使用-g 标志进行编译时，该信息才包含在对象文件中。

3.6.1 BPF 程序标记

该 tag 是程序指令的 SHA（安全哈希算法）校验和，可用作程序的另一个标识符。每次加载或卸载程序时，ID 可能会变化，但标签将保持不变。bpftool 工具可以通过 ID、名称、标签或固定路径引用 BPF 程序，因此在此示例中，以下所有命令将输出相同的结果

- bpftool prog show id 540
- bpftool prog show name hello
- bpftool prog show tag d35b94b4c0c10efb
- bpftool prog show pinned /sys/fs/bpf/hello

可以存在多个具有相同名称的程序，甚至可以存在具有相同标签的多个程序实例，但 ID 和路径将始终是唯一的。

3.6.2 转译的 BPF 字节码

bytes_xlated 字段告诉我们经过“转换”后的 eBPF 代码有多少字节，这是经过验证器处理后的 eBPF 字节码（可能还经过内核修改）。

现在，用 bpftool 来看看“Hello World”代码的转换版本：

```
$ bpftool prog dump xlated name hello
int hello(struct xdp_md * ctx):
; bpf_printk("Hello World %d", counter);
   0: (18) r6 = map[id:165][0]+0
   2: (61) r3 = *(u32 *)(r6 +0)
```

```
      3: (18) r1 = map[id:166][0]+0
      5: (b7) r2 = 15
      6: (85) call bpf_trace_printk#-78032
; counter++;
      7: (61) r1 = *(u32 *)(r6 +0)
      8: (07) r1 += 1
      9: (63) *(u32 *)(r6 +0) = r1
; return XDP_PASS;
      10: (b7) r0 = 2
      11: (95) exit
```

这看起来与之前在 llvm-objdump 的输出中看到的反汇编代码非常相似。偏移地址相同，指令看起来也类似。

3.6.3 即时编译的机器码

翻译后的字节码非常底层，但还不是机器码。eBPF 使用即时编译器 (JIT) 再将 eBPF 字节码转换为在目标 CPU 上运行的机器码。bytes_jited 字段显示在经过这步转换后，程序的长度为 108 字节。

为了提升性能，eBPF 程序通常会进行 JIT 编译。其实，还有一种选择是在运行时解释 eBPF 字节码。eBPF 指令集和寄存器的设计相当接近机器指令，使得解释相对简单且快速，但 JIT 编译后的程序会更快，并且大多数架构现在都支持 JIT。

bpftool 工具可以生成汇编语言形式的 JIT 编译代码转储。如果你对汇编语言不熟悉，不用担心，这看起来可能完全无法理解！我只是用它来说明 eBPF 代码从源代码到可执行机器指令的所有转换过程。以下是命令及其输出：

```
$ bpftool prog dump jited name hello
int hello(struct xdp_md * ctx):
bpf_prog_d35b94b4c0c10efb_hello:

; bpf_printk("Hello World %d", counter);
0:  hint    #34
4:  stp     x29, x30, [sp, #-16]!
8:  mov     x29, sp
c:  stp     x19, x20, [sp, #-16]!
10: stp     x21, x22, [sp, #-16]!
14: stp     x25, x26, [sp, #-16]!
18: mov     x25, sp
1c: mov     x26, #0
20: hint    #36
```

```
   24: sub     sp, sp, #0
   28: mov     x19, #-140733193388033
   2c: movk    x19, #2190, lsl #16
   30: movk    x19, #49152
   34: mov     x10, #0
   38: ldr     w2, [x19, x10]
   3c: mov     x0, #-205419695833089
   40: movk    x0, #709, lsl #16
   44: movk    x0, #5904
   48: mov     x1, #15
   4c: mov     x10, #-6992
   50: movk    x10, #29844, lsl #16
   54: movk    x10, #56832, lsl #32
   58: blr     x10
   5c: add     x7, x0, #0
   ; counter++;
   60: mov     x10, #0
   64: ldr     w0, [x19, x10]
   68: add     x0, x0, #1
   6c: mov     x10, #0
   70: str     w0, [x19, x10]
   ; return XDP_PASS;
   74: mov     x7, #2
   78: mov     sp, sp
   7c: ldp     x25, x26, [sp], #16
   80: ldp     x21, x22, [sp], #16
   84: ldp     x19, x20, [sp], #16
   88: ldp     x29, x30, [sp], #16
   8c: add     x0, x7, #0
   90: ret
```

某些打包的 bpftool 发行版尚未包含转储 JIT 输出的支持，如果是这种情况，你会看到 “Error: No libbfd support.”。你可以按照https://github.com/libbpf/bpftool上的说明自己构建 bpftool 工具。

现在 “Hello World” 程序已经加载到内核中，但此时它还尚未与事件关联起来，因此不会有有触发它运行的事件。要触发运行，它需要附加到一个事件上。

3.7 附加到事件

eBPF 程序类型必须与其附加的事件类型匹配。上述 eBPF 程序是一个 XDP 程序，所以你可以使用 bpftool 将示例 eBPF 程序附加到网络接口的 XDP 事件上，像这样：

```
$ bpftool net attach xdp id 540 dev eth0
```

在撰写本文时，bpftool 工具不支持附加所有程序类型的功能，但它最近已扩展为自动附加 k(ret)probes、u(ret)probes 和 tracepoints。

我使用的程序的 ID 是 540，但你也可以使用名称（前提是唯一的）或标签来标识要附加的程序。在这个示例中，我将程序附加到网络接口 eth0 上。

你可以使用 bpftool 查看所有网络附加的 eBPF 程序：

```
$ bpftool net list
xdp:
eth0(2) driver id 540

tc:

flow_dissector:
```

ID 为 540 的程序已附加到 eth0 接口的 XDP 事件上。此输出还提供了一些关于网络堆栈中其他潜在事件的线索，你也可以将 eBPF 程序附加到这些事件上，例如 tc 和 flow_dissector。

你还可以使用 ip link 命令检查网络接口，输出结果类似以下内容（为清晰起见，删除了一些细节）：

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
   UNKNOWN mode DEFAULT
group default qlen 1000
    ...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc
    fq_codel state UP
mode DEFAULT group default qlen 1000
    ...
    prog/xdp id 540 tag 9d0e949f89f1a82c jited
    ...
```

在此示例中，有两个接口：回环接口 lo，用于将流量发送到本机上的进程；以及 eth0 接口，连接本机与外部世界。此输出还显示 eth0 接口上有一个 JIT 编译的 eBPF 程序，其标识为 540，标签为 9d0e949f89f1a82c，附加到其 XDP 事件上。

你还可以使用 ip link 命令将 XDP 程序附加到网络接口和卸载它。在本章末尾的练习中，我已经包含了这部分内容，并在第 7 章中有进一步的示例。

此时，hello eBPF 程序应该在每次接收到网络数据包时产生跟踪输出。你可以通过运行 cat /sys/kernel/debug/tracing/trace_pipe 来查看。这应该会输出类似如下内容的内容：

```
<idle>-0     [003] d.s.. 655370.944105: bpf_trace_printk:
   Hello World 4531
<idle>-0     [003] d.s.. 655370.944587: bpf_trace_printk:
   Hello World 4532
<idle>-0     [003] d.s.. 655370.944896: bpf_trace_printk:
   Hello World 4533
```

如果你忘记了跟踪管道的位置，可以使用命令 bpftool prog tracelog 获得输出。

与你在第 2 章看到的输出相比，这次每个事件的开头都没有与之关联的命令或进程 ID；相反，每行跟踪输出的开头可以看到 <idle>-0。在第 2 章中，每个系统调用事件发生是因为在用户空间执行命令的进程调用了系统调用 API。这些进程 ID 和命令是执行 eBPF 程序的上下文的一部分。但是在这个示例中，XDP 事件发生是由于网络数据包到达。这个数据包没有与之关联的用户空间进程——在 hello eBPF 程序被触发的时候，系统除了将其接收到内存中外，还没有对数据包做任何处理，也不知道数据包的内容或目的地。

你可以看到被跟踪输出的 counter 值每次递增 1，这是符合预期的。在源代码中，counter 是一个全局变量。让我们看看在 eBPF 中如何通过映射（map）来实现类似 counter 这样的全局变量。

3.8 全局变量

在前一章中，你学到了 eBPF 映射（map）是一种可以从 eBPF 程序或用户空间访问的数据结构。由于一个映射可以被程序不同的进程多次访问，所以它可以用于多次执行之间保存状态。此外，多个程序也可以访问同一个映射。由于这些特性，映射的语义本身就表明它可以作为全局变量。

在 2019 年添加对全局变量的支持之前，eBPF 程序员必须显式地编写映射来执行任务。

你之前看到 bpftool 显示这个示例程序使用了两个标识为 165 和 166 的映射。（如果你自己尝试的话，可能会看到不同的标识，因为这些标识是在内核中创建映射时分配的。）

bpftool 工具可以显示加载到内核中的映射。为清晰起见，我只显示与示例的 “Hello World” 程序相关的 165 和 166 两个条目：

```
$ bpftool map list
165: array name hello.bss   flags 0x400
    key 4B value 4B max_entries 1 memlock 4096B
    btf_id 254
```

```
166: array name hello.rodata flags 0x80
    key 4B value 15B max_entries 1 memlock 4096B
    btf_id 254 frozen
```

在从 C 程序编译的对象文件中，bss（Block Started by Symbol）段通常用于存储全局变量，你可以使用 bpftool 检查其内容，例如：

```
$ bpftool map dump name hello.bss
[{
    "value": {
        ".bss": [{
            "counter": 11127
           }
        ]
    }
 }
]
```

也可以使用 bpftool map dump id 165 来检索相同的信息。如果多次运行这两个命令，你会发现计数器值增加了，因为每次接收到网络数据包时程序都会运行一次，counter 会加一。

正如你将在第 5 章中了解到的那样，如果 BTF（BPF Type Format）信息可用，bpftool 可以优雅地打印映射中的字段名称（在这里是变量名 counter）。而 BTF 信息只有在编译时使用了-g 标志才会包含在内。如果在编译步骤中省略了该标志，你只会看到类似于以下的输出：

```
$ bpftool map dump name hello.bss
key: 00 00 00 00 value: 19 01 00 00
Found 1 element
```

没有 BTF 信息，bpftool 就无法知道源代码中使用的变量名是什么。由于此映射中只有一个项，你可以推断出十六进制值 19 01 00 00 就是 counter 的值（十进制中为 281，因为字节序是从低位开始排序的）。

这里的 eBPF 程序使用映射来读写全局变量，映射其实也可用于保存静态数据。

名为 hello.rodata 的映射提示了这可能是与 hello 程序相关的静态数据。你可以转储该映射的内容，以查看它保存的字符串：

```
$ bpftool map dump name hello.rodata
[{
        "value": {
            ".rodata": [{
```

```
                "hello._____fmt": "Hello World %d"
                }
            ]
        }
    }
]
```

如果没有使用-g 标志编译对象，将会看到类似以下的输出：

```
$ bpftool map dump id 166
key: 00 00 00 00 value: 48 65 6c 6c 6f 20 57 6f  72 6c 64 20
   25 64 00
Found 1 element
```

这个映射中有一个键值对，值包含 12 字节的数据，以 0 结尾。这些字节很可能是字符串"Hello World %d" 的 ASCII 表示形式。

现在我们已经完成了对这个程序及其映射的检查，是时候进行清理工作了，让我们来将程序从事件中分离。

3.9 分离程序

你可以使用以下命令将程序从网络接口中卸载：

```
$ bpftool net detach xdp dev eth0
```

如果命令成功执行，将不会有任何输出。但你可以通过 bpftool net list 输出中缺少的 XDP 条目来确认程序已经分离：

```
$ bpftool net list
xdp:

tc:

flow_dissector:
```

然而，程序仍然还在内核中：

```
$ bpftool prog show name hello
395: xdp name hello tag 9d0e949f89f1a82c gpl
    loaded_at 2022-12-19T18:20:32+0000 uid 0
    xlated 48B jited 108B memlock 4096B map_ids 4
```

3.10 卸载程序

目前 bpftool 中没有 bpftool prog load 的反向操作，但是你可以通过删除固定的伪文件来从内核中删除 eBPF 程序：

```
$ rm /sys/fs/bpf/hello
$ bpftool prog show name hello
```

这个 bpftool 命令没有任何输出，因为程序已经从内核中卸载。

3.11 BPF 间调用

在前面的章节中，你看到了尾调用的示例，并提到现在还可以从 eBPF 程序中调用函数。让我们看一个简单的例子，这个例子可以像尾调用一样附加到 sys_enter 跟踪点上，但这次它将跟踪系统调用的操作码。你可以在第 3 章的 hello-func.bpf.c 文件中找到这段代码。

为了说明问题，我编写了一个非常简单的函数，它从跟踪点参数中提取系统调用的操作码：

```
static __attribute((noinline)) int get_opcode(struct
    bpf_raw_tracepoint_args *ctx) {
     return ctx->args[1];
}
```

编译器可能会将这个非常简单的函数内联展开，因为我只会在一个地方调用它。为了说明这个例子的目的，我特意添加了 ___attribute((noinline)) 来强制编译器遵循我的要求，不内联展开。正常情况下，你不应该加上这个属性，应该让编译器根据需要自行优化。

调用这个函数的 eBPF 函数如下所示：

```
SEC("raw_tp")
int hello(struct bpf_raw_tracepoint_args *ctx) {
    int opcode = get_opcode(ctx);
    bpf_printk("Syscall: %d", opcode);
    return 0;
}
```

将编译后的 eBPF 代码加载到内核中，并使用 bpftool 确认加载成功：

```
$ bpftool prog load hello-func.bpf.o /sys/fs/bpf/hello
$ bpftool prog list name hello
893: raw_tracepoint name hello tag 3d9eb0c23d4ab186 gpl
    loaded_at 2023-01-05T18:57:31+0000 uid 0
```

```
xlated 80B jited 208B memlock 4096B map_ids 204
btf_id 302
```

在这个例子中，我特别关注的是检查 eBPF 字节码以查看 get_opcode() 函数的内容。

```
$ bpftool prog dump xlated name hello
int hello(struct bpf_raw_tracepoint_args * ctx):
; int opcode = get_opcode(ctx);                          // 1
    0: (85) call pc+7#bpf_prog_cbacc90865b1b9a5_get_opcode
; bpf_printk("Syscall: %d", opcode);
    1: (18) r1 = map[id:193][0]+0
    3: (b7) r2 = 12
    4: (bf) r3 = r0
    5: (85) call bpf_trace_printk#-73584
; return 0;
    6: (b7) r0 = 0
    7: (95) exit
int get_opcode(struct bpf_raw_tracepoint_args * ctx): // 2
; return ctx->args[1];
    8: (79) r0 = *(u64 *)(r1 +8)
; return ctx->args[1];
    9: (95) exit
```

1. 在这里，你可以看到 hello() eBPF 程序调用了 get_opcode() 函数。在偏移量为 0 的 eBPF 指令是 0x85，根据指令集文档，这对应于” 函数调用”。它不会执行下一条指令，而是会跳转到后面七条指令（pc+7），也就是偏移量为 8 的指令。

2. 这是 get_opcode() 的字节码，如你所希望的，第一条指令在偏移量为 8 的位置。

函数调用指令需要将当前状态压入 eBPF 虚拟机的堆栈，以便在被调用的函数退出后，可以在调用函数中继续执行。由于堆栈大小限制为 512 字节，BPF 到 BPF 的调用不能太多，以免嵌套太深。

如果想了解更多关于尾调用和 BPF 到 BPF 调用的详细信息，可以阅读 Cloudflare 的博客上 Jakub Sitnicki 的优秀文章：“Assembly within! BPF tail calls on x86 and ARM”。

3.12 总结

在本章中，你看到了一些 C 示例代码如何转换成 eBPF 字节码，然后再编译为机器码，以便在内核中执行。你还学习了如何使用 bpftool 工具来检查加载到内核中的程序和映射，并将 eBPF 程序附加到 XDP 事件上。

此外，你还看到了不同类型的 eBPF 程序，它们由不同类型的事件触发。XDP 事件由

网络接口上的数据包到达触发，而 kprobe 和 tracepoint 事件则由命中内核代码中的某个特定点触发。我将在第 7 章中讨论一些其他类型的 eBPF 程序。

你还了解了如何使用映射来实现 eBPF 程序的全局变量，并了解了 BPF 到 BPF 的函数调用 (尾调用)。

下一章将更详细地介绍在 bpftool（或任何其他用户空间代码）加载程序并将其附加到事件时，系统调用层级上发生了什么。

3.13 练习

以下是一些进一步探索 BPF 程序的建议：

1. 尝试使用 ip link 命令来附加和解除附加 XDP 程序。例如：

```
$ ip link set dev eth0 xdp obj hello.bpf.o sec xdp
$ ip link set dev eth0 xdp off
```

2. 运行第二章的任何一个 BCC 示例，并在程序运行时在新终端窗口用 bpftool 检查加载的程序。以下是运行 hello-map.py 示例时的输出：

```
$ bpftool prog show name hello
197: kprobe name hello tag ba73a317e9480a37 gpl
    loaded_at 2022-08-22T08:46:22+0000 uid 0
    xlated 296B jited 328B memlock 4096B map_ids 65
    btf_id 179
    pids hello-map.py(2785)
```

你还可以使用 bpftool prog dump 命令查看这些程序的字节码和机器码。

3. 运行第二章的 hello-tail.py 示例，并在其运行时查看加载的程序，你会发现每个尾调用程序都单独列出了，类似于以下内容：

```
$ bpftool prog list
...
120: raw_tracepoint name hello tag b6bfd0e76e7f9aac gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 160B jited 272B memlock 4096B map_ids 29
        btf_id 124
        pids hello-tail.py(3590)
121: raw_tracepoint name ignore_opcode tag a04f5eef06a7f555 gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 16B jited 72B memlock 4096B
        btf_id 124
        pids hello-tail.py(3590)
```

```
122: raw_tracepoint name hello_exec tag 931f578bd09da154 gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 112B jited 168B memlock 4096B
        btf_id 124
        pids hello-tail.py(3590)
123: raw_tracepoint name hello_timer tag 6c3378ebb7d3a617 gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 336B jited 356B memlock 4096B
        btf_id 124
        pids hello-tail.py(3590)
```

你还可以使用 bpftool prog dump xlated 命令查看字节码指令，并将其与前面的 “BPF to BPF Calls” 进行比较

4. 对于这个建议要小心，最好只是思考为什么会发生这种情况，而不是去尝试它！如果从一个 XDP 程序中返回值为 0，这对应于 XDP_ABORTED，它告诉内核中止对该数据包的进一步处理。这可能有点违反直觉，因为在 C 中，返回值为 0 通常表示成功，但事实就是如此。因此，如果您尝试修改程序返回 0，并将其附加到虚拟机的 eth0 接口上，所有网络数据包都将被丢弃。如果你正在使用 SSH 连接到该机器，这将是相当不幸的，你可能需要重启机器才能恢复访问！

在容器中运行 XDP 程序，这会附加到虚拟以太网口上。https://github.com/lizrice/lb-from-scratch中有一个这样的示例。

第四章　bpf() 系统调用

用户空间程序希望内核代表它们执行某些操作时，会通过系统调用 API 发出请求。因此，如果用户空间程序要将 eBPF 程序加载到内核中，就必定涉及系统调用。实际上，有一个名为 bpf() 的系统调用，本章中，我会向你展示如何使用它来加载 eBPF 程序以及与 eBPF 程序和映射进行交互。

值得注意的是，运行在内核中的 eBPF 代码不是通过系统调用来访问映射。系统调用接口仅由用户空间程序使用。相反，eBPF 程序使用辅助函数来读写映射。

如果你写 eBPF 程序，很可能不会直接调用这些 bpf() 系统调用。本书后面讨论的库提供了这些系统调用更高级的抽象，以使编码更容易。也就是说，这些抽象通常与本章中所见的底层系统调用命令直接对应。无论你使用哪个库，都需要掌握本章中提到的底层操作——加载程序、创建和访问映射等。

在向你展示 bpf() 系统调用示例前，来看看bpf() 的 man 页中的内容，即 bpf() 用于”对 eBPF 映射或程序执行命令”。bpf() 的签名如下：

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

bpf() 的第一个参数 cmd 指定要执行的命令。bpf() 系统调用不仅仅用于某一个特定任务——有许多不同的命令都可用来操作 eBPF 程序和映射。图 4-1 概述了用户空间代码可能使用的一些常见命令，包括加载 eBPF 程序、创建映射、将程序附加到事件上以及访问映射中的键-值对。

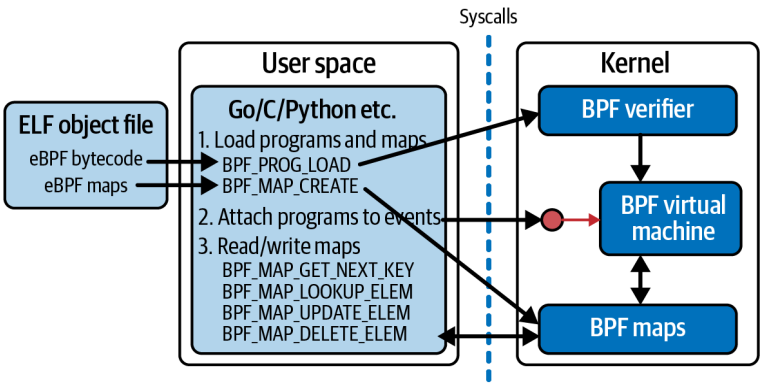

图 4.1: 用户空间程序通过系统调用与内核中的 eBPF 程序和映射进行交互

bpf() 系统调用的 attr 参数保存着 cmd 命令的所有参数，而 size 用来表示 attr 的字节大小。

你已经在第一章中见过 strace，当时我使用它来展示用户空间代码如何通过系统调用 API 进行多个请求。在本章中，我将使用 strace 来演示如何使用 bpf() 系统调用。strace 的输出包括每个系统调用的参数，但为了避免输出过于混乱，我将省略 attr 参数的许多细节，除非它特别值得关注。

你可以在github.com/lizrice/learning-ebpf找到代码以及运行代码的环境设置说明。本章的代码位于 chapter4 目录中。

在本示例中，我将用一个名为 hello-buffer-config.py 的 BCC 程序，它建立在第二章的示例之上。与 hello-buffer.py 示例类似，该程序在每次运行时都向 perf 缓冲区发送一条消息，将内核中的 execve() 系统调用事件的信息传递给用户空间。hello-buffer-config.py 这个版本新加的功能是允许为每个用户 ID 配置不同的消息。

下面是 eBPF 程序源码：

```
struct user_msg_t {                              // 1
    char message[12];
};

BPF_HASH(config, u32, struct user_msg_t);  // 2

BPF_PERF_OUTPUT(output);                         // 3

struct data_t {                                  // 4
    int pid;
    int uid;
    char command[16];
    char message[12];
};

int hello(void *ctx) {                           // 5
    struct data_t data = {};
    struct user_msg_t *p;
    char message[12] = "Hello World";

    data.pid = bpf_get_current_pid_tgid() >> 32;
    data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

    bpf_get_current_comm(&data.command, sizeof(data.command))
        ;
```

```
    p = config.lookup(&data.uid);           // 6
    if (p != 0) {
        bpf_probe_read_kernel(&data.message, sizeof(data.
            message), p->message);
    } else {
        bpf_probe_read_kernel(&data.message, sizeof(data.
            message), message);
    }

    output.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
```

1. 这行代码表明一个结构体定义：user_msg_t，用于保存一个含 12 字符的消息。

2. 使用 BCC 宏 BPF_HASH 定义一个哈希表映射 config。它将保存类型为 user_msg_t 的值，以 u32 类型的键进行索引，这个大小是适合用户 ID 的。（如果不指定键和值的类型，BCC 默认为两者都使用 u64。）

3. perf 缓冲区的输出与第 2 章完全相同。你可以向缓冲区提交任意数据，因此这里不需要指定任何数据类型...

4. 本示例程序始终提交一个 data_t 结构，与第 2 章的示例代码一样。

5. eBPF 程序的其余部分与你之前看到的 hello() 版本没有变化。

6. 唯一的区别是，在使用辅助函数获取用户 ID 后，会查找该用户 ID 作为键的 config 哈希映射中的条目。如果存在匹配的条目，值里将包含一条消息，该消息会替代默认值“Hello World”。

Python 代码中有两行额外的代码：

```
b["config"][ct.c_int(0)] = ct.create_string_buffer(b"Hey root
    !")
b["config"][ct.c_int(501)] = ct.create_string_buffer(b"Hi
    user 501!")
```

这两行代码为用户 ID 为 0 和 501 的 config 哈希表定义了消息，分别对应于 root 用户和虚拟机上的用户 ID。此代码使用 Python 的 ctypes 包确保键和值与 C 中的 user_msg_t 定义使用相同的类型。

下面是这个示例的一些输出，以及我在第二个终端中为获取输出而运行的命令：

```
Terminal 1                          Terminal 2
$ ./hello-buffer-config.py
37926 501 bash Hi user 501!         ls
```

```
37927 501 bash Hi user 501!     sudo ls
37929 0 sudo Hey root!
37931 501 bash Hi user 501!     sudo -u daemon ls
37933 1 sudo Hello World
```

既然你知道了这个程序的功能，接下来就展示下运行时 bpf() 系统调用的情况。使用 strace 来运行 eBPF 程序，并指定-e bpf 以表示只看 bpf() 系统调用：

```
$ strace -e bpf ./hello-buffer-config.py
```

如果你自己尝试运行这个命令，将会看到对 bpf() 系统调用的多个调用。对于每个调用，你都会看到 bpf() 系统调用里承载的具体命令。大致如下：

```
bpf(BPF_BTF_LOAD, ...) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY
   ...) = 4
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_HASH...) = 5
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE,...
   prog_name="hello",...) = 6
bpf(BPF_MAP_UPDATE_ELEM, ...}
...
```

我不会讨论每个调用的参数，只会重点关注那些我认为有助于讲述用户空间程序与 eBPF 程序交互的部分。

4.1 加载 BTF 数据

第一个 bpf() 调用如下所示：

```
bpf(BPF_BTF_LOAD, {btf="\237\353\1\0...}, 128) = 3
```

可以在输出中看到的命令是 BPF_BTF_LOAD。这只是一组有效命令中的其中一个，至少在撰写本文时。这些命令在内核源代码中有最全面文档。

如果你使用的是比较旧的 Linux 内核，可能不会看到带有这个命令的调用，因为它与BTF（BPF Type Format）有关。BTF 允许 eBPF 程序在不同的内核版本之间移植–在一台机器上编译程序，并在另一台使用不同内核版本（不同内核数据结构）的机器上使用它。

这个 bpf() 调用将一个 BTF 数据块加载到内核中，而 bpf() 系统调用的返回码（3）是一个文件描述符，用来引用该数据。

文件描述符是打开文件（或类似文件的对象）的标识符。如果你打开一个文件（使用 open() 或 openat() 系统调用），返回代码将是文件描述符，然后这个文件描述符会作为参

数传递给其他系统调用，例如 read() 或 write()，以对该文件执行操作。在这里，数据块不完全是一个文件，但它被理解成文件（Linux 下一切皆是文件），所以就赋予了一个文件描述符作为标识符，以便于对其进行操作。

4.2 创建映射

接下来的 bpf() 调用创建了 perf 缓冲区映射：

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    , key_size=4,
value_size=4, max_entries=4, ... map_name="output", ...},
    128) = 4
```

你可能已经猜到了，从命令名 BPF_MAP_CREATE 可以看出，这个调用是用来创建一个 eBPF 映射。该映射的类型是 PERF_EVENT_ARRAY，命名为 output。此 perf 映射中的键和值的长度为 4 字节。还有一个限制，即该映射可以容纳的键值对的数量最多为 4 个，由 max_entries 字段定义；我将在后面解释为什么该映射中有四个条目。返回值 4 是用户空间代码用来访问该映射的文件描述符。

下一个 bpf() 系统调用创建了 config 映射：

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_HASH, key_size=4,
    value_size=12,
max_entries=10240... map_name="config", ...btf_fd=3,...},
    128) = 5
```

这个映射是一个哈希表映射，键的长度为 4 字节（对应于保存用户 ID 的 32 位整数），值的长度为 12 字节（与 msg_t 结构的长度相匹配）。我没有指定表大小，因此用了 BCC 的默认大小，即 10240 个条目。这个 bpf() 系统调用返回的文件描述符为 5。

可以看到字段 btf_fd=3，它告诉内核使用先前获取的 BTF 文件描述符 3。BTF 信息描述了数据结构的布局，包括映射中使用的键和值类型的信息。这种信息很方便类似 bpftool 这样的工具用来干净地打印出映射转储信息，使其可读性大大加强-你在第 3 章中已经见过一个示例了。

4.3 加载程序

目前为止，在这个示例程序中，你看到了使用系统调用将 BTF 数据加载到内核并创建一些 eBPF 映射。接下来要做的是用以下 bpf() 系统调用将 eBPF 程序加载到内核中

```
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt
    =44,
```

```
insns=0xffffa836abe8, license="GPL", ... prog_name="hello",
    ...
expected_attach_type=BPF_CGROUP_INET_INGRESS, prog_btf_fd
    =3,...}, 128) = 6
```

有几个字段值得关注:

- prog_type 字段描述了程序类型，这里表示它要附加到 kprobe 上。
- insn_cnt 字段表示“指令计数”。这是程序中的字节码指令数量。
- 构成这个 eBPF 程序的字节码指令在内存中保存地址存储在 insns 字段。
- 这个程序许可协议为 GPL 许可，以便可以用 GPL 许可的 BPF 辅助函数。
- 程序名为 hello。
- BPF_CGROUP_INET_INGRESS 的 expected_attach_type 似乎与网络流量有关，但这个 eBPF 程序将被附加到一个 kprobe 上，expected_attach_type 字段仅适用于某些程序类型，而 BPF_PROG_TYPE_KPROBE 不是其中之一。BPF_CGROUP_INET_INGRESS 是 BPF 附加类型列表中的第一个，所以它的值为 0。
- prog_btf_fd 字段告诉内核使用先前加载的 BTF 数据块与该程序配对。这里的值 3 对应于从 BPF_BTF_LOAD 系统调用中看到的文件描述符（它与 config 映射使用相同的 BTF 数据块）。

如果程序未通过验证，系统调用将返回一个负值，但这里它返回了文件描述符 6。这个文件描述符具有如表 4-1 所示的含义。

表 4.1: 加载并运行 hello-buffer-config.py 后的文件描述符

File descriptor	Represents
3	BTF 数据
4	输出 perf buffer map
5	配置 hash table map
6	eBPF 程序: hello

4.4 从用户空间修改映射

在用户空间的 Python 源码中，你看到了为 ID 为 0 的 root 用户和 ID 为 501 的用户配置特殊消息的代码行:

```
b["config"][ct.c_int(0)] = ct.create_string_buffer(b"Hey root
    !")
b["config"][ct.c_int(501)] = ct.create_string_buffer(b"Hi
    user 501!")
```

通过类似以下的系统调用，你可以看到这些条目在映射中定义:

```
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=5, key=0xffffa7842490, value
   =0xffffa7a2b410, flags=BPF_ANY}, 128) = 0
```

BPF_MAP_UPDATE_ELEM 命令用于更新映射中的键值对。BPF_ANY 标志表示如果该键在映射中不存在则创建。这里有两个这样的调用，对应于为两个不同用户 ID 配置的两个条目。

map_fd 字段表示正在操作的映射。这种情况下，你可以看到当前操作值是 5，这是在创建 config 映射时先前返回的文件描述符值。

文件描述符是由内核为特定进程分配的，因此值 5 仅对该特定的用户空间进程有效，即运行 Python 程序的进程。然而，多个用户空间程序（和内核中的多个 eBPF 程序）可以同时访问相同的映射。访问同一映射的两个用户空间程序可能会被分配不同的文件描述符；同样，两个用户空间程序可能对完全不同的映射具有相同的文件描述符值。

键和值都是指针，因此你无法从 strace 输出中获取键或值。然而，你可以使用 bpftool 查看映射的内容，并看到类似以下的内容：

```
$ bpftool map dump name config
[{
        "key": 0,
        "value": {
            "message": "Hey root!"
        }
    },{
        "key": 501,
        "value": {
            "message": "Hi user 501!"
        }
    }
]
```

bpftool 是如何知道怎样格式化这个输出的呢？例如，它是如何知道值是一个结构，其中包含一个名为 message 的字符串字段？答案是它使用了在 BPF_MAP_CREATE 系统调用中包含的 BTF 信息，BTF 里面有具体的定义。

现在你已经看到了用户空间如何与内核交互，加载程序和映射以及更新映射信息。目前为止，你见过的系统调用序列中，程序尚未被附加到事件上。这一步是必须的，否则，程序将永远不会被触发。

值得注意的是：不同类型的 eBPF 程序可以以各种不同的方式附加到不同的事件上！本章后面将展示此示例中用于附加到 kprobe 事件的系统调用，这种情况下不涉及 bpf() 调用。与之相对的，，本章末尾的练习中将向你展示另一个示例，其中会使用 bpf() 系统调用将程序附加到原始跟踪点事件。

在详细讨论细节之前，我想讨论一下停止运行程序时会发生什么。如果你这么做了，就会发现程序和映射会自动卸载，这是因为内核使用了引用计数来跟踪它们，停止了就自动卸载。

4.5 BPF 程序和映射引用

通过 bpf() 系统调用将 BPF 程序加载到内核时，会返回一个文件描述符。在内核中，这个文件描述符是对程序的引用。进行系统调用的用户空间进程拥有这个文件描述符；进程退出时，文件描述符被释放，程序的引用计数减少。没有对 BPF 程序的引用时，内核会删除它。

将程序固定到文件系统会创建一个额外的引用。

4.5.1 固定（Pinning）

你已经在第 3 章中看到了程序固定的示例，使用的是以下命令：

```
bpftool prog load hello.bpf.o /sys/fs/bpf/hello
```

固定的对象并不是真正的持久化到磁盘文件。它们被创建在一个虚拟文件系统上，它的行为类似于基于磁盘的文件系统，具有目录和文件。但它们只保存于内存中，这意味着系统重启后就变了。

如果 bpftool 只用来加载程序而不进行固定那将是无意义的，因为文件描述符在 bpftool 退出时被释放，如果引用计数为零，程序将被删除，因此不会有任何有用的功能。但将其固定到文件系统意味着程序有了一个引用，因此程序在命令完成后仍然保持加载状态。

将 BPF 程序附加到触发它的事件时，引用计数也会增加。这些引用计数的行为取决于 BPF 程序类型。有一些与跟踪相关的类型（如 kprobe 和 tracepoint）始终与用户空间进程相关联；对于这些类型的 eBPF 程序，内核的引用计数在该进程退出时减少。附加到网络堆栈或 cgroup（“控制组”）中的程序不与任何用户空间进程相关联，因此即使加载它们的用户空间程序退出，它们仍然保持不变。你在使用 ip link 命令加载 XDP 程序时已经见过这个示例了：

```
ip link set dev eth0 xdp obj hello.bpf.o sec xdp
```

ip 命令完成退出后内核中仍然加载着 XDP 程序：

```
$ bpftool prog list
...
1255: xdp name hello tag 9d0e949f89f1a82c gpl
        loaded_at 2022-11-01T19:21:14+0000 uid 0
        xlated 48B jited 108B memlock 4096B map_ids 612
```

这个程序的引用计数非零，因为它附加到了 XDP 事件，该事件在 ip link 命令完成后仍然存在。

eBPF 映射也有引用计数，当引用计数降到零时，映射会被清理。每个使用映射的 eBPF 程序都会增加引用计数值，用户空间程序持有的每个文件描述符对映射的引用也会增加计数值。

可能 eBPF 程序的源代码中定义了映射，但实际上程序并没有引用它。如果想存储关于程序的元数据，你可以将其定义为全局变量，这些信息将存储在映射中。如果 eBPF 程序对映射没有任何操作，程序到映射之间不会自动产生引用计数。有一个 BPF(BPF_PROG_BIND_MAP) 系统调用可以将映射与程序关联起来，这样在用户空间程序加载器退出不再持有映射的文件描述符引用时，映射也不会立即被清理。

映射还可以固定到文件系统，用户空间程序可以通过路径来访问映射。

Alexei Starovoitov 在他的博客文章 “Lifetime of BPF Objects” 中对 BPF 引用计数和文件描述符进行了很好的阐述。

创建对 BPF 程序引用的另一种方法是使用 BPF 链接。

4.5.2 BPF 链接

BPF 链接在 eBPF 程序和其附加的事件之间提供了一个抽象层。BPF 链接本身可以固定到文件系统，这会为程序创建一个引用。意味着加载程序到内核的用户空间进程可以终止，但程序仍然保持加载状态。用户空间加载器程序的文件描述符被释放，减少了对程序的引用计数，但由于 BPF 链接，引用计数仍然非零。

如果你按照本章末尾的练习进行操作，你将看到 BPF 链接的实际效果。现在，让我们回到 hello-buffer-config.py 中使用的一系列 bpf() 系统调用。

4.6 eBPF 中涉及的其他系统调用

回顾一下，到目前为止，你已经看到了一些 bpf() 系统调用，这些调用将 BTF 数据、程序和映射以及映射数据添加到内核中。

本章的剩余部分相对详细地介绍了使用性能缓冲区、环形缓冲区、kprobe 和映射时涉及的系统调用。并不是所有的 eBPF 程序都需要执行这些操作，所以如果你时间紧或者觉得内容过于详细，可以跳到本章总结部分。

4.6.1 初始化 Perf 缓冲区

bpf(BPF_MAP_UPDATE_ELEM) 调用用于向 config 映射中添加条目。接下来，输出显示了一些类似于以下调用的内容：

```
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0xffffa7842490, value
   =0xffffa7a2b410,
flags=BPF_ANY}, 128) = 0
```

这些调用与定义 config 映射条目的调用非常相似，只是这种情况下，映射的文件描述符是 4，表示输出性能缓冲区映射。

与之前一样，键和值都是指针，因此无法从这个 strace 输出中确定键或值的数值。这个系统调用重复了四次，所有参数的值都相同，尽管无法确定指针所持有的值是否在每次调用间发生了变化。仔细观察这些 BPF_MAP_UPDATE_ELEM bpf() 调用，还存在一些关于如何设置和使用缓冲区的未解答的问题：

- 为什么会有四个 BPF_MAP_UPDATE_ELEM 调用？是否与输出映射的最大条目数为四有关？
- 在这四个 BPF_MAP_UPDATE_ELEM 之后，strace 输出中不再出现其他的 bpf() 系统调用。这看起来有点怪，因为映射是为了让 eBPF 程序在每次触发时写入数据，而用户空间代码显示数据。显然，这些数据并不是通过 bpf() 系统调用从映射中获取的，那么它是如何获取的呢？

为了解释这些问题，我会在运行此示例时让 strace 显示更多的系统调用，像这样：

```
$ strace -e bpf,perf_event_open,ioctl,ppoll ./hello-buffer-
    config.py
```

为简洁起见，我将忽略与本例 eBPF 功能无关的 ioctl() 调用。

4.6.2 附加到 Kprobe 事件

文件描述符 6 被分配为表示 eBPF 程序 hello 加载到内核后的文件描述符。要将 eBPF 程序附加到事件上，还需要一个表示该特定事件的文件描述符。以下是 strace 输出中显示的为 execve() kprobe 创建文件描述符的行：

```
perf_event_open({type=0x6 /* PERF_TYPE_??? */, ...},...) = 7
```

根据 perf_event_open() 系统调用的man 页，它“创建一个允许测量性能信息的文件描述符”。从输出中可以看出，strace 不知道如何解释类型参数的值，但如果进一步查看 man 页，其中描述了 Linux 如何支持动态类型的性能测量单元：

```
...there is a subdirectory per PMU instance under /sys/bus/
    event_source/devices. In
each subdirectory there is a type file whose content is an
    integer that can be used in the
type field.
```

确实，如果在此目录下查找，你会找到一个名为 kprobe/type 的文件：

```
$ cat /sys/bus/event_source/devices/kprobe/type
6
```

从中可以看出，perf_event_open() 调用的类型值设置为 6，表示它是一种 kprobe 类型的 perf 事件。

strace 没有输出足够的细节来显示 kprobe 是否附加到 execve() 系统调用上，我希望这里有足够的信息来让读者相信返回的文件描述符代表的就是这种情况。

perf_event_open() 的返回码为 7，表示 kprobe 的 perf 事件的文件描述符，而你知道文件描述符 6 代表的是 hello eBPF 程序。perf_event_open() 的 man 页还解释了如何使用 ioctl() 来在两者之间创建附加关系：

```
PERF_EVENT_IOC_SET_BPF [...] allows attaching a Berkeley
   Packet Filter (BPF) program
to an existing kprobe tracepoint event. The argument is a BPF
    program file descriptor
that was created by a previous bpf(2) system call.
```

这里解释了在 strace 输出中看到的下一个 ioctl() 系统调用，其中的参数是指两个文件描述符：

```
ioctl(7, PERF_EVENT_IOC_SET_BPF, 6) = 0
```

还有另一个 ioctl() 调用，用来打开 kprobe 事件：

```
ioctl(7, PERF_EVENT_IOC_ENABLE, 0) = 0
```

有了这些设置，在这台机器上运行 execve() 时，eBPF 程序就会触发。

4.6.3 设置和读取 Perf 事件

我之前提过与输出性能缓冲区相关的四个 bpf(BPF_MAP_UPDATE_ELEM) 调用。通过添加额外的系统调用跟踪，strace 输出显示了下面四个类似的信息：

```
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /*
   PERF_ATTR_SIZE_??? */,
config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, X, -1,
   PERF_FLAG_FD_CLOEXEC) = Y

ioctl(Y, PERF_EVENT_IOC_ENABLE, 0) = 0

bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0xffffa7842490, value
   =0xffffa7a2b410,
flags=BPF_ANY}, 128) = 0
```

X 表示输出中的值 0、1、2 和 3，对应调用的四个实例。根据 perf_event_open() 系统调用的手册，你会发现这是 CPU 相关的调用，而在它之前的字段是 pid 或进程 ID。根

据手册：

```
pid == -1 and cpu >= 0
This measures all processes/threads on the specified CPU.
```

调用发生四次的原因是我的笔记本电脑有四个 CPU 核，这就解释了为什么"output"性能缓冲区映射中有四个条目：每个 CPU 核心都有一个条目。它也解释了映射类型名称 BPF_MAP_TYPE_PERF_EVENT_ARRAY 中的"array" 部分，因为该映射不仅代表一个性能环形缓冲区，而是一个缓冲区数组，每个核心一个。

编写 eBPF 程序你不需要关注 CPU 核心数这样的细节，因为这将由第 10 章中讨论的 eBPF 库为你处理。

每个 perf_event_open() 调用都返回一个文件描述符，我用 Y 来表示，它们的值分别为 8、9、10 和 11。ioctl() 系统调用为每个文件描述符启用了性能输出。BPF_MAP_UPDATE_ELEM bpf() 系统调用将映射条目设置为指向每个 CPU 核心的性能环形缓冲区，以指示可以写入数据的位置。

然后，用户空间代码可以对这四个输出流文件描述符中的所有文件描述符使用 ppoll() 调用，以便无论哪个核心运行了 execve() kprobe 事件触发的 eBPF 程序 hello，都可以获取数据输出。下面是调用 ppoll() 的系统调用：

```
ppoll([{fd=8, events=POLLIN}, {fd=9, events=POLLIN}, {fd=10,
   events=POLLIN},
{fd=11, events=POLLIN}], 4, NULL, NULL, 0) = 1 ([{fd=8,
   revents=POLLIN}])
```

如果你运行示例程序会发现 ppoll() 调用会阻塞，直到有数据从其中一个文件描述符中读出。在触发 execve() 后，才会将返回数据写到屏幕，这导致 eBPF 程序写入数据，然后用户空间使用 ppoll() 调用来检索数据。

在第 2 章中，我提到如果内核版本为 5.8 或更高，现在更推荐的是使用 BPF 环形缓冲区而不是perf 缓冲区。

4.7 环形缓冲区

根据内核文档的描述，相较于 perf 缓冲区，环形缓冲区更受推荐，部分原因是出于性能考虑，但更重要的是为了确保数据的顺序，即使数据由不同的 CPU 核心提交。环形缓冲区只有一个，在所有核心之间共享。

将 hello-buffer-config.py 转换为使用环形缓冲区需要的更改并不多。在 GitHub 仓库中，你可以找到这个示例 chapter4/hello-ring-buffer-config.py。表 4-2 显示了它们间的区别。

由于这些更改只涉及输出缓冲区，与加载程序、配置映射和将程序附加到 kprobe 事件相关的系统调用保持不变。创建输出环形缓冲区映射的 bpf() 系统调用如下所示：

表 4.2: 使用 BCC perf 缓冲区和环形缓冲区的区别

hello-buffer-config.py	hello-ring-buffer-config.py
BPF_PERF_OUTPUT(output);	BPF_RINGBUF_OUTPUT(output, 1);
output.perf_submit(ctx, &data,sizeof(data));	output.ringbuf_output(&data,sizeof(data), 0);
b["output"].open_perf_buffer(print_event)	b["output"].open_ring_buffer(print_event)
b.perf_buffer_poll()	b.ring_buffer_poll()

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_RINGBUF, key_size
   =0, value_size=0,
max_entries=4096, ... map_name="output", ...}, 128) = 4
```

在 strace 输出中的一个重大区别是，你观察到的一系列不同的 perf_event_open()、ioctl() 和 bpf(BPF_MAP_UPDATE_ELEM) 系统调用没有展示设置 perf 缓冲区的过程。对于环形缓冲区，只有一个文件描述符在所有 CPU 核心之间共享。

在撰写本文时，BCC 在处理 perf 缓冲区时使用了之前展示的 ppoll 机制，但它使用了更新的 epoll 机制来等待环形缓冲区中的数据。现在，先来了解下 ppoll 和 epoll 间的区别。

在 perf 缓冲区示例中，hello-buffer-config.py 生成了一个 ppoll() 系统调用，就像这样：

```
ppoll([{fd=8, events=POLLIN}, {fd=9, events=POLLIN}, {fd=10,
   events=POLLIN},
{fd=11, events=POLLIN}], 4, NULL, NULL, 0) = 1 ([{fd=8,
   revents=POLLIN}])
```

请注意，它传递了文件描述符集合 8、9、10 和 11，用于用户空间进程检索数据。每当此 poll 事件返回数据时，必须再次调用 ppoll() 来重新设置相同的文件描述符集合。而使用 epoll 时，文件描述符集合由内核管理。

hello-ring-buffer-config.py 设置对输出环形缓冲区的访问权限时可以看到以下一系列与 epoll 相关的系统调用：

首先，用户空间程序请求在内核中创建一个新的 epoll 实例：

```
epoll_create1(EPOLL_CLOEXEC) = 8
```

这将返回文件描述符 8。然后调用 epoll_ctl()，告诉内核将文件描述符 4（输出缓冲区）添加到该 epoll 实例的文件描述符集合中：

```
epoll_ctl(8, EPOLL_CTL_ADD, 4, {events=EPOLLIN, data={u32=0,
   u64=0}}) = 0
```

用户空间程序使用 epoll_pwait() 等待环形缓冲区中的数据。此调用仅在数据可用时返回：

```
epoll_pwait(8, [{events=EPOLLIN, data={u32=0, u64=0}}], 1,
   -1, NULL, 8) = 1
```

当然，如果你用像 BCC（或 libbpf 或本书后面介绍的任何其他库）这样的框架编写代码，你实际上不需要了解如何通过 perf 或环形缓冲区从内核获取信息这样的细节。希望你能发现这些内部细节的趣味并了解它们的工作原理。

然而，你可能发现自己需要编写代码从用户空间访问映射，了解如何实现这一点会很有帮助。在本章前面，我使用 bpftool 来检查配置映射的内容。由于它是在用户空间运行的工具，所以可以用 strace 来查看它在检索这些信息时进行的系统调用。

4.8 从映射中读取信息

下面的命令显示了 bpftool 在读取配置映射内容时进行的一系列 bpf() 系统调用：

```
$ strace -e bpf bpftool map dump name config
```

该序列包括两个主要步骤：

- 迭代所有映射，找查具有名称 config 的映射。
- 如果找到匹配的映射，则迭代该映射中的所有元素。

4.8.1 查找映射

输出以一系列重复的相似调用开始，因为 bpftool 遍历所有映射，查找具有名称 config 的映射：

```
bpf(BPF_MAP_GET_NEXT_ID, {start_id=0,...}, 12) = 0   // 1
bpf(BPF_MAP_GET_FD_BY_ID, {map_id=48...}, 12) = 3    // 2
bpf(BPF_OBJ_GET_INFO_BY_FD, {info={bpf_fd=3, ...}}, 16) = 0 // 3
bpf(BPF_MAP_GET_NEXT_ID, {start_id=48, ...}, 12) = 0 // 4
bpf(BPF_MAP_GET_FD_BY_ID, {map_id=116, ...}, 12) = 3
bpf(BPF_OBJ_GET_INFO_BY_FD, {info={bpf_fd=3...}}, 16) = 0
```

1. BPF_MAP_GET_NEXT_ID 获取 start_id 指定值之后的下一个映射 ID。

2. BPF_MAP_GET_FD_BY_ID 返回指定映射 ID 的文件描述符。

3. BPF_OBJ_GET_INFO_BY_FD 检索由文件描述符引用的对象的信息。此信息包括其名称，以便 bpftool 可以检查这是否是它正在寻找的映射。

4. 重复流程 1 获取下一个映射 ID。

对于内核中加载的每个映射，都有这三个系统调用组，你还应该注意到用于 start_id 和 map_id 的值与这些映射的 ID 匹配。没有多条映射就不会有这样的的重复调用，BPF_MAP_GET_NE 返回 ENOENT 的值，如下所示：

```
bpf(BPF_MAP_GET_NEXT_ID, {start_id=133,...}, 12) = -1 ENOENT
   (No such file or
directory)
```

如果找到了匹配的映射，bpftool 将持有它的文件描述符，以便可以从该映射中读取数据。

4.8.2 读取映射元素

现在，bpftool 已经获得了对将要读取的映射的文件描述符引用。来看一下读取这些信息的系统调用序列：

```
bpf(BPF_MAP_GET_NEXT_KEY, {map_fd=3, key=NULL,  // 1
next_key=0xaaaaf7a63960}, 24) = 0
bpf(BPF_MAP_LOOKUP_ELEM, {map_fd=3, key=0xaaaaf7a63960, // 2
value=0xaaaaf7a63980, flags=BPF_ANY}, 32) = 0
[{                                                  // 3
    "key": 0,
    "value": {
        "message": "Hey root!"
    }
bpf(BPF_MAP_GET_NEXT_KEY, {map_fd=3, key=0xaaaaf7a63960, // 4
next_key=0xaaaaf7a63960}, 24) = 0
bpf(BPF_MAP_LOOKUP_ELEM, {map_fd=3, key=0xaaaaf7a63960,
value=0xaaaaf7a63980, flags=BPF_ANY}, 32) = 0
    },{
        "key": 501,
        "value": {
            "message": "Hi user 501!"
        }
bpf(BPF_MAP_GET_NEXT_KEY, {map_fd=3, key=0xaaaaf7a63960, // 5
next_key=0xaaaaf7a63960}, 24) = -1 ENOENT (No such file or
   directory)
    }                                               // 6
]
+++ exited with 0 +++
```

1. 首先，应用程序（bpftool）使用 bpf() 系统调用的 BPF_MAP_GET_NEXT_KEY 参数来查找映射中的下一个有效键。通过将键参数设置为 NULL 指针，它请求地图中的第

一个有效键。内核将键写入 next_key 指定的位置。

2. 给定一个键，应用程序请求其关联的值，并将其写入 value 指定的内存位置。

3. 此时，bpftool 获取了第一个键值对的内容，并将此信息写入屏幕。

4. 然后，bpftool 移动到地图中的下一个键，检索其值，并将此键值对写入屏幕。

5. 接下来的 BPF_MAP_GET_NEXT_KEY 调用返回 ENOENT，表示映射中没有更多条目了。

6. 在此，bpftool 完成了数据向屏幕的输出，并退出。

请注意，bpftool 将文件描述符 3 分配给 config，而 hello-buffer-config.py 中使用的文件描述符是 4，两者引用相同的映射。我之前提到的，文件描述符是进程特定的。

上述 bpftool 行为的分析展示了用户空间程序如何遍历可用的映射和其中存储的键值对。

4.9 总结

在本章中，你了解了用户空间代码如何使用 bpf() 系统调用来加载 eBPF 程序和映射。你看到了使用 BPF_PROG_LOAD 和 BPF_MAP_CREATE 命令创建程序和映射。

你了解到内核会跟踪对 eBPF 程序和映射的引用数量，并在引用计数降至零时释放它们。你还了解了将 BPF 对象固定到文件系统并使用 BPF 链接创建其他引用这样的概念。

你看到了使用 BPF_MAP_UPDATE_ELEM 在用户空间中创建映射条目的示例。还有类似的命令 BPF_MAP_LOOKUP_ELEM 和 BPF_MAP_DELETE_ELEM，用于从映射中检索和删除值。还有命令 BPF_MAP_GET_NEXT_KEY，用于查找映射中存在的下一个键。你可以使用它来遍历所有有效条目。

你还看到了用户空间程序示例，它们使用 perf_event_open() 和 ioctl() 将 eBPF 程序附加到 kprobe 事件。对于其他类型的 eBPF 程序，附加方法可能会有很大的不同，其中一些甚至使用 bpf() 系统调用。例如，有一个 bpf(BPF_PROG_ATTACH) 系统调用可用于附加 cgroup 程序，以及 bpf(BPF_RAW_TRACEPOINT_OPEN) 用于原始跟踪点。

我还展示了如何使用 BPF_MAP_GET_NEXT_ID、BPF_MAP_GET_FD_BY_ID 和 BPF_OBJ_GET_INFO_BY_FD 来定位内核中保存的映射和其他对象。

本章中还有一些未包含的其他 bpf() 命令，但目前你已经了解到了足够多的内容，对 eBPF 也有了更好的认识。

你还看到了一些 BTF 数据被加载到内核中，我提过 bpftool 使用这些信息来了解数据结构的格式，以便将其漂亮地打印出来。我还没有解释 BTF 数据的具体内容以及如何通过 BFT 来确保 eBPF 程序在内核版本间可移植。

4.10 练习

如果你希望进一步探索 bpf() 系统调用，下面是一些建议：

1. 确认通过 BPF_PROG_LOAD 系统调用的 insn_cnt 字段是否与使用 bpftool 转储翻译后的 eBPF 字节码的指令数量相对应（这是 bpf() 系统调用的man 页中所记录的）。

2. 运行两个示例程序的实例，以便有两个名为 config 的迎神。如果你运行 bpftool map dump name config，输出将包括两个不同映射。在 strace 下运行此命令，并通过系统调用输出跟踪不同的文件描述符的使用。你能看到它在哪里检索映射信息以及在哪里检索其中存储的键值对吗?

3. 运行示例程序时，使用 bpftool map update 修改 config 映射。使用 sudo -u 用户名来检查 eBPF 程序是否接受了这些变更。

4. 运行 hello-buffer-config.py 时，使用 bpftool 将该程序固定到 BPF 文件系统，如下所示：

```
bpftool prog pin name hello /sys/fs/bpf/hi
```

退出程序，并使用 bpftool prog list 检查 hello 程序是否仍然加载在内核中。你可以通用 rm /sys/fs/bpf/hi 来删除它。

5. 与附加到 kprobe 相比，附加到原始跟踪点在系统调用层面上要简单得多，因为它只涉及到一个 bpf() 系统调用。尝试将 hello-buffer-config.py 转换为附加到 sys_enter 的原始跟踪点，使用 BCC 的 RAW_TRACEPOINT_PROBE 宏（如果你完成了第 2 章的练习，你可能已经有一个合适的程序了）。你不需要在 Python 代码中显式附加程序，因为 BCC 会为你处理。在 strace 下运行此程序，你应该会看到类似于以下的系统调用：

```
bpf(BPF_RAW_TRACEPOINT_OPEN, {raw_tracepoint={name="sys_enter",
prog_fd=6}}, 128) = 7
```

跟踪点在内核中的名称为 sys_enter，具有文件描述符 6 的 eBPF 程序正在附加到该跟踪点。从现在开始，每当内核中的执行到达该跟踪点时，它将触发 eBPF 程序。

6. 运行BCC 工具集中的 opensnoop 应用程序。该工具将设置一些 BPF 链接，你可以使用 bpftool 查看，如下所示：

```
$ bpftool link list
116: perf_event prog 1849
        bpf_cookie 0
        pids opensnoop(17711)
117: perf_event prog 1851
        bpf_cookie 0
        pids opensnoop(17711)
```

确认程序 ID 与列出已加载的 eBPF 程序的输出匹配：

```
$ bpftool prog list
...
1849: tracepoint name tracepoint__syscalls__sys_enter_openat
```

```
        tag 8ee3432dcd98ffc3 gpl run_time_ns 95875 run_cnt 121
        loaded_at 2023-01-08T15:49:54+0000 uid 0
        xlated 240B jited 264B memlock 4096B map_ids 571,568
        btf_id 710
        pids opensnoop(17711)
1851: tracepoint name tracepoint__syscalls__sys_exit_openat
        tag 387291c2fb839ac6 gpl run_time_ns 8515669 run_cnt 120
        loaded_at 2023-01-08T15:49:54+0000 uid 0
        xlated 696B jited 744B memlock 4096B map_ids 568,571,569
        btf_id 710
        pids opensnoop(17711)
```

7. opensnoop 运行时，尝试使用 bpftool link pin id 116 /sys/fs/bpf/mylink（使用 bpftool link list 输出的一个链接 ID）来固定其中一个链接。你会发现即使在终止 opensnoop 之后，链接和相应的程序仍然保留在内核中。

8. 查看第 5 章的示例代码，你会发现使用 libbpf 库编写的 hello-buffer-config.py。libbpf 库会自动设置与其加载到内核中的程序的 BPF 链接。使用 strace 检查它进行的 bpf() 系统调用，并查看 bpf(BPF_LINK_CREATE) 系统调用。

第五章　CO-RE, BTF 和 Libbpf

在上一章中，你第一次接触到了 BTF（BPF Type Format）。本章将讨论其存在的原因以及如何使用它来保证 eBPF 程序在不同版本的内核上可移植。BTF 是 BPF “编译一次，到处运行”（CO-RE）这一承诺的关键部分，解决了不同内核版本间 eBPF 程序可移植性问题。

eBPF 程序访问内核数据结构，需要包含相关的 Linux 头文件，以便 eBPF 代码能够正确地定位这些数据结构中的字段。然而，Linux 内核处于不断开发中，这意味着在不同的内核版本间，内部数据结构可能会发生变化。将一台机器上编译的 eBPF 对象文件加载到具有不同内核版本的机器上是无法保证数据结构相同的。

CO-RE 方法以高效的方式解决可移植性问题。它允许 eBPF 程序包含编译时使用的数据结构布局信息，并提供了一种机制来调整字段的访问方式。如果在目标机器上的数据结构布局与之不同，只要程序不访问目标机器内核中不存在的字段或数据结构，程序就可以在不同的内核版本间进行移植。

但在深入讨论 CO-RE 的工作原理之前，让我们先来看看它为何有效。

5.1　BCC 实现可移植性的方法

在第 2 章中，我使用BCC展示了一个基本的 eBPF 程序 “Hello World”。BCC 项目是第一个流行的用于实现 eBPF 程序的项目，为用户空间和内核提供了一个相对容易理解的框架，适用于没有太多内核经验的程序员。为解决内核间的可移植性问题，BCC 采取了在目标机器上即时编译 eBPF 代码的方法。这种方法存在一些问题：

- 每台运行代码的目标机器上都需要安装编译工具链，以及内核头文件（这些文件并不总是默认存在）。
- 必须等待编译完成，然后才能启动工具，这导致每次启动工具时都有几秒的延迟。
- 如果在一组相同的机器上运行该工具，那么重复编译就是在浪费计算资源。
- 一些基于 BCC 的项目将它们的 eBPF 源代码和工具链打包到了容器镜像中，这样可以更容易地分发到机器上。但它并不能解决内核头文件存在的问题，而且如果每台机器上安装了多个 BCC 容器，意味着更多的重复。
- 嵌入式设备可能没有足够的内存来运行编译任务。

由于这些问题，如果你计划开发一个重要的 eBPF 项目，我不建议使用传统的 BCC

方法，特别是如果计划将你的项目分发给其他人使用。在本书中，我给出了一些基于 BCC 的示例，因为它是学习 eBPF 基本概念的好办法，尤其是 Python 用户空间代码非常简洁易读。如果你对此更为熟悉并且想快速组装一些简单的东西，BCC 框架是个完全不错的选择。但对于现代化的 eBPF 开发来说，这不是最佳方案。

CO-RE 方法为 eBPF 程序的跨内核可移植性问题提供了更好的解决方案。

iovisor/bcc上的 BCC 项目包含了各种命令行工具，可以用来观察 Linux 机器中各种行为信息。tools目录中的工具主要是使用我在本节中描述的基于 Python 实现的传统可移植性方法。在 BCC 的libbpf-tools目录中工具则是使用 C 编写的，它们利用了 libbpf 和 CO-RE，并且不会遇到刚刚列出的问题。

5.2 CO-RE 概述

CO-RE（编译一次，到处运行）方法在 eBPF 中含几个关键要素 [2]：

• BTF（BPF 类型格式）

BTF是一种用于表达数据结构布局和函数签名的格式。在 CO-RE 中，它用于确定编译时和运行时使用的结构间的差异。bpftool 这样的工具也可以借助 BTF 以人类可读的格式转储数据结构。从 Linux 5.4 开始，内核默认支持 BTF。

• 内核头文件

Linux 内核源码包括描述其使用数据结构的头文件，这些头文件在不同版本的 Linux 间可能会发生变化。eBPF 程序员可以选择包含单个头文件，或者可以使用 bpftool 从运行中的系统生成一个称为 vmlinux.h 的头文件，其中包含 BPF 程序可能需要的有关内核的所有数据结构信息。

• 编译器支持

Clang 编译器扩展，使用-g 标志编译 eBPF 程序时，它会包含基于描述内核数据结构的 BTF 信息导出的 CO-RE 重定位。GCC 编译器在12 版中也为 BPF 目标添加了 CO-RE 支持。

• 数据结构重定位的库支持

用户空间程序将 eBPF 程序加载到内核时，CO-RE 方法要求调整字节码以适应编译时数据结构与目标机器上数据结构间的差异。这个工作是基于编译对象中的 CO-RE 重定位信息完成的。有一些库就可以处理重定位问题：libbpf是最初的 C 库，包括重定位功能，Cilium eBPF 库为 Go 程序员提供了相同的功能，而 Aya 则为 Rust 提供了重定位支持。

• 可选的 BPF 框架

可以从已编译的 BPF 对象文件自动生成一个框架，其中包含用户空间代码可调用的帮助函数，也有用于管理 BPF 程序的生命周期，还可以将程序加载到内核中、附加到事件等。如果你用 C 代码编写用户空间代码，可以使用 bpftool gen skeleton 生成此框架。这些函数是更高级的抽象，对开发人员来说比直接使用底层库（如 libbpf、cilium/ebpf 等）更方便。

Andrii Nakryiko 撰写了一篇出色的博文，详细介绍了 CO-RE 的背景，以及它的工作

原理和用法。他还编写了权威的BPF CO-RE 参考指南，因此如果你计划自己编写代码，请务必阅读该指南。他的libbpf-bootstrap指南介绍了如何从头开始构建一个带有 CO-RE + libbpf + skeletons 的 eBPF 应用程序，这也是必读的内容。

既然你对 CO-RE 的要素有了了解，那么接下来就深入了解它们的工作原理，从探索 BTF 开始。

5.3 BPF 类型格式 (BTF)

BTF 信息描述了数据结构和代码在内存中的布局，这些信息有各种不同的用途。

5.3.1 BTF 用途

在 CO-RE 这一节中讨论 BTF 的主要原因是为了了解 eBPF 程序编译时结构的布局与目标机器上的布局间的差异，以便于在程序加载到内核时进行适当的调整。我将在本章后面讨论重定位过程，但现在让我们也考虑些 BTF 信息的其他用途。

了解结构的布局方式以及结构中每个字段的类型，可以将结构的内容以人类可读的形式进行打印。例如，从计算机的角度来看，字符串只是一系列字节，但将这些字节转换为字符串对人类更友好，更易理解。在前一章中，你已经看到过一个例子，bpftool 使用 BTF 信息来格式化映射转储的输出。

BTF 信息还包括行和函数信息，这样 bpftool 可以将源代码插入到翻译或 JIT 程序转储的输出中。当你阅读第 6 章时，你还将看到源代码信息与验证器日志输出交织在一起，这也源于 BTF 信息。

BTF 信息还对 BPF 自旋锁（spin locks）起着重要作用。自旋锁用于防止两个 CPU 核心同时访问同一映射值。该锁必须是映射值结构的一部分，如下所示：

```
struct my_value {
    ... <other fields>
    struct bpf_spin_lock lock;
... <other fields>
};
```

在内核中，eBPF 程序使用 bpf_spin_lock() 和 bpf_spin_unlock() 辅助函数来获取和释放锁。只有 BTF 信息中含有用来描述结构中锁的字段，才能使用这些辅助函数。

自旋锁是在内核版本 5.1 中添加的。对自旋锁的使用有许多限制：只能用于哈希或数组映射类型，并且不能在跟踪或套接字过滤类型的 eBPF 程序中使用。有关自旋锁的更多信息，请阅读 lwn.net 上有关BPF 并发管理的文章。

现在你已经了解了 BTF 信息的用途，下面通过一些示例来具体地了解 BTF。

5.3.2 使用 bpftool 列出 BTF 信息

与程序和映射一样，你可以使用 bpftool 显示 BTF 信息。以下命令列出了加载到内核中的所有 BTF 数据：

```
bpftool btf list
1: name [vmlinux] size 5843164B
2: name [aes_ce_cipher] size 407B
3: name [cryptd] size 3372B
...
149: name <anon> size 4372B prog_ids 319  map_ids 103
        pids hello-buffer-co(7660)
155: name <anon> size 37100B
        pids bpftool(7784)
```

（为简洁起见，我省略了结果中的许多条目。）

列表中的第一条是 vmlinux，它对应于我之前提到的 vmlinux 文件，该文件包含有关当前运行的内核 BTF 信息。

前面的示例复用了第 4 章的内容。后面的示例，可在github.com/lizrice/learning-ebpf的 chapter5 目录中找到。

要获取这个示例输出，我在运行第 4 章的 hello-buffer-config 示例时执行了这个命令。你可以在以 149: 开头的行上看到描述该进程正在使用的 BTF 信息的条目。

```
149: name <anon> size 4372B prog_ids 319  map_ids 103
    pids hello-buffer-co(7660)
```

这行的含义如下：

- 这个 BTF 信息块的 ID 是 149。
- 它是一个约 4 KB 大小的匿名 BTF 信息块。
- 它被 prog_id 为 319 的 BPF 程序和 map_id 为 103 的 BPF 映射使用。
- 它还被进程 ID 为 7660 的进程（括号内显示）使用，该进程正在运行名为 hello-buffer-config 的可执行文件（名称截断为了 15 个字符）。

这些程序、映射和 BTF 标识符与通过 bpftool 输出的内容相匹配：

```
bpftool prog show name hello
319: kprobe name hello tag a94092da317ac9ba gpl
    loaded_at 2022-08-28T14:13:35+0000 uid 0
    xlated 400B jited 428B memlock 4096B map_ids 103,104
    btf_id 149
    pids hello-buffer-co(7660)
```

这两组信息间似乎唯一不完全匹配的是程序中引用的额外的 map_id，即 104。这是性能事件缓冲区映射，它不使用 BTF 信息，因此在与 BTF 相关的输出中不显示。

bpftool 可以转储程序和映射的内容，它也可以用来查看包含在数据块中的 BTF 类型信息。

5.3.3 BTF 类型

使用命令 bpftool btf dump id <id>，你可以查看 BTF 信息。使用之前获取的 ID 149 运行这个命令时，我得到了 69 行输出，每行都是一个类型定义。我描述下前几行，这应该能够帮助你很好地理解如何解释其余部分。前几行的 BTF 信息与配置哈希映射相关，其源码中定义如下：

```
struct user_msg_t {
    char message[12];
};
BPF_HASH(config, u32, struct user_msg_t);
```

这个哈希表键的类型是 u32，值类型是 struct user_msg_t。该结构包含一个 12 字节的 message 字段。让我们看看这些类型在相应的 BTF 信息中是如何定义的。

BTF 输出的前三行如下：

```
[1] TYPEDEF 'u32' type_id=2
[2] TYPEDEF '__u32' type_id=3
[3] INT 'unsigned int' size=4 bits_offset=0 nr_bits=32
   encoding=(none)
```

方括号中的数字是类型 ID（因此，以 [1] 开始的第一行定义了类型 ID 1，以此类推）。让我们更详细地了解这三种类型：

- 类型 1 定义了一个名为 u32 的类型，其类型由 type_id = 2 定义，即以 [2] 开始的行定义的类型。
- 类型 2 名为 ___u32，其类型由类型 type_id = 3 定义。
- 类型 3 是一个整数类型，名为 unsigned int，长度为 4 字节。

这三种类型都是 32 位无符号整数类型的同义词。在 C 中，整数的长度是与平台相关的，因此 Linux 定义了诸如 u32 之类的类型，以显式地定义特定长度的整数。在这台机器上，u32 对应无符号整数。引用这些类型的用户空间代码应该使用带有下划线前缀的同义词，如 ___u32。

BTF 输出中的其他几种类型如下所示：

```
[4] STRUCT 'user_msg_t' size=12 vlen=1
        'message' type_id=6 bits_offset=0
[5] INT 'char' size=1 bits_offset=0 nr_bits=8 encoding=(none)
```

```
[6] ARRAY '(anon)' type_id=5 index_type_id=7 nr_elems=12
[7] INT '__ARRAY_SIZE_TYPE__' size=4 bits_offset=0 nr_bits=32
    encoding=(none)
```

这些信息与用于 config 映射值的 user_msg_t 结构相关：

- 类型 4 是 user_msg_t 结构本身，总长度为 12 字节。它包含一个名为 message 的字段，由类型 6 定义。vlen 字段表示该定义中有多少个字段。
- 类型 5 被命名为 char，是一个 1 字节整数，这正是 C 程序员要的。
- 类型 6 将 message 字段定义为具有 12 个元素的数组。每个元素的类型为 5（即 char），并且该数组由类型 7 索引。
- 类型 7 是一个 4 字节的整数。

通过这些定义，你可以完整地了解 user_msg_t 结构在内存中的布局，如图 5-1 所示。

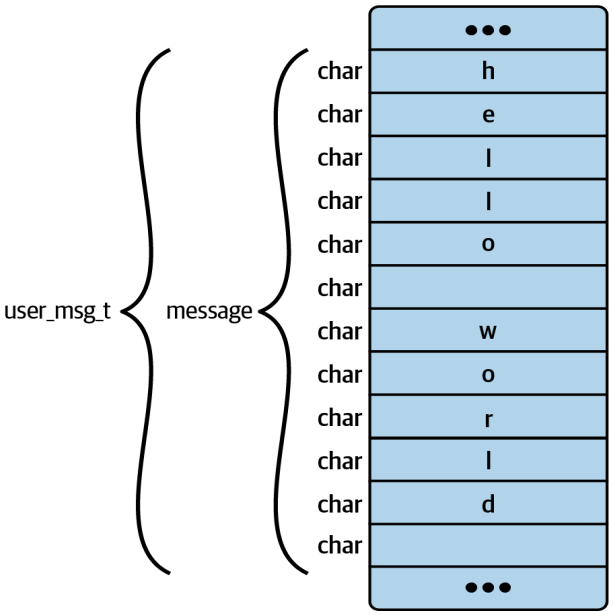

图 5.1: 一个 user_msg_t 结构占用 12 字节内存空间

到目前为止，所有数据结构的 bits_offset 都为 0，但这里有一个具有多字段的结构：

```
[8] STRUCT '____btf_map_config' size=16 vlen=2
        'key' type_id=1 bits_offset=0
        'value' type_id=4 bits_offset=32
```

这是存储在名为 config 映射中的键值对定的义。我没有自己定义这个 ____btf_map_config 类型，而是由 BCC 生成的。键的类型是 u32，值是 user_msg_t 结构，对应于类型 1 和类型 4。该结构 BTF 信息的另一个重要部分是值在结构的起始位置后 32 位开始。这是完全合理的，因为前 32 位需要用于保存键。

在 C 中，结构字段会自动对齐到内存边界，因此你不能简单地假设一个字段总是紧随前一个字段直接存储在内存中。例如，考虑这样一个结构：

```
struct something {
    char letter;
    u64 number;
}
```

在字段 letter 之后，会有 7 字节未使用的内存，以使得 64 位数字对齐到可被 8 整除的内存位置上。

某些情况下，可以打开编译器的紧密对齐选项以节省这些未使用空间，但这样干反而会导致性能下降，并且至少在我的经验中，这种做法很少见。更为常见的是，C 程序员会手动设计数据结构以便于更有效的利用内存。

5.3.4 带有 BTF 信息的映射

你现在了解了与映射相关的 BTF 信息。现在让我们看看在创建映射时如何将这些 BTF 数据传递给内核。

在第 4 章中，你见过用 bpf(BPF_MAP_CREATE) 系统调用来创建映射。它接受一个 bpf_attr 结构作为参数，在内核中定义如下（有省略）：

```
struct { /* anonymous struct used by BPF_MAP_CREATE command */
    __u32   map_type; /* one of enum bpf_map_type */
    __u32   key_size; /* size of key in bytes */
    __u32   value_size; /* size of value in bytes */
    __u32   max_entries; /* max number of entries in a map */
    ...
    char    map_name[BPF_OBJ_NAME_LEN];
    ...
    __u32   btf_fd;            /* fd pointing to a BTF type data */
    __u32   btf_key_type_id;   /* BTF type_id of the key */
    __u32   btf_value_type_id; /* BTF type_id of the value */
    ...
};
```

在引入 BTF 之前，btf_* 字段在 bpf_attr 结构中是不存在的，内核对键或值的结构一无所知。key_size 和 value_size 定义了所需内存大小，但它们只被视为字节数。通过额外传递定义键和值类型的 BTF 信息，内核可以对它们进行检查，而像 bpftool 这样的工具则利用这些类型信息来进行优雅的打印输出，正如之前讨论的那样。然而，要注意分别为键和值都传递了 BTF 类型 ID。你先前看到的 _____btf_map_config 结构在映射定义中并没有被内核使用，它只在用户空间中由 BCC 使用。

5.3.5 函数和函数原型的 BTF 数据

到目前为止，示例输出中的 BTF 数据与数据类型有关，但是 BTF 数据还包含函数和函数原型的信息。下面是描述 hello 函数的 BTF 数据块中的信息：

```
[31] FUNC_PROTO '(anon)' ret_type_id=23 vlen=1
        'ctx' type_id=10
[32] FUNC 'hello' type_id=31 linkage=static
```

在第 32 行中，你可以看到名为 hello 的函数被定义为前一行定义的类型。这是一个函数原型，它返回一个类型 ID 为 23 的值，并带有一个参数（vlen=1），名为 ctx，类型 ID 为 10。为完整起见，以下是输出中其他关联类型的定义：

```
[10] PTR '(anon)' type_id=0
[23] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
```

类型 10 是一个匿名指针，其默认类型为 0，在 BTF 输出中没有显式包含，但被定义为一个void 指针。

类型 23 的返回值是一个 4 字节整数，encoding=SIGNED 表示它是有符号整数，即它可以是正数或负数。这对应于 hello-buffer-config.py 源码中的函数定义，如下所示：

```
int hello(void *ctx)
```

到目前为止，我展示的 BTF 信息示例是通过列出 BTF 数据块内容得到的。现在让我们来看看如何获取特定于某个映射或程序的 BTF 信息。

5.3.6 检查映射和程序的 BTF 数据

如果想检查与特定映射相关的 BTF 类型，bpftool 就可以轻松完成。例如，下面是 config 的输出：

```
bpftool btf dump map name config
[1] TYPEDEF 'u32' type_id=2
[4] STRUCT 'user_msg_t' size=12 vlen=1
        'message' type_id=6 bits_offset=0
```

类似地，你可以使用 bpftool btf dump prog <prog identity> 来检查与特定程序相关的 BTF 信息。你自己可以查看man 页以获取更多详细信息。

如果想更好地理解 BTF 类型数据的生成和去重，Andrii Nakryiko 在这个主题上还有一篇出色的博文。

现在，你应该了解了 BTF 如何描述数据结构和函数的格式。用 C 编写的 eBPF 程序需要定义类型和结构的头文件。让我们看看如何轻松地为 eBPF 程序可能需要的任何内核数据类型生成头文件。

5.4 生成内核头文件

如果在启用了 BTF 的内核上运行 bpftool btf list 命令，你会看到许多预先存在的 BTF 数据块，如下所示：

```
$ bpftool btf list
1: name [vmlinux] size 5842973B
2: name [aes_ce_cipher] size 407B
3: name [cryptd] size 3372B
...
```

第一项 ID 为 1，名为 vmlinux，是正在运行的这台（虚拟）机上的内核所使用的所有数据类型、结构和函数定义的 BTF 信息。

eBPF 程序需要引用内核数据结构和类型的定义。在 CO-RE 出现之前，通常需要找出 Linux 内核源码中的多个头文件中哪个包含你感兴趣的结构定义，但现在有了更简单的方法，因为启用了 BTF 的工具可以从内核中包含的 BTF 信息生成适当的头文件。

这个头文件通常被称为 vmlinux.h，你可以使用 bpftool 生成它，像这样：

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

这个文件定义了所有内核的数据类型，因此在你的 eBPF 程序源文件中包含这个生成的 vmlinux.h 文件将提供你可能需要的任何 Linux 数据结构的定义。将源码编译成 eBPF 对象文件时，该对象中会包含此头文件中使用到的定义匹配的 BTF 信息。之后，在目标机器上运行程序时，将程序加载到内核的用户空间程序会根据这个构建时的 BTF 信息和目标机器上内核的 BTF 信息间的差异进行调整。

自从 Linux 5.4 版本起，BTF 信息以/sys/kernel/btf/vmlinux 文件的形式默认包含在 Linux 内核中，libbpf 可以用于生成旧版本内核的 BTF 数据。换句话说，如果你想在目标机器上运行一个支持 CO-RE 的 eBPF 程序，而目标机器内核又不提供 BTF 信息，那么你可以自己提供目标机器的 BTF 数据。在BTFHub上有关于如何生成 BTF 文件的信息，以及各种 Linux 发行版的存档。

BTFHub 仓库还包含有关BTF 内部的进一步阅读材料，如果你想深入了解这个主题，可以参考这些材料。

接下来，让我们看看如何使用各种策略来编写支持 CO-RE 而在不同内核间移植的 eBPF 程序。

5.5 CO-RE eBPF 程序

在本章的后面部分，我将展示一些与内核代码进行交互的用户空间代码，但在本节中，先讨论内核侧。

eBPF 程序要被编译为 eBPF 字节码，目前支持这一功能的编译器是 Clang 或 gcc，以

及 Rust 编译器 rustcc。在第 10 章中，我将讨论使用 Rust 的一些库选项，但在本章中，我假设你用的是 C 语言写代码，并使用 Clang 和 libbpf 库。

现在来考虑一个名为 hello-buffer-config.c 的示例程序。它与前一章中使用 BCC 框架的 hello-buffer-config.py 示例非常相似，但这个版本是用 C 语言编写的，使用到了 libbpf 和 CO-RE。

如果你有基于 BCC 的 eBPF 代码想迁移到 libbpf，请参考 Andrii Nakryiko 在他网站上整理出色而全面的指南。BCC 本身提供了一些便捷方式供人使用，，而 libbpf 的使用方式有所不同；libbpf 提供了一套宏和库函数，使 eBPF 程序员的工作更加简化。在下面俄示例中，我将指出 BCC 和 libbpf 之间的一些差异。

你可以在github.com/lizrice/learning-ebpf仓库的 chapter5 目录中找到本节的 C 语言 eBPF 程序。

首先，看一下 hello-buffer-config.bpf.c，这个文件实现了在内核中运行的 eBPF 程序。本章后面部分，我将展示 hello-buffer-config.c 中的用户空间代码，它加载程序并显示输出，就像第 4 章中的 BCC 实现中的 Python 代码一样。

与任何 C 程序一样，eBPF 程序需要包含头文件。

5.5.1 头文件

hello-buffer-config.bpf.c 的前几行指定了它所需的头文件：

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include "hello-buffer-config.h"
```

这五个文件分别是 vmlinux.h 文件、libbpf 中的一些头文件，以及我自己编写的一个应用程序特定的头文件。让我们看看为 libbpf 程序所需的头文件为什么是这种模式。

头文件信息

如果你编写的 eBPF 程序涉及任何内核数据结构或类型，最简单的选择是包含本章描述的 vmlinux.h 文件。另外，也可以包含 Linux 源码中的单个头文件，或者在自己的代码中手动定义这些类型，如果你真的想这样做的话。如果要使用 libbpf 的 BPF 辅助函数，你需要包含 vmlinux.h 或 linux/types.h 来获取 BPF 辅助函数源码所引用的类型（如 u32、u64 等）的定义。

vmlinux.h 文件是从内核源码头文件派生来的，但不包含那些头文件中的 #define 值。例如，如果你的 eBPF 程序解析以太网数据包，你可能需要常量定义来表明包含的协议（例如 0x0800 表示 IP 数据包，0x0806 表示 ARP 数据包等）。如果你不包含定义这些值

的if_ether.h文件，就需要在自己的代码中复制这些常量值。在 hello-buffer-config 中我不需要这些值的定义。

libbpf 中的头文件

要在你的 eBPF 代码中使用任何 BPF 辅助函数，都需要包含 libbpf 的头文件。

关于 libbpf 有一件稍微令人困惑的事情是，它不仅仅是一个用户空间库。你会发现在用户空间和 eBPF C 代码中都需要包含来自 libbpf 的头文件。

在本书撰写时，常见的做法是将 libbpf 作为子模块并从源代码进行构建/安装，这也是我在本书示例代码仓库中所做的。如果将其作为子模块包含，你只需要从 libbpf/src 目录运行 make install 命令即可。我认为不久的将来，很可能会看到 libbpf 作为常见的 Linux 发行版软件包广泛提供，特别是因为 libbpf 现已发布了1.0 版本这个重要里程碑。

应用程序特定头文件

常见的做法是用一个特定于应用程序的头文件来定义用户空间和 eBPF 部分都使用的结构。在我的示例中，hello-buffer-config.h 头文件定义了 data_t 结构，我在其中用它来将事件数据从 eBPF 程序传递到用户空间。它几乎与 BCC 版本的代码中的结构相同，如下所示：

```
struct data_t {
    int pid;
    int uid;
    char command[16];
    char message[12];
    char path[16];
};
```

与之前的版本唯一的区别是我添加了一个名为 path 的字段。

将这个结构定义放入单独的头文件中以便我在 hello-buffer-config.c 的用户空间代码中引用它。在 BCC 版本中，内核和用户空间代码都定义在单个文件中，BCC 在幕后做了一些工作，使得该结构对 Python 用户空间代码可用。

5.5.2 定义映射

在包含头文件之后，hello-buffer-config.bpf.c 源码的接下来几行定义了用于映射的结构，如下所示：

```
struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(u32));
    __uint(value_size, sizeof(u32));
```

```
} output SEC(".maps");

struct user_msg_t {
    char message[12];
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, u32);
    __type(value, struct user_msg_t);
} my_config SEC(".maps");
```

这比 BCC 示例中的代码更多。使用 BCC 时，config 映射是用以下宏建的：

```
BPF_HASH(config, u64, struct user_msg_t);
```

在没有用 BCC 的情况下，这个宏不可用，所以在 C 中你必须手动编写它。我用了 ___uint 和 ___type。它们与 ___array 一起定义在bpf/bpf_helpers_def.h中，如下所示：

```
#define __uint(name, val) int (*name)[val]
#define __type(name, val) typeof(val) *name
#define __array(name, val) typeof(val) *name[]
```

这些宏通常在基于 libbpf 的程序普遍使用，我认为它们使得映射定义更容易阅读。

在 vmlinux.h 中定义的名称 “config” 与一个定义发生了冲突，所以我在这个示例中将映射重命名为 “my_config”。

5.5.3 eBPF 程序段

在 libbpf 中使用需要用 SEC() 宏为每个 eBPF 程序标记程序类型，例如：

```
SEC("kprobe")
```

这会在编译后的 ELF 对象中生成一个名为 kprobe 的段，因此 libbpf 知道将其作为 BPF_PROG_TYPE_KPROBE 加载。我们将在第 7 章进一步讨论不同的程序类型。

根据程序类型，你还可以使用段名来指定程序将附加到的事件。libbpf 库使用此信息自动设置事件附加，而无需用户空间代码显式设置。因此，如要自动附加到 ARM 架构机器上的 execve 系统调用的 kprobe，可以将段指定如下：

```
SEC("kprobe/__arm64_sys_execve")
```

这需要你了解 CPU 体系结构上系统调用的函数名称(或通过查看目标机器上的/proc/kallsyms 文件来找，该文件列出了所有内核符号，包括其函数名称）。但是 libbpf 可以通过使用 k(ret)syscall 段名为你简化工作,它告诉加载程序自动附加到体系结构特定函数中的 kprobe 中:

```
SEC("ksyscall/execve")
```

有效的段名和格式均在libbpf 文档中列出。过去，段名的要求非常宽松，因此可能会遇到在 libbpf 1.0 之前编写的 eBPF 程序，其段名不符合新版的要求。

段定义指定了 eBPF 程序应该附加的位置，然后是程序本身。与之前一样，eBPF 程序本身是以 C 写的。在示例代码中，它被称为 hello()，与你在第 4 章中看到的 hello() 函数非常相似。让我们来看下两个版间的区别:

```
SEC("ksyscall/execve")
int BPF_KPROBE_SYSCALL(hello, const char *pathname)
{
    struct data_t data = {};
    struct user_msg_t *p;

    data.pid = bpf_get_current_pid_tgid() >> 32;
    data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

    bpf_get_current_comm(&data.command, sizeof(data.command))
       ;
    bpf_probe_read_user_str(&data.path, sizeof(data.path),
       pathname);

    p = bpf_map_lookup_elem(&my_config, &data.uid);
    if (p != 0) {
        bpf_probe_read_kernel(&data.message, sizeof(data.
           message), p->message);
    } else {
        bpf_probe_read_kernel(&data.message, sizeof(data.
           message), message);
    }

    bpf_perf_event_output(ctx, &output, BPF_F_CURRENT_CPU,
                          &data, sizeof(data));
    return 0;
}
```

我用了 libbpf 中定义的BPF_KPROBE_SYSCALL宏，它使得按名称访问系统调用的参数变得更简单。对于 execve() 来说，第一个参数是将要执行的程序的路径名。eBPF 程序的名称是 hello。

由于这个宏使得访问 execve() 的路径名参数变得非常容易，我将其包含在发送到 perf 缓冲区 output 的数据中。请注意，复制内存需要使用 BPF 辅助函数。

这里，bpf_map_lookup_elem() 是用于在映射中查找值的 BPF 辅助函数，给定一个键。BCC 的等效操作是 p = my_config.lookup(&data.uid)。在将 C 代码传递给编译器之前，BCC 会将其重写为使用底层的 bpf_map_lookup_elem() 函数。当你使用 libbpf 时，在编译之前不会对代码进行重写，所以你必须直接写入辅助函数。

还有另一个类似的例子，直接写入缓冲区的辅助函数是 bpf_perf_event_output()，而 BCC 为我提供了方便的等效操作 output.perf_submit(ctx, &data, sizeof(data))。

唯一的其他区别是在 BCC 版本中，我将 message 字符串定义为 hello() 函数内的局部变量。BCC 不支持全局变量。在 C 版本中，我将其定义为全局变量，就像这样：

```
char message[12] = "Hello World";
```

chapter4/hello-buffer-config.py 中，hello 函数的定义略有不同，如下所示：

```
int hello(void *ctx)
```

BPF_KPROBE_SYSCALL 宏是 libbpf 中方便的新增功能之一。你不必使用这个宏，但它确实简化了很多工作。它完成了为传递给系统调用的所有参数提供具有名称的参数的繁重工作。在这种情况下，它提供了一个指向包含即将运行的可执行文件路径的字符串的路径名参数，这是 execve() 系统调用的第一个参数。

你非常仔细地观察，可能会注意到在我提供的 hello-buffer-config.bpf.c 源码中，并没有明确定义 ctx 变量，尽管如此，我仍然能够在向输出 perf 缓冲区提交数据时使用它，就像这样：

```
bpf_perf_event_output(ctx, &output, BPF_F_CURRENT_CPU, &data,
    sizeof(data));
```

ctx 变量确实存在，在 libbpf 的 bpf/bpf_tracing.h 中的 BPF_KPROBE_SYSCALL 宏定义中，你也可以在那里找到一些相关的注释。使用一个没有明确定义的变量可能会令人困惑，但它的存在还是非常有帮助的，至少可以直接访问它。

5.5.4 使用 CO-RE 进行内存访问

对于性能追踪类的 eBPF 程序来说，通过 bpf_probe_read_*() 函数族的辅助函数进行内存访问是有限制的。（还有一个名为 bpf_probe_write_user() 的辅助函数，但它仅供“实验使用”）。问题是，eBPF 验证器通常不会允许你像在常规 C 代码中那样通过指针来直接读取内存（例如，x = p->y）。

libbpf 库利用 BTF 信息为 bpf_probe_read_*() 这些辅助函数提供了 CO-RE 包装，并在不同的内核版本之间实现了内存访问调用的可移植性。以下是 bpf_core_read.h 头文件中的一个包装器示例：

```
#define bpf_core_read(dst, sz, src)
    bpf_probe_read_kernel(dst, sz,
        (const void *)__builtin_preserve_access_index(src))
```

函数 bpf_core_read() 直接调用了 bpf_probe_read_kernel()，唯一的区别就在于它使用内置函数 ___builtin_preserve_access_index() 包装了 src 字段。这告诉 Clang 在访问内存地址的 eBPF 指令中生成 CO-RE 重定位信息。

___builtin_preserve_access_index() 是对“常规” C 代码的扩展，将其添加到 eBPF 中还需要对 Clang 编译器进行更改以支持它并生成这些 CO-RE 重定位项。增加的这些扩展和更改是一些 C 编译器目前还无法生成 eBPF 字节码的原因。有关 Clang 为支持 eBPF CO-RE 所需的更改的详细信息，请参阅 LLVM 邮件列表。

CO-RE 重定位项告诉 libbpf 在将 eBPF 程序加载到内核时要考虑到 BTF 差异，要重新写入地址，如果在目标内核中，src 在其包含结构中的偏移量不同，重新写入的指令将会处理。

libbpf 库提供了 BPF_CORE_READ() 宏,这样你就可以在一行中编写多个 bpf_core_read() 调用，而不需要为每个指针解引用单独调用辅助函数。例如，如果要执行类似 d = a->b->c->d 的操作，你可以这样写：

```
struct b_t *b;
struct c_t *c;

bpf_core_read(&b, 8, &a->b);
bpf_core_read(&c, 8, &b->c);
bpf_core_read(&d, 8, &c->d);
```

但使用下面这种写法更加简洁：

```
d = BPF_CORE_READ(a, b, c, d);
```

然后，你就可以使用 bpf_probe_read_kernel() 辅助函数从指针 d 中读取数据。Andrii 的指南中有对此的很好描述。

5.5.5 许可证定义

eBPF 程序必须声明其许可证。示例如下：

```
char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

现在 hello-buffer-config.bpf.c 中的所有代码都齐了，可以将其编译为一个目标文件。

5.6 为 CO-RE 编译 eBPF 程序

在第 3 章中，你看到了一个将 C 代码编译为 eBPF 字节码的 Makefile。现在，我们深入了解下使用的选项，并看看它们在 CO-RE/libbpf 程序中的必要性。

5.6.1 调试信息

必须使用-g 选项，以便编译结果中包含调试信息，这对于 BTF 是必要的。然而，-g 选项还会将 DWARF 调试信息添加到目标文件中，这对于 eBPF 程序是不需要的，因此你可以通过运行以下命令将其剥离以减小目标文件的大小：

```
llvm-strip -g <object file>
```

5.6.2 编译优化

使用-O2 优化标志（2 级或更高级）以便 Clang 产生能够通过验证器的 BPF 字节码。其中一个点就是 Clang 会默认输出 callx <register> 来调用辅助函数，但 eBPF 不支持从寄存器调用，所以加了-O2 标志会避免这个问题。

5.6.3 目标平台架构

如果使用 libbpf 定义了宏，需要在编译时指定目标架构。libbpf 头文件 bpf/bpf_tracing.h 定义了几个特定于平台的宏，例如这个示例中使用的 BPF_KPROBE 和 BPF_KPROBE_SYSCALL。BPF_KPROBE 宏可用于将 eBPF 程序附加到 kprobe，而 BPF_KPROBE_SYSCALL 是专门用于系统调用 kprobe 的变体。

kprobe 的参数是一个 pt_regs 结构体，它保存了 CPU 寄存器内容的副本。由于寄存器是与架构相关的，pt_regs 结构体的定义取决于你所运行的架构。这意味着，如果你要使用这些宏，就需要告诉编译器目标架构是什么。你可以通过设置-D __TARGET_ARCH($ARCH) 来实现，其中 $ARCH 是架构名称，如 arm64、amd64 等。

此外，请注意，如果不用宏，仍然需要针对 kprobe 使用特定于架构的代码来访问寄存器信息。也许 CO-RE 应该改成 “在每个架构上编译一次，到处运行”！

5.6.4 Makefile

以下是编译 CO-RE 对象的 Makefile 指令：

```
hello-buffer-config.bpf.o: %.o: %.c
    clang \
        -target bpf \
        -D __TARGET_ARCH_$(ARCH) \
        -I/usr/include/$(shell uname -m)-linux-gnu \
```

```
        -Wall \
        -O2 -g \
        -c $< -o $@
    llvm-strip -g $@
```

如果你用的是示例代码，应该能够在 chapter5 目录中运行 make 来构建 eBPF 目标文件 hello-buffer-config.bpf.o（以及我将在稍后介绍的用户空间可执行文件）。现在，来检查下该目标文件，看看它是否包含 BTF 信息。

5.6.5 对象文件中的 BTF 信息

BTF（BPF Type Format）的内核文档描述了 BTF 数据如何以两个部分的形式编码在 ELF 目标文件中：.BTF 部分包含数据和字符串信息，而.BTF.ext 部分是函数和行信息。你可以使用 readelf 命令查看这些部分是否存在，例如：

```
$ readelf -S hello-buffer-config.bpf.o | grep BTF
[10] .BTF             PROGBITS        0000000000000000 000002c0
[11] .rel.BTF         REL             0000000000000000 00000e50
[12] .BTF.ext         PROGBITS        0000000000000000 00000b18
[13] .rel.BTF.ext     REL             0000000000000000 00000ea0
```

使用 bpftool，也可以查看目标文件中的 BTF 数据，如下所示：

```
bpftool btf dump file hello-buffer-config.bpf.o
```

输出的格式与你在本章前面看到的从已加载的程序和映射中转储 BTF 信息时的输出格式相同。

现在，让我们看看如何利用这些 BTF 信息使程序能够在具有不同内核版本和不同数据结构的其他机器上运行。

5.7 BPF 重定位

libbpf 库将 eBPF 程序调整为与目标机器内核上的数据结构布局匹配，即使这种布局与代码编译时的内核不同。为实现这一点，libbpf 需要由 Clang 在编译过程中生成的 BPF CO-RE 重定位信息。

你可以从linux/bpf.h头文件的 struct bpf_core_relo 定义中了解更多关于重定位的工作原理：

```
struct bpf_core_relo {
    __u32 insn_off;
    __u32 type_id;
    __u32 access_str_off;
```

```
    enum bpf_core_relo_kind kind;
};
```

eBPF 程序的 CO-RE 重定位数据包含了每条需要重定位指令的一个 bpf_core_relo 结构。假设该指令将一个寄存器设置为结构体内的字段值。该指令的 bpf_core_relo 结构（由 insn_off 字段标识）编码了该结构体的 BTF 类型（type_id 字段），并指示了相对于该结构体如何访问该字段（access_str_off 字段）。

正如你刚刚看到的，内核数据结构的重定位数据是由 Clang 自动生成并编码在 ELF 目标文件中的。是 vmlinux.h 文件开头附近的以下行使得 Clang 执行了此操作：

```
#pragma clang attribute push (__attribute__((
   preserve_access_index)),
apply_to = record)
```

preserve_access_index 属性告诉 Clang 为类型定义生成 BPF CO-RE 重定位。clang attribute push 部分表示该属性应用于所有定义，直到出现 clang attribute pop，该属性出现在文件末尾。这意味着 Clang 为 vmlinux.h 中定义的所有类型生成重定位信息。

使用 bpftool 加载 BPF 程序并打开-d 标志以启用调试信息，可以看到重定位过程的输出。例如：

```
bpftool -d prog load hello.bpf.o /sys/fs/bpf/hello
```

这会生成大量的输出，与重定位相关的部分如下所示：

```
libbpf: CO-RE relocating [24] struct user_pt_regs: found
   target candidate [205]
struct user_pt_regs in [vmlinux]
libbpf: prog 'hello': relo #0: <byte_off> [24] struct
   user_pt_regs.regs[0]
(0:0:0 @ offset 0)
libbpf: prog 'hello': relo #0: matching candidate #0 <
   byte_off> [205] struct
user_pt_regs.regs[0] (0:0:0 @ offset 0)
libbpf: prog 'hello': relo #0: patched insn #1 (LDX/ST/STX)
   off 0 -> 0
```

这个例子中，你可以看到 hello 程序的 BTF 信息中的类型 ID 24 指向名为 user_pt_regs 的结构体。libbpf 库将其与内核中的一个名为 user_pt_regs 的结构体进行匹配，该结构体在 vmlinux 的 BTF 数据集中的类型 ID 为 205。实际上，因为我在同一台机器上编译和加载了该程序，类型定义是相同的，所以在这个例子中，相对于结构体开头的偏移量仍然保持不变，对指令 #1 的“修补”也不会改变它。

许多应用中，你可能不希望用户运行 bpftool 来加载 eBPF 程序。相反，你希望将此功能构建到一个专门的用户空间程序中，以可执行文件的形式提供给用户。现在来考虑下如何编写这个用户空间程序。

5.8 CO-RE 用户空间程序

在不同的编程语言中，有不同的框架支持 CO-RE，它们通过在加载 eBPF 程序到内核时实现重定位来支持 CO-RE。在本章中，我将展示使用 libbpf 的 C 代码示例；其他选择包括使用 Go 包 cilium/ebpf 和 libbpfgo，以及 Rust 的 Aya。

5.9 用户空间的 Libbpf 库

libbpf 库是一个用户空间库，如果你在 C 中编写应用程序的用户空间部分，你可以直接使用该库。如果你愿意，也可以在不使用 CO-RE 的情况下使用这个库。Andrii Nakryiko 在他的 libbpf-bootstrap博文中有一个关于如何在不使用 CO-RE 的情况下使用该库的例子。

该库提供了一些函数来封装 bpf() 和相关的系统调用，你在第 4 章中已经了解过这些系统调用，这些函数可以用于将程序加载到内核并将其附加到事件上，或者从用户空间访问映射信息。通过自动生成的 BPF 骨架代码，是使用这些抽象的传统和最简单的方式。

5.9.1 BPF 框架

你可以用 bpftool 从现有的 eBPF ELF 文件中自动生成骨架代码，例如：

```
bpftool gen skeleton hello-buffer-config.bpf.o > hello-buffer
    -config.skel.h
```

查看生成的骨架头文件，你会发现它包含了 eBPF 程序和映射的结构定义，以及一些以 hello_buffer_config_bpf___ 开头的函数（根据对象文件的名称生成）。这些函数负责管理 eBPF 程序和映射的生命周期。你不一定非要使用骨架代码，如果你愿意，可以直接调用 libbpf 中的函数，但是自动生成的代码通常会节省一些输入。

在生成的骨架文件的末尾，你会看到一个名为 hello_buffer_config_bpf___elf_bytes 的函数，它返回 ELF 对象文件 hello-buffer-config.bpf.o 的字节内容。一旦生成了骨架，我们实际上就不再需要该对象文件了。你可以通过运行 make 生成 hello-buffer-config 可执行文件，然后删除.o 文件来测试这一点；可执行文件中包含了 eBPF 字节码。

如果你愿意，也可以使用 libbpf 函数 bpf_object___open_file 从 ELF 文件中加载 eBPF 程序和映射，而不是使用骨架文件中的字节内容。

以下是使用生成的骨架代码管理示例中 eBPF 程序和映射的用户空间代码的大纲。为清晰起见，省略了一些细节和错误处理，但你可以在第 5 章的 hello-buffer-config.c 中找到完整的源代码。

```
... [other #includes]
#include "hello-buffer-config.h"      // 1
#include "hello-buffer-config.skel.h"

... [some callback functions]
int main()
{
    struct hello_buffer_config_bpf *skel;
    struct perf_buffer *pb = NULL;
    int err;
    libbpf_set_print(libbpf_print_fn); // 2
    skel = hello_buffer_config_bpf__open_and_load(); // 3
...
    err = hello_buffer_config_bpf__attach(skel); // 4
...
    pb = perf_buffer__new(bpf_map__fd(skel->maps.output), 8,
       handle_event, lost_event, NULL, NULL); // 5
...
    while (true) {                    // 6
        err = perf_buffer__poll(pb, 100);
... }

    perf_buffer__free(pb);            // 7
    hello_buffer_config_bpf__destroy(skel);
    return -err;
}
```

1. 该文件包含了自动生成的骨架头文件，以及我手动编写的用于在用户空间和内核代码之间共享的头文件。

2. 这段代码设置了一个回调函数，用于打印由 libbpf 生成的日志信息。

3. 创建了一个表示在 ELF 文件中定义的所有映射和程序的 skel 结构，并将它们加载到内核中。

4. 程序会自动附加到相应的事件上。

5. 这个函数创建了一个用于处理 perf 缓冲输出的结构。

6. 在这里，不断轮询 perf 缓冲区。

7. 这是清理代码。

让我们详细了解其中的一些步骤。

加载程序和映射到内核

对自动生成函数的第一个调用是这样的：

```
skel = hello_buffer_config_bpf__open_and_load();
```

顾名思义，这个函数包含了两个阶段：打开和加载。”打开”阶段涉及读取 ELF 数据并将其转换为表示 eBPF 程序和映射的结构。”加载”阶段将这些映射和程序加载到内核中，并根据需要执行 CO-RE 重定位修复。

这两个阶段可以分开处理，因为骨架代码提供了单独的 name___open() 和 name___load() 函数。这样你就有可以在加载之前操作 eBPF 信息。通常在加载程序之前进行配置是常见的做法。例如，可以将计数器全局变量 c 初始化为某个值，就像这样：

```
skel = hello_buffer_config_bpf__open();
if (!skel) {
    // Error ...
}
skel->data->c = 10;
err = hello_buffer_config_bpf__load(skel);
```

hello_buffer_config_bpf___open() 和 hello_buffer_config_bpf___load() 返回的数据类型是一个名为 hello_buffer_config_bpf 的结构体，该结构体在骨架头文件中定义，包含了对象文件中定义的所有映射、程序和数据的信息。

骨架对象（在本例中为 hello_buffer_config_bpf）只是来自 ELF 文件的用户空间表示。一旦加载到内核中，即便你更改了对象中的值，也不会对内核里加载的数据产生影响。因此，在加载后更改 skel->data->c 的值不会产生任何效果。

访问映射

默认情况下，libbpf 还会创建在 ELF 文件中定义的映射，但有时你可能希望编写一个重用现有映射的 eBPF 程序。在前一章中，你已经看到了一个示例，其中 bpftool 迭代所有映射，寻找与指定名称匹配的映射。使用映射的另一个常见原因是在两个不同的 eBPF 程序间共享信息，因此只有一个程序应该创建映射。bpf_map___set_autocreate() 函数允许你覆盖 libbpf 的自动创建行为。

那么如何访问现有映射呢?映射可以被固定，如果你知道路径，就可以使用 bpf_obj_get() 获得现有映射的文件描述符。这里有一个非常简单的示例（在 chapter5/find-map.c 中）：

```
struct bpf_map_info info = {};
unsigned int len = sizeof(info);

int findme = bpf_obj_get("/sys/fs/bpf/findme");
if (findme <= 0) {
```

```
    printf("No FD\n");
} else {
    bpf_obj_get_info_by_fd(findme, &info, &len);
    printf("Name: %s\n", info.name);
}
```

你可以使用 bpftool 创建一个映射来尝试这个示例，就像这样：

```
$ bpftool map create /sys/fs/bpf/findme type array key 4
   value 32 entries 4
name findme
```

执行 find-map 会输出如下结果:

```
Name: findme
```

让我们回到 hello-buffer-config 示例和骨架代码。

附加到事件

示例中的下一个骨架函数将程序附加到 execve 系统调用函数上：

```
err = hello_buffer_config_bpf__attach(skel);
```

libbpf 库会自动从 SEC() 定义中获取附加点信息，如果你没有定义附加点，那么也可以使用一系列的 libbpf 函数，例如 bpf_program___attach_kprobe，bpf_program___attach_xdp 等来附加不同类型的程序。

管理事件缓冲区

设置 perf 缓冲区使用了 libbpf 自身定义的一个函数，而不是骨架中的函数：

```
pb = perf_buffer__new(bpf_map__fd(skel->maps.output), 8,
   handle_event, lost_event, NULL, NULL);
```

你可以看到 perf_buffer___new() 函数将 “output” 映射的文件描述符作为第一个参数。handle_event 参数是一个回调函数，在 perf 缓冲区中有新数据到达时调用，而 lost_event 在 perf 缓冲区中没有足够的空间供内核写入数据条目时调用。在示例中，这些函数只是向屏幕输出消息。

最后，程序必须轮询 perf 缓冲区：

```
while (true) {
    err = perf_buffer__poll(pb, 100);
    ...
}
```

100 是超时时间，单位为毫秒。之前设置的回调函数将在数据到达或缓冲区已满时调用。

最后，为了清理数据，我释放了 perf 缓冲区，并销毁了内核中的 eBPF 程序和映射，代码如下：

```
perf_buffer__free(pb);
hello_buffer_config_bpf__destroy(skel);
```

libbpf 中还有一整套与 perf_buffer_ 和 ring_buffer_ 相关的函数，可帮助你管理事件缓冲区。

如果你编译并运行这个示例程序 hello-buffer-config，你将看到以下输出（与第 4 章中看到的非常相似）：

```
23664 501 bash      Hello World
23665 501 bash      Hello World
23667 0   cron      Hello World
23668 0   sh        Hello World
```

5.9.2 Libbpf 代码示例

有许多优秀的基于 libbpf 的 eBPF 程序示例可供参考和借鉴，包括：

- libbpf-bootstrap项目，旨在提供一组示例程序帮助你入门。
- BCC 项目已将许多原始的基于 BCC 的工具迁移到 libbpf 版本中，你可以在libbpf-tools 目录中找到它们。

5.10 总结

CO-RE 使得 eBPF 程序能够在与其构建环境不同的内核上运行，这极大地提高了 eBPF 的移植性，使得工具开发人员能够向用户和客户提供可用于生产环境的工具。

在本章中，你了解了 CO-RE 是如何通过将类型信息编码到编译的目标文件中，并在加载到内核时使用重定位来重写指令，从而实现这一目标的。你还初步了解了使用 libbpf 编写 C 代码的方法：包括在内核中运行的 eBPF 程序和管理这些程序的用户空间程序，这些程序基于自动生成的 BPF 骨架代码。

5.11 练习

以下是一些可以进一步探索 BTF、CO-RE 和 libbpf 的练习：

1. 尝试使用 bpftool btf dump map 和 bpftool btf dump prog 命令查看与映射和程序相关的 BTF 信息。请记住，你可以多种方式指定单个映射和程序。

2. 比较相同程序在 ELF 目标文件形式和加载到内核后使用 bpftool btf dump file 和 bpftool btf dump prog 输出的结果。它们应该是相同的。

3. 检查使用 bpftool -d prog load hello-buffer-config.bpf.o /sys/fs/bpf/hello 命令输出的调试信息。你将看到每个部分的加载过程，许可证的检查、重定位以及描述每个 BPF 程序指令的输出。

4. 尝试使用来自 BTFHub 的不同 vmlinux 头文件构建一个 BPF 程序，并在 bpftool 的调试输出中查看更改偏移量的重定位。

5. 修改 hello-buffer-config.c 程序，使其可以使用映射为不同的用户 ID 配置不同的消息（类似于第 4 章中的 hello-buffer-config.py）。

6. 尝试更改 SEC() 中的段名称，可以使用你自己的名称。当你尝试将程序加载到内核时，你应该会收到错误，因为 libbpf 无法识别段名称。这说明了 libbpf 如何使用段名称来确定这是何种类型的 BPF 程序。你可以尝试编写自己的附加代码，显式地附加到你选择的事件上，而不是依赖 libbpf 的自动附加功能。

第六章　eBPF 验证器

前面章节已经多次提到过验证这一步骤，即 eBPF 程序加载到内核时，验证过程会确保程序的安全性。本章中，我们将深入了解验证器是如何工作以实现这个目标的。

验证涉及到检查程序每条可执行路径，并确保每条指令的安全性。验证器还会对字节码进行一些更新，以准备执行。在本章中，我将展示一些验证失败的示例，通过从有效的示例开始，对代码逐步进行修改，直到验证器验证失败。

本章示例代码位于github.com/lizrice/learning-ebpf的 chapter6 目录中。

本章并不试图涵盖验证器得所有检查，而是在提供一个概览，通过说明性示例来辅助读者处理自己写 eBPF 代码时可能遇到的验证错误。

要记住的一点是，验证器是在 eBPF 字节码上工作，而不是直接在源码上工作。字节码依赖于编译器的输出。由于诸如编译器优化之类的因素，源码的更改在字节码中可能并不完全符合预期，因此字节码中得验证器可能无法得出你期望的结果。例如，验证器会拒绝无法到达的指令，但编译器在验证步骤前就可能将其优化掉了。

6.1　验证过程

验证器会分析程序以评估所有可能的执行路径。它按顺序逐步检查指令，对其进行评估 (不执行)。在验证过程中，它通过一个称为 bpf_reg_state 的结构来跟踪每个寄存器的状态（eBPF 虚拟机中的寄存器）。该结构包括一个名为 bpf_reg_type 的字段，用于描述寄存器中保存的值类型，包括以下几种：

- NOT_INIT，表示寄存器尚未被设置为一个值。
- SCALAR_VALUE，表示寄存器已设置为一个不表示指针的值。
- 几种 PTR_TO_* 类型，表示寄存器保存了指向某个值的指针。这个值可以是如下几种：

–PTR_TO_CTX：寄存器保存了指向传递给 BPF 程序的上下文的指针。

–PTR_TO_PACKET：寄存器指向一个网络数据包（在内核中以 skb->data 形式保存）。

–PTR_TO_MAP_KEY 或 PTR_TO_MAP_VALUE：寄存器指向映射键或值。

还有其他几种 PTR_TO_* 类型，你可以在linux/bpf.h头文件中找到这些类型完整的枚举集合。

bpf_reg_state 结构还跟踪寄存器可能持有值的范围，验证器会使用这些信息来确定是否正在尝试执行无效操作。

每当验证器遇到一个分支都需要根据是否继续按顺序执行或跳转到不同的指令进行决策，验证器会将当前所有寄存器的状态复制到栈上，并探索其中一个可能的执行路径。过程中它继续评估指令，直到到达程序末尾的返回指令（或达到它将处理的指令数量限制，目前为 100 万条指令），此时它会从栈中弹出一个分支数据以进行下一步的评估。如果它发现可能导致无效操作的指令，则验证失败。

验证每个可能性得代价非常高，因此实际上有一种称为状态修剪的优化方法，它避免再评估程序中等效的执行路径。处理程序时，验证器记录程序内部某些指令处所有寄存器的状态。如果它稍后以匹配状态的寄存器再次到达同一指令，就无需继续验证该路径的剩余部分，因为先前得验证已经证明是有效得。

目前，有不少开发工作正投入到验证器优化及其修剪过程中。验证器以前会在每个跳转指令之前和之后存储修剪状态，但分析表明，平均每四条指令存储一次状态，其中绝大多数修剪状态永远不会匹配。事实证明，无论是否存在分支，每 10 条指令存储一次修剪状态更高效。

你可以在内核文档中阅读有关验证器工作原理的更多信息。

6.2 验证日志

程序验证失败，验证器会生成一份日志，显示其判断出程序无效的原因。如果你使用 bpftool prog load 命令，验证器日志会输出到 stderr。使用 libbpf 编写程序时，可以用 libbpf_set_print() 函数设置一个处理程序，该处理程序将显示（或对错误进行其他操作）日志信息。

如果真想深入了解验证器工作原理，可以让它在验证成功和失败时都生成日志。hello-verifier.c 文件中有一个示例，涉及将保存验证器日志内容的缓冲区传递给将程序加载到内核的 libbpf 调用，以便将日志写到屏幕。

验证器日志包括验证器工作量摘要，大致如下所示：

```
processed 61 insns (limit 1000000) max_states_per_insn 0
   total_states 4
peak_states 4 mark_read 3
```

这里，验证器处理了 61 条指令，包括通过不同路径到达相同指令时可能多次处理该指令。请注意，一百万条值令的限制是程序中指令数量的上限；如果代码中存在分支，验证器会处理某些指令多次。存储的状态总数为 4，在这个简单的程序中与存储状态的峰值数相匹配。如果其中一些状态被修剪，峰值数可能低于总数。

日志输出包括验证器分析的 BPF 指令，以及相应的 C 代码（如果使用-g 标志构建了包含调试信息的目标文件）和验证器状态信息的摘要。以下是验证器日志的示例摘录，与 hello-verifier.bpf.c 程序的前几行相关：

```
0: (bf) r6 = r1
; data.counter = c;

1: (18) r1 = 0xffff800008178000
3: (61) r2 = *(u32 *)(r1 +0)

 R1_w=map_value(id=0,off=0,ks=4,vs=16,imm=0) R6_w=ctx(id=0,
    off=0,imm=0)
 R10=fp0
; c++;

4: (bf) r3 = r2
5: (07) r3 += 1
6: (63) *(u32 *)(r1 +0) = r3
 R1_w=map_value(id=0,off=0,ks=4,vs=16,imm=0) R2_w=inv(id=1,
    umax_value=4294967295,

 var_off=(0x0; 0xffffffff)) R3_w=inv(id=0,umin_value=1,
    umax_value=4294967296,

 var_off=(0x0; 0x1fffffff)) R6_w=ctx(id=0,off=0,imm=0) R10=
    fp0
```

1. 因为在编译过程中使用了-g 标志，所以日志中包含源码行，以便更好地理解输出与源码的关系。

2. 这是日志中输出的一些寄存器状态信息。它告诉我们，当前阶段，寄存器 1 包含一个映射值，寄存器 6 保存上下文，寄存器 10 是帧（或栈）指针，用于保存局部变量。

3. 这是另一个寄存器状态信息的示例。你可以看到每个（初始化的）寄存器中保存的值类型，以及寄存器 2 和 3 中值的范围。

寄存器 6 保存上下文，在验证器日志中以 R6_w=ctx(id=0,off=0,imm=0) 表示。这是在字节码的第一行设置的，其中将寄存器 1 复制到寄存器 6。当调用 eBPF 程序时，寄存器 1 始终保存传递给程序的上下文参数。为什么要将它复制到寄存器 6 呢？当调用 BPF 助手函数时，该调用的参数要通过寄存器 1 到寄存器 5 传递。助手函数不会修改寄存器 6 到 9 的内容，因此将上下文保存到寄存器 6 意味着代码在调用助手函数时不会丢失上下文信息。

寄存器 0 用于保存助手函数的返回值，也用于保存 eBPF 程序的返回值。寄存器 10 始终保存指向 eBPF 栈帧的指针（eBPF 程序无法修改它）。

第 6 条指令后寄存器 2 和寄存器 3 的状态信息如下：

```
R2_w=inv(id=1,umax_value=4294967295,var_off=(0x0; 0xffffffff)
    )
R3_w=inv(id=0,umin_value=1,umax_value=4294967296,var_off=(0x0
    ; 0x1ffffffff))
```

寄存器 2 没有最小值，在这里显示的 umax_value 是十进制的 0xFFFFFFFF，对应于可以保存在 8 字节寄存器中的最大值。换句话说，寄存器可以保存其可能范围内的任何值。

在第 4 条指令中，将寄存器 2 的内容复制到寄存器 3，然后第 5 条指令将其值加 1。因此，寄存器 3 可以具有大于或等于 1 的任何值。其中 umin_value 设置为 1，umax_value 设置为 0xFFFFFFFF。

验证器使用每个寄存器状态及其可能值的范围信息来确定程序的可能路径。之前提到的状态修剪也是利用的这些信息。如果验证器在相同的代码位置，具有相同类型和每个寄存器的可能值范围时，就没有必要进一步评估此路径了。此外，如果当前状态是先前看到的状态的子集，也可以进行修剪。

6.3 可视化控制流

验证器会探索 eBPF 程序的所有可能路径，如果你要调试问题，了解这些路径会很有辅助。bpftool 工具可以通过以DOT 格式生成程序的控制流图，然后你可以将其转换为图像格式，如下所示：

```
$ bpftool prog dump xlated name kprobe_exec visual > out.dot
$ dot -Tpng out.dot > out.png
```

这将生成一个控制流的可视化表示，类似于图 6-1 所示的形式。

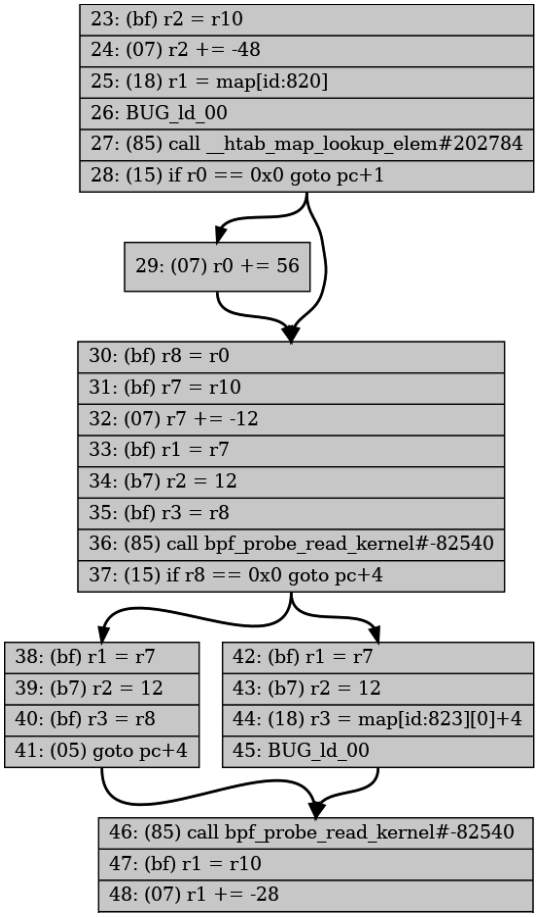

图 6.1: 从控制流图中提取的片段（完整图像 chapter6/kprobe_exec.png）。

6.4 验证辅助函数

在 eBPF 程序中，直接调用内核函数是不允许的（除非它已经被注册为 kfunc）。但是 eBPF 提供了许多辅助 (辅助) 函数，便于程序从内核中获取信息。有一个bpf-helpers 手册对辅助函数进行了全面地记录。

辅助函数要和对应的 BPF 程序类型相匹配才是有效的。例如，辅助函数 bpf_get_current_pid_tgid 检索当前用户空间的进程 ID 和线程 ID，但是从网络接口接收数据包触发的 XDP 程序中调用它是没有意义的，因为这里没有用户空间进程。你可以通过将 hello-verifier.bpf.c 中的 hello eBPF 程序的 SEC() 定义从 kprobe 更改为 xdp 来检查这一点。在加载此程序时，验证器会输出以下消息：

```
...
16: (85) call bpf_get_current_pid_tgid#14
unknown func bpf_get_current_pid_tgid#14
```

未知的 func 并不意味着该函数完全未知，只是在此 BPF 程序类型中未知。（BPF 程序类型是下一章的主题，现在你可以仅仅将其看作是适用于不同类型事件的程序。）

6.5 辅助函数参数

如果你查看kernel/bpf/helpers.c文件或类似文件，你会发现每个辅助函数都有一个类似于 bpf_map_lookup_elem() 的 bpf_func_proto 结构体，如下所示：

```
const struct bpf_func_proto bpf_map_lookup_elem_proto = {
    .func       = bpf_map_lookup_elem,
    .gpl_only   = false,
    .pkt_access = true,
    .ret_type   = RET_PTR_TO_MAP_VALUE_OR_NULL,
    .arg1_type  = ARG_CONST_MAP_PTR,
    .arg2_type  = ARG_PTR_TO_MAP_KEY,
};
```

这个结构体定义了辅助函数的参数和返回值的约束。由于验证器跟踪每个寄存器中保存的值类型，它可以发现你试图向辅助函数传递的错误类型参数。例如，向下面这样传入错误的参数来调用 bpf_map_lookup_elem()：

```
p = bpf_map_lookup_elem(&data, &uid);
```

这里不再传递 &my_config（指向映射的指针），而是传递了 &data（指向本地变量结构体的指针）。从编译器的角度来看，这是有效的，因此你可以构建出 BPF 对象文件 hello-verifier.bpf.o。但当你将程序加载到内核时，在验证日志中会看到以下错误消息：

```
27: (85) call bpf_map_lookup_elem#1
R1 type=fp expected=map_ptr
```

这里，fp 代表 (栈) 帧指针，它是存储本地变量的堆栈内存区域。寄存器 1 加载了名为 data 的本地变量的地址，但该函数期望的是指向映射的指针（如前面所示的 bpf_func_proto

结构体中的 arg1_type 字段）。通过跟踪每个寄存器中存储的值的类型，验证器能够发现错误。

6.6 检查许可证

如果你用的是受 GPL 许可的 BPF 辅助函数，eBPF 程序也必须具有兼容 GPL 的许可证。在第 6 章示例代码 hello-verifier.bpf.c 的最后一行定义了"license" 部分，其中包含字符串"Dual BSD/GPL"。如果删除此行，验证器的输出将包含以下信息：

```
...
37: (85) call bpf_probe_read_kernel#113
cannot call GPL-restricted function from non-GPL compatible program
```

这是因为 bpf_probe_read_kernel() 辅助函数的 gpl_only 字段被设置为 true，而此程序中又调用了其他辅助函数，这些辅助函数不受 GPL 许可的限制，因此验证器不会对其使用提出异议。

BCC 项目维护着一个辅助函数列表，指示它们是否受 GPL 许可。如果你对辅助函数的实现方式感兴趣，可以在BPF 和 XDP 参考指南中的相应部分找到更多信息。

6.7 检查内存访问

验证器执行了许多检查，以确保 BPF 程序只访问它们被授权访问的内存。

例如，在处理网络数据包时，XDP 程序只允许访问构成该网络数据包的内存位置。大多数 XDP 程序的开头非常类似于以下内容

```
SEC("xdp")
int xdp_load_balancer(struct xdp_md *ctx)
{
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
...
```

作为程序上下文传递给 XDP 程序的 xdp_md 结构描述了接收到的网络数据包。该结构中的 ctx->data 字段是数据包起始的内存位置，ctx->data_end 是数据包的最后一个位置。验证器将确保程序不会超出这些边界。例如，hello_verifier.bpf.c 中以下程序是有效的：

```
SEC("xdp")
int xdp_hello(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    bpf_printk("%x", data_end);
```

```
    return XDP_PASS;
}
```

变量 data 和 data_end 非常相似，但验证器足够聪明，能够识别 data_end 与数据包的结束位置相关。你的程序需要检查从数据包中读取的任何值是否不超出该位置，并且不允许你通过修改 data_end 值来” 作弊”。尝试在 bpf_printk() 调用之前添加以下行：

```
data_end++;
```

验证器会报错，如下所示：

```
; data_end++;
1: (07) r3 += 1
R3 pointer arithmetic on pkt_end prohibited
```

访问数组时，需要确保不会超出数组边界。在示例代码中，有从 message 数组中读取一个字符的示例：

```
if (c < sizeof(message)) {
    char a = message[c];
    bpf_printk("%c", a);
}
```

这里的代码是正确的，因为明确地检查来大小，确保计数变量 c 不超过 message 数组的大小。如果出现以下这种多计算一个数据的错误，代码将是无效的。

```
if (c <= sizeof(message)) {
    char a = message[c];
    bpf_printk("%c", a);
}
```

代码验证失败，并输出类似以下的错误消息：

```
invalid access to map value, value_size=16 off=16 size=1
R2 max value is outside of the allowed memory range
```

从这条消息中可以很清楚地看出，存在对映射值的无效访问，因为寄存器 2 可能包含一个超出映射索引范围的值。如果你要调试此错误，会希望深入查看日志以查明哪行代码有问题。日志在发出错误消息之前以如下形式结束：

```
; if (c <= sizeof(message)) {
30: (25) if r1 > 0xc goto pc+10
 R0_w=map_value_or_null(id=2,off=0,ks=4,vs=12,imm=0) R1_w=inv
    (id=0,
```

```
 umax_value=12,var_off=(0x0; 0xf)) R6=ctx(id=0,off=0,imm=0)
    ...
; char a = message[c];
31: (18) r2 = 0xffff800008e00004
33: (0f) r2 += r1
last_idx 33 first_idx 19
regs=2 stack=0 before 31: (18) r2 = 0xffff800008e00004
regs=2 stack=0 before 30: (25) if r1 > 0xc goto pc+10
regs=2 stack=0 before 29: (61) r1 = *(u32 *)(r8 +0)
34: (71) r3 = *(u8 *)(r2 +0)
 R0_w=map_value_or_null(id=2,off=0,ks=4,vs=12,imm=0) R1_w=
    invP(id=0,
 umax_value=12,var_off=(0x0; 0xf)) R2_w=map_value(id=0,off=4,
    ks=4,vs=16,
 umax_value=12,var_off=(0x0; 0xf),s32_max_value=15,
    u32_max_value=15)
 R6=ctx(id=0,off=0,imm=0) ...
```

1. 从错误中逆推，最后一个寄存器状态信息显示寄存器 2 的最大值为 12。

2. 在第 31 条指令中，寄存器 2 被设置为内存中的一个地址，然后逐渐递增其值与寄存器 1 的值相加。输出显示对应于访问 message[c] 的代码行，因此可以推断寄存器 2 被设置为指向 message 数组，然后通过寄存器 1 中的值 c 递增。

3. 进一步追溯寄存器 1 的值，日志显示它的最大值为 12（十六进制为 0x0c）。然而，message 被定义为一个包含 12 个字节的字符数组，所以只有索引 0 到 11 在其范围内。因此，可以看出错误源于源代码中对 c <= sizeof(message) 的测试。

在第 2 步，我从验证器日志中包含的源码推断出了一些寄存器与它们所表示的源码变量间的关系。你可以通过查看验证器日志来验证这一点，如果代码是不带调试信息编译的，那么确实可能需要这样做。鉴于调试信息存在，可以直接使用调试信息。

message 数组声明为全局变量，你可能还记得在第 3 章中讨论过全局变量是如何使用映射实现的。这就解释了为什么错误消息提到了“对映射值的无效访问”。

6.8 指针解引用前先检查

在 C 程序中，一个让程序崩溃的简单方法就是在指针的值为零（也称为 null）时对指针解引用。指针表示值在内存中的存储位置，而零不是一个有效的内存位置。eBPF 验证器要求在解引用指针前对其进行检查，以避免这种类型的崩溃。

在 hello-verifier.bpf.c 示例中，以下代码行用于查找可能存在于 my_config 哈希表映射中的自定义消息：

```
p = bpf_map_lookup_elem(&my_config, &uid);
```

如果该映射中没有与 uid 对应的条目，p（指向消息结构 msg_t 的指针）将设为零。下面是一小段代码，试图对这个可能为空的指针进行解引用：

```
char a = p->message[0];
bpf_printk("%c", a);
```

这段代码编译能通过，但验证器会拒绝它，并给出以下错误信息：

```
; p = bpf_map_lookup_elem(&my_config, &uid);
25: (18) r1 = 0xffff263ec2fe5000
27: (85) call bpf_map_lookup_elem#1
28: (bf) r7 = r0
; char a = p->message[0];
29: (71) r3 = *(u8 *)(r7 +0)
R7 invalid mem access 'map_value_or_null'
```

1. 辅助函数调用的返回值存储在寄存器 0 中。在这里，该值存储在寄存器 7 中。这意味着寄存器 7 现在保存了局部变量 p 的值。

2. 这条指令试图对指针值 p 进行解引用。验证器一直追踪寄存器 7 的状态，并知道它可能保存指向映射值的指针，或者是 null。

验证器拒绝了对空指针的解引用尝试，但如果你显式地检查了，程序验证是可以通过的：

```
if (p != 0) {
    char a = p->message[0];
    bpf_printk("%d", cc);
}
```

一些辅助函数已经内置了指针检查。例如，如果查看 bpf-helpers 的 man 页，你会发现 bpf_probe_read_kernel() 的函数签名如下：

```
long bpf_probe_read_kernel(void *dst, u32 size, const void *
   unsafe_ptr)
```

这个函数的第三个参数称为 unsafe_ptr。这是一个辅助函数的示例，它通过检查来保证代码安全。你可以传递一个可能为 null 的指针，但只能放到 unsafe_ptr 参数。辅助函数在尝试解引用之前会检查这个参数是否为 null。

6.9 访问上下文

每个 eBPF 程序都可以作为参数传递一些上下文信息，但根据程序和附加类型的不同，它可能只能访问其中一些上下文信息。例如，跟踪点程序接收一些跟踪点数据的指针。数据的格式取决于特定的跟踪点，但它们都以一些共同的字段开头，然而这些共同的字段对于 eBPF 程序来说是不可访问的。只有特定于跟踪点的字段可以访问。尝试读取或写入错误的字段会导致无效的 bpf_context 访问错误。

6.10 运行至完成

验证器要确保 eBPF 程序能够运行完；否则，存在无限消耗资源的风险。为此，通过对总指令数设定一个限制来避免这种风险。就像之前提到的，限制一百万条指令。这个限制是硬编码在内核中的，不是可配置项。如果验证器在处理了这么多指令还没有到达 BPF 程序的结尾，它就会拒绝该程序，验证失败。

创建一个永远不能完成的程序的方法是编写一个循环，那么就来看看如何在 eBPF 程序中创建循环。

6.11 循环

直到内核5.3 版本为止，循环都存在限制。通过相同的指令进行循环遍历需要向后跳转到较早的指令，而验证器过去不允许这样做。eBPF 程序员可以用 #pragma unroll 编译器指令来绕过这个限制，告诉编译器为每次循环生成一组相同（或非常相似）的字节码指令。这样就可以减少程序员编写重复代码，当然，生成的字节码中是有重复指令的。

Linux 内核从 5.3 版本开始，验证器在检查所有可能的执行路径时都会向前和向后跟踪分支。这意味着它可以接受一些循环，只要执行路径保持在一百万条指令范围内。

在 xdp_hello 程序中可以看到一个循环的示例。通过验证的循环版本如下所示：

```
for (int i=0; i < 10; i++) {
    bpf_printk("Looping %d", i);
}
```

验证器日志将显示它已经按照循环执行路径循环了 10 次。在此过程中，没有达到一百万条指令的限制。在本章的练习中，还有另一个版本的循环，它将达到该限制并且无法通过验证。

Linux 内核在 5.17 版本中引入了一个新的辅助函数 bpf_loop()，它使得验证器不仅能够接受循环，而且能够以更高效的方式实现。这个辅助函数将最大迭代次数作为第一个参数，并且还传递一个在迭代时调用的函数。验证器只需要验证该函数中的 BPF 指令一次，无论它被调用多少次。该函数可以返回非零值以指示无需再次调用它（用于在达到期望结果后提前终止循环）。

还有一个辅助函数bpf_for_each_map_elem()，它是为映射中的每个项提供的回调函数。

6.12 检查返回代码

eBPF 程序的返回码存储在寄存器 0（R0）中。如果程序未对 R0 进行初始化，验证会失败，例如：

```
R0 !read_ok
```

你可以通过将函数中的所有代码注释掉来尝试此操作；例如，将 xdp_hello 示例修改为以下内容：

```
SEC("xdp")
int xdp_hello(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    // bpf_printk("%x", data_end);
    // return XDP_PASS;
}
```

这会导致验证失败。然而，如果你将包含辅助函数 bpf_printf() 的行重新加入，验证不会失败，即使源码中没有显式设置返回值！

这是因为寄存器 0 还可用于保存辅助函数的返回值。在 eBPF 程序中从辅助函数返回后，寄存器 0 不再处于未初始化状态，所以验证成功。

6.13 无效指令

正如在第 3 章中关于 eBPF 虚拟机的讨论那样，eBPF 程序由一组字节码指令组成。验证器检查程序中的指令是否为有效的字节码指令（例如，仅使用已知的操作码）。

如果编译器生成了无效的字节码，那将被视为编译错误，因此除非你选择手动编写 eBPF 字节码，否则不太可能遇到这种类型的验证错误。然而，最近还添加了一些指令，例如原子操作。如果你的编译字节码使用了这些指令，在较旧的内核上将无法通过验证。

6.14 不可达指令

验证器还会拒绝具有不可达指令的程序。通常情况下，这些指令在编译器优化时会被消除掉。

6.15 总结

要写出通过验证器验证的代码似乎是一门魔法，最开始看似有效的代码可能会被拒绝，并抛出看似任意的错误。随着时间的推移，验证器进行了许多改进优化，在本章中，你已经看到了几个示例，其中验证器日志给出了一些提示，辅助你找出问题所在。

如果你了解 eBPF 虚拟机的原理，那么这些提示会更有帮助。该机器使用一组寄存器来作为临时存储器，在执行 eBPF 程序时逐步进行。验证器跟踪每个寄存器的类型和值范围，以确保 eBPF 程序的安全运行。

如果你尝试自己编写 eBPF 程序，可能需要在遇到验证错误时有人帮助你解决。eBPF 社区的 Slack 频道就是寻求帮助的好地方，当然，很多人在StackOverflow上也找到了解决办法。

6.16 练习

这里有一些导致验证器错误的方法，看看你是否能将验证器日志输出与错误进行对应：

1. 在“检查内存访问”那部分你看到了验证器拒绝超出全局消息数组末尾的访问。在示例代码中还有一段类似的代码，用类似的方式访问局部变量 data.message ：

```
if (c < sizeof(data.message)) {
    char a = data.message[c];
    bpf_printk("%c", a);
}
```

尝试调整代码，使其出现相同的偏移一位的错误（将 < 替换为 <=），你会看到从栈 R2 读取无效变量偏移量的错误信息。

2. 在示例代码 xdp_hello 中找到被注释掉的循环。尝试添加以下循环：

```
for (int i=0; i < 10; i++) {
    bpf_printk("Looping %d", i);
}
```

你会在验证器日志中看到一系列重复的行，类似于：

```
42: (18) r1 = 0xffff800008e10009
44: (b7) r2 = 11
45: (b7) r3 = 8
46: (85) call bpf_trace_printk#6
 R0=inv(id=0) R1_w=map_value(id=0,off=9,ks=4,vs=26,imm=0)
    R2_w=inv11
 R3_w=inv8 R6=pkt_end(id=0,off=0,imm=0) R7=pkt(id=0,off=0,r
    =0,imm=0)
```

```
 R10=fp0
last_idx 46 first_idx 42
regs=4 stack=0 before 45: (b7) r3 = 8
regs=4 stack=0 before 44: (b7) r2 = 11
```

根据日志，就可以确定追踪循环变量 i 的寄存器是哪一个。

3. 现在尝试添加一个会验证失败的循环，代码如下：

```
for (int i=0; i < c; i++) {
    bpf_printk("Looping %d", i);
}
```

你会发现验证器试图完整地遍历这个循环，但在完成之前达到了指令上限 100 万（因为全局变量 c 没有上界）。

4. 编写一个连接到追踪点的程序。预览一下“追踪点”一节，你可以看到上下文参数的结构定义以如下字段开始：

```
unsigned short common_type;
unsigned char common_flags;
unsigned char common_preempt_count;
int common_pid;
```

请自定义创建一个类似的结构，并将程序中的上下文参数声明为指向此结构的指针。在程序中，尝试访问这些字段，你会发现验证器失败并报告无效的 bpf_context 访问。

第七章　eBPF 编程和附加类型

在前面的章节中，你看到了许多 eBPF 程序的示例，并且可能注意到它们附加到了不同类型的事件上。一些示例附加到了 kprobe 上，一些又附加到了 XDP 事件上。这只是内核中众多附加点中的两个。在本章中，我们将深入了解不同的程序类型以及它们如何附加到不同的事件上。

你可以使用 github.com/lizrice/learning-ebpf上的代码和说明来构建和运行本章中的示例。本章的代码位于 chapter7 目录中。

在撰写本文时，一些示例在 ARM 处理器上还不受支持。请查看 chapter7 目录中的 README 文件以获取更多信息。

目前在 uapi/linux/bpf.h 中枚举了约 30 种程序类型，并且有超过 40 种附加类型。附加类型更具体地定义了程序的附加位置；对于许多程序类型，可以从程序类型推断出附加类型，但某些程序类型可以附加到内核中的多个不同位置，因此还必须指定附加类型。

本书不是一本参考手册，因此我不会涵盖每种单独的 eBPF 程序类型。在你阅读本书时，很可能又添加了新的 eBPF 程序类型！

7.1　程序上下文参数

所有 eBPF 程序都接受一个上下文参数，该参数是一个指针，它指向的结构取决于触发该程序的事件类型。eBPF 程序员需要编写能够接受适当类型上下文的程序。如果事件是跟踪点，那么上下文参数指向网络数据包是没有意义的。定义不同类型的程序使得验证器能确保上下文信息得到适当处理，并强制执行有关可允许辅助函数的规则。

要深入了解传递给不同 BPF 程序类型的上下文数据的详细信息，请查看 Alan Maguire 在 Oracle 博客上发布的这篇文章。

7.2　辅助函数和返回值

验证器会检查程序使用的所有辅助函数是否与其程序类型兼容。上一章中的示例演示并表明在 XDP 程序中不允许使用 bpf_get_current_pid_tgid() 辅助函数。在接收数据包并触发 XDP 事件的时候，并没有涉及用户空间的进程或线程，因此在那种情况下调用获取当前进程和线程 ID 的函数是没有意义的。

程序类型还决定了程序的返回值的含义。再以 XDP 为例，返回值告诉内核在 eBPF 程序完成处理后应该如何处理数据包，可能将其传递到网络堆栈、丢弃它或将其重定向到不同的接口。如果 eBPF 程序是由命中特定跟踪点触发的（不涉及网络数据包），那么这些返回值就没有任何意义。

关于辅助函数有一个手册（由于 BPF 子系统的持续开发，声明可能还不完整）。

你可以使用 bpftool feature 命令获取内核中每种程序类型可用的辅助函数列表。该命令显示系统配置，并列出所有可用的程序类型和映射类型，甚至列出每种程序类型支持的所有辅助函数。

辅助函数被视为 UAPI 的一部分，即 Linux 内核的外部稳定接口。因此，一旦在内核中定义了辅助函数，它就不应在未来发生变化，即便内核内部函数和数据结构可能会发生变化。

虽然在不同内核间存在变化，但 eBPF 程序员都需要能够从 eBPF 程序中访问一些内部函数。这可以通过称为 BPF 内核函数或 kfunc 的机制来实现。

7.3 Kfuncs

Kfuncs 允许将内核函数注册到 BPF 子系统中，以便验证器允许 eBPF 程序调用这些函数。每种 eBPF 程序类型对允许调用的 kfunc 都有注册。

与辅助函数不同，kfuncs 不提供兼容性保证，因此 eBPF 程序员必须考虑不同内核版本间存在的变化。

在撰写本文时，存在一组 核心 BPF kfuncs，这些函数允许 eBPF 程序获取和释放对任务和 cgroup 的引用。

综上，eBPF 程序的类型决定了它可以附加到哪些事件上，进而定义了它接收的上下文信息的类型。程序类型还定义了它可以调用的辅助函数和 kfuncs 的集合。

程序类型通常分为两类：跟踪（或性能）程序类型、网络相关程序类型。

7.4 Tracing(跟踪)

附加到 kprobe、tracepoint、raw tracepoint、fentry/fexit 探针和 perf 事件的程序旨在为内核中的 eBPF 程序提供一种高效的方式，将有关事件的跟踪信息报告到用户空间。这些与跟踪相关的类型并不会影响内核对它们所附加事件的响应方式。

有时将它们称为“与性能相关的”程序。例如，bpftool perf 子命令允许你查看附加到与性能相关事件的程序，如下所示：

```
$ sudo bpftool perf show
pid 232272 fd 16: prog_id 392 kprobe func __x64_sys_execve offset 0
pid 232272 fd 17: prog_id 394 kprobe func do_execve offset 0
pid 232272 fd 19: prog_id 396 tracepoint sys_enter_execve
pid 232272 fd 20: prog_id 397 raw_tracepoint sched_process_exec
```

```
pid 232272 fd 21: prog_id 398 raw_tracepoint sched_process_exec
```

上述输出是在 chapter7 目录中运行 hello.bpf.c 文件的示例代码时看到的，它将不同的程序附加到与 execve() 相关的各种事件上。我将在本节中讨论所有这些类型，但总体上，这些程序包括：

- 附加到 execve() 系统调用入口点的 kprobe。
- 附加到内核函数 do_execve() 的 kprobe。
- 放置在 execve() 系统调用入口处的 tracepoint。
- 在处理 execve() 过程中调用的两个版本的 raw tracepoint，其中一个版本启用了 BTF。

要使用任何与跟踪相关的 eBPF 程序类型，你需要具备 CAP_PERFMON 和 CAP_BPF 或 CAP_SYS_ADMIN 权限。

7.4.1 Kprobes 和 Kretprobes

在第 1 章中，我讨论了 kprobe 的概念。你可以将 kprobe 程序附加到内核中的几乎任何位置。通常情况下，使用 kprobe 附加到函数的入口点，使用 kretprobe 附加到函数的出口点，但是你也可以使用 kprobe 将其附加到入口点之后的某个指令位置。如果选择这样做，你需要确保运行的内核版本在你认为的位置上具有要附加的指令！附加到内核函数的入口点和出口点可能相对稳定些，任意的代码行在不同版本间可能被修改。

在 bpftool perf list 的输出中，你可以看到两个 kprobe 的偏移量都为 0。

在内核编译时，编译器有可能选择对任何给定的内核函数进行“内联”操作；也就是说，编译器可能会在调用函数的位置插入机器码，以实现函数在调用函数中的功能。如果某个函数恰好被内联，那么你的 eBPF 程序将无法找到 kprobe 的入口点进行附加。

附加 kprobe 到系统调用入口点

本章的第一个示例 eBPF 程序名为 kprobe_sys_execve，它是一个附加到 execve() 系统调用的 kprobe。函数及其定义如下：

```
SEC("ksyscall/execve")
int BPF_KPROBE_SYSCALL(kprobe_sys_execve, char *pathname)
```

附加到系统调用的一个原因是它们是稳定的接口，在内核版本间不会发生变化。然而，出于某些原因，不应将系统调用 kprobe 用于安全工具。

附加 kprobes 到其他内核函数

你可以找到许多基于 eBPF 的工具，这些工具使用 kprobe 将其附加到系统调用，但是正如之前提到的，kprobe 也可以附加到内核中的任何非内联函数。我在 hello.bpf.c 中提供了一个示例，将 kprobe 附加到函数 do_execve() 上，它的定义如下：

```
SEC("kprobe/do_execve")
int BPF_KPROBE(kprobe_do_execve, struct filename *filename)
```

由于 do_execve() 不是一个系统调用，所以与前面的示例有一些不同之处：

• SEC 名称的格式与附加到系统调用入口点的格式完全相同，不需要定义特定于平台的变体，因为像大多数内核函数一样，do_execve() 在所有平台上都是通用的。

• 我用了 BPF_KPROBE 宏而不是 BPF_KPROBE_SYSCALL。两者意图相同，只是后者处理系统调用参数。

• 还有一个重要的区别：系统调用的 pathname 参数是一个指向字符串（char *）的指针，但是对于这个函数，参数被称为 file name，它是一个指向 struct filename 的指针，struct filename 是内核中使用的数据结构。

你可能想知道我是如何知道要为此参数使用此类型的，其实内核中的 do_execve() 函数具有以下签名：

```
int do_execve(struct filename *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
```

忽略 do_execve() 的参数 ___argv 和 ___envp，只声明 filename 参数，使用 struct filename * 类型来匹配内核函数的定义。考虑到参数在内存中的顺序排列方式，如果你想使用前面的参数，忽略最后 n 个参数是可以的，但是如果你想使用列表中前面的参数，则不能忽略更前面的的参数。

这个 filename 结构是内核定义的，这也表明了 eBPF 编程是内核编程。我不得不查找 do_execve() 的定义来找到它的参数以及 struct filename 的定义。可执行文件的名称由 filename->name 指向。

```
const char *name = BPF_CORE_READ(filename, name);
bpf_probe_read_kernel(&data.command, sizeof(data.command),
   name);
```

简要回顾一下：syscall kprobe 的上下文参数是一个表示用户空间传递给系统调用的值的结构体。而“常规”（非系统调用）kprobe 的上下文参数是一个表示由调用它的内核代码传递给被调用函数的参数的结构体，因此该结构体取决于函数的定义。

kretprobes 和 kprobes 非常相似，只是它在函数返回时触发，并且可以访问返回值而不是参数。

kprobes 和 kretprobes 是一种附加到内核函数的方式，但如果你正在运行最新的内核，还有一种更新的选择可以考虑。

7.4.2 Fentry/Fexit

从内核版本 5.5 开始（适用于 x86 处理器；对于 ARM 处理器，BPF trampoline 支持在 Linux 6.0 中引入），引入了一种更高效的机制来跟踪内核函数的进入和退出，即 BPF trampoline。如果你使用的是足够新的内核版本，fentry/fexit 现在是跟踪内核函数进入或退出的首选方法。你可以在 kprobe 或 fentry 类型的程序中编写相同的代码。

在 chapter7/hello.bpf.c 中有一个名为 fentry_execve() 的 fentry 程序示例。我使用了 libbpf 宏 BPF_PROG 来声明这个 kprobe 的 eBPF 程序。这是另一个方便的封装，可以访问类型化的参数而不是通用的上下文指针，此版本适用于 fentry、fexit 和 tracepoint 程序类型。定义如下：

```
SEC("fentry/do_execve")
int BPF_PROG(fentry_execve, struct filename *filename)
```

段名称告诉 libbpf 将其附加到 do_execve() 内核函数的 fentry hook 开始处。与 kprobe 示例中一样，上下文参数反映了传递给你要附加这个 eBPF 程序的内核函数的参数。

Fentry 和 fexit 的挂钩点设计得比 kprobe 更高效，如果想在函数结束时生成一个事件，fexit hook 可以访问函数的输入参数，而 kretprobe 不能。你可以在 libbpf-bootstrap 的示例中查看这个例子。kprobe.bpf.c 和 fentry.bpf.c 两个效果都是一样的，都挂钩到 do_unlinkat() 内核函数。附加到 kretprobe 的 eBPF 程序具有以下签名：

```
SEC("kretprobe/do_unlinkat")
int BPF_KRETPROBE(do_unlinkat_exit, long ret)
```

BPF_KRETPROBE 宏会将其扩展为一个从 do_unlinkat() 退出的 kretprobe 程序。eBPF 程序只接收一个参数 ret，它保存了 do_unlinkat() 的返回值。与 fexit 版本进行比较：

```
SEC("fexit/do_unlinkat")
int BPF_PROG(do_unlinkat_exit, int dfd, struct filename *name
    , long ret)
```

在这个版本中，程序不仅可以访问返回值 ret，还可以访问 do_unlinkat() 的输入参数 dfd 和 name。

7.4.3 跟踪点 (Tracepoints)

跟踪点是内核代码中标记的位置。它们不是专属于 eBPF 的，长期以来一直被用于生成内核跟踪输出，并用在诸如SystemTap之类的工具中。与使用 kprobes 附加到任意指令不同，跟踪点在内核版本之间是稳定的。

你可以查看/sys/kernel/tracing/available_events 来查看内核中可用的跟踪子系统集合，如下所示：

```
$ cat /sys/kernel/tracing/available_events
tls:tls_device_offload_set
tls:tls_device_decrypted
...
syscalls:sys_exit_execveat
syscalls:sys_enter_execveat
syscalls:sys_exit_execve
syscalls:sys_enter_execve
...
```

我使用的 5.15 版本的内核在此列表中定义了 1,400 多个跟踪点。跟踪点 eBPF 程序的 SEC 定义应与这些项中得某个匹配，以便 libbpf 可以自动将其附加到跟踪点上。定义的格式为 SEC("tp/跟踪子系统/跟踪点名称")。

在 chapter7/hello.bpf.c 文件中，你会找到一个与 syscalls:sys_enter_execve 跟踪点匹配的示例，该跟踪点在内核开始处理 execve() 调用时被触发。SET 定义告诉 libbpf 这是一个跟踪点程序，并指定了它应该附加到的位置，如下所示：

```
SEC("tp/syscalls/sys_enter_execve")
```

关于跟踪点的上下文参数，我马上会介绍，但首先让考虑一下在没有 BTF 的情况下我们还需要什么信息。每个跟踪点都有一个描述跟踪出的字段格式。作为示例，下面是 execve() 系统调用进入跟踪点的格式：

```
$ cat /sys/kernel/tracing/events/syscalls/sys_enter_execve/
   format
name: sys_enter_execve
ID: 622
format:
    field:unsigned short common_type;   offset:0; size:2;
       signed:0;
    field:unsigned char common_flags;   offset:2; size:1;
       signed:0;
    field:unsigned char common_preempt_count; offset:3; size
       :1; signed:0;
    field:int common_pid;               offset:4; size:4;
       signed:1;

    field:int __syscall_nr;             offset:8;  size:4;
       signed:1;
    field:const char * filename;        offset:16; size:8;
```

```
        signed:0;
    field:const char *const * argv;     offset:24; size:8;
        signed:0;
    field:const char *const * envp;     offset:32; size:8;
        signed:0;

print fmt: "filename: 0x%08lx, argv: 0x%08lx, envp: 0x%08lx",
((unsigned long)(REC->filename)), ((unsigned long)(REC->argv)
    ),
((unsigned long)(REC->envp))
```

我使用这些信息在 chapter7/hello.bpf.c 中定义了一个叫做 my_syscalls_enter_execve 的结构体。

```
struct my_syscalls_enter_execve {
    unsigned short common_type;
    unsigned char common_flags;
    unsigned char common_preempt_count;
    int common_pid;
    long syscall_nr;
    long filename_ptr;
    long argv_ptr;
    long envp_ptr;
};
```

eBPF 程序不允许访问这四个字段中的前四个。如果尝试访问它们，程序将在验证时失败并显示无效的 bpf_context 访问错误。

我在示例中使用了一个指向这种类型的指针作为上下文参数，如下所示：

```
int tp_sys_enter_execve(struct my_syscalls_enter_execve *ctx) {
```

然后你可以访问这个结构体的内容。例如，你可以按如下方式获取 filename 指针：

```
bpf_probe_read_user_str(&data.command, sizeof(data.command),
    ctx->filename_ptr);
```

使用 tracepoint 程序类型时，传递给 eBPF 程序的结构体已经从一组原始参数映射过来。为提高性能，可以使用原始 tracepoint eBPF 程序类型直接访问这些原始参数。部分定义应该以 raw_tp（或 raw_tracepoint）开头，而不是 tp。你需要将参数从 ___u64 转换为跟踪点结构体使用的类型（当跟踪点是系统调用的入口时，这些参数取决于芯片架构）。

7.4.4 启用 BTF 的追踪点

在前面的示例中,我编写了一个名为 my_syscalls_enter_execve 的结构体来定义 eBPF 程序的上下文参数。但是，当在 eBPF 代码中定义一个结构体或解析原始参数时存在一个风险，即代码可能与运行的内核不匹配。好消息是，BTF 可以解决这个问题。

有了 BTF 支持，将在 vmlinux.h 中定义一个与传递给 tracepoint eBPF 程序的上下文结构相匹配的结构体。你的 eBPF 程序应该使用以下形式的 SEC 定义:SEC("tp_btf/tracepoint name")，其中 tracepoint name 是/sys/kernel/tracing/available_events 中列出的可用事件之一。chapter7/hello.bpf.c 中的程序如下，结构体名称与 tracepoint 名称相匹配，前缀为 trace_event_raw_。

```
SEC("tp\_btf/sched\_process\_exec")
int handle\_exec(struct trace\_event\_raw\_sched\_process\_exec *ctx)
```

7.4.5 用户空间附加

到目前为止，我已经展示了 eBPF 程序附加到内核源码定义的事件的示例。在用户空间代码中也有类似的附加点：uprobes 和 uretprobes 用于附加到用户空间函数的入口和退出点，以及用户静态定义的跟踪点（USDTs）用于附加到应用程序代码或用户空间库中指定的跟踪点。所有这些都使用 BPF_PROG_TYPE_KPROBE 程序类型。有很多公开附加到用户空间事件的程序。以下是来自 BCC 项目的示例：

- bashreadline和bfunclatency工具附加到 uprobes 和 uretprobes。
- BCC 中的USDT 示例。

如果你使用 libbpf，SEC() 宏允许你为这些用户空间探测点定义自动附加点。你可以在libbpf 文档中找到所需的 SEC 名称格式。例如，要将一个 uprobes 附加到 OpenSSL 的 SSL_write() 函数的开始位置，你可以使用以下方式定义 eBPF 程序的部分：

```
SEC("uprobe/usr/lib/aarch64-linux-gnu/libssl.so.3/SSL\_write")
```

在检测用户空间代码时，有一些需要注意的问题：

- 注意此示例中共享库路径与特定架构相关，因此你可能需要架构的定义。
- 除非你控制运行代码的机器，否则无法知道安装了哪些用户空间库和应用程序。
- 应用程序可能构建成独立的二进制文件，因此不会触发共享库中附加的探测点。
- 容器通常在其自己的文件系统上运行，并安装了一组自己的依赖项。容器中使用的共享库的路径与主机机器上的共享库路径不同。
- eBPF 程序可能要知道应用程序使用的语言。例如，C 语言中函数参数通常用寄存器传递，而 Go 语言使用堆栈传递，因此保存寄存器信息的 pt_args 结构可能没有用。

话虽如此，有很多实用的工具可以使用 eBPF 为用户空间应用程序提供支持。例如，可以注入 SSL 库来跟踪解密后的加密信息。另一个例子是使用Parca等工具对应用程序进行连续性分析。

7.4.6 Linux 安全模块 (LSM)

BPF_PROG_TYPE_LSM 程序附加到 Linux Security Module (LSM) API 上，这是内核中的一个稳定接口，最初用于内核模块强制执行安全策略，现在 eBPF 安全工具也可以使用此接口。

使用 bpf(BPF_RAW_TRACEPOINT_OPEN) 附加 BPF_PROG_TYPE_LSM 程序，在许多方面它们都像跟踪程序一样处理。BPF_PROG_TYPE_LSM 程序的一个有趣特点是返回值会影响内核的行为方式。非零的返回代码表示安全检查未通过，因此内核将不会执行被要求完成的任何操作。这与性能相关的程序类型存在显著差异，后者忽略返回代码。Linux 内核文档涵盖了LSM BPF 程序。

LSM 程序类型不是唯一在安全领域发挥作用的类型。下一节你将看到许多与网络相关的程序类型用于网络安全领域 (允许或拒绝网络流量等操作)。本章中，你看到了一组内核和用户空间跟踪程序类型如何实现对整个系统的可见性。下一组要考虑的 eBPF 程序类型是让我们能够钩入网络栈的程序类型，不仅可以观察数据的传输和接收，还可以对其处理方式产生影响。

7.5 网络

有许多 eBPF 程序类型为了在网络栈的各个点通过时处理网络消息。图 7-1 显示了一些常用程序类型的附加位置。所有这些程序类型都需要获得 CAP_NET_ADMIN 和 CAP_BPF 或 CAP_SYS_ADMIN 权限才能被允许。

传递给这些程序类型的上下文是网络消息，尽管结构类型取决于内核在网络栈的相应点上具有的数据。在网络栈的底部，数据以第 2 层网络数据包的形式保存，它们就是一系列字节，在网络上传输或准备传输。在网络栈的顶部，应用程序使用套接字，内核创建套接字缓冲区来处理从这些套接字发送和接收的数据。

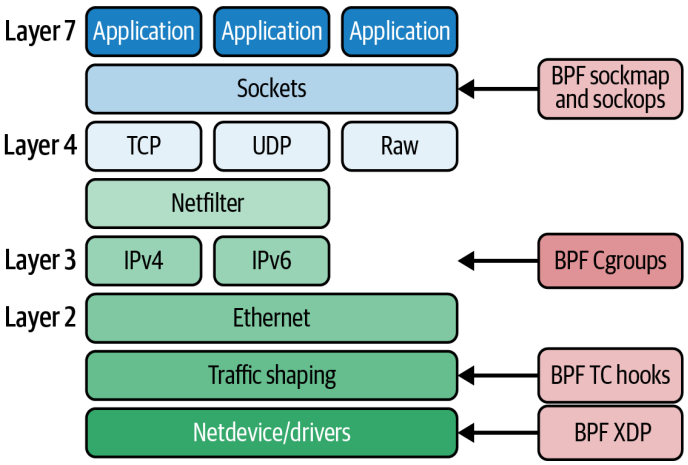

图 7.1: BPF 程序类型可以钩入网络栈中的各层

网络七层模型超出了本书的范围，但在许多其他书籍、文章和培训课程中有所涵盖。我在《容器安全》（O'Reilly）的第 10 章中讨论过这个问题。对于本书而言，知道第 7 层涵盖了面向应用程序使用的格式（如 HTTP、DNS 或 gRPC）、TCP 位于第 4 层、IP 位于第 3 层，以太网和 WiFi 位于第 2 层就足够了。网络栈的一个作用是在这些不同的格式之间进行消息转换。

网络程序类型和本章前面介绍的与跟踪相关的程序类型之间的一个重要区别是，网络程序类型通常旨在允许自定义网络行为。这涉及两个主要特征：

1. 使用 eBPF 程序的返回码告诉内核如何处理网络数据包——可以是按照通常方式处理、丢弃或重定向到其他目的地。

2. 允许 eBPF 程序修改网络数据包、套接字配置参数等。

在下一章中，你将看到如何利用这些特征构建强大的网络功能，但现在，让我们先来概述一下 eBPF 程序类型。

7.5.1　套接字（Sockets）

在网络栈的顶部，一组与套接字和套接字操作相关的网络程序类型如下：

- BPF_PROG_TYPE_SOCKET_FILTER 是最早添加到内核中的程序类型。从名称中你可能猜到了它用于套接字过滤，但这并不意味着过滤应用程序发送或接收的数据。它是用于过滤可以发送到诸如 tcpdump 等可观察性工具的套接字数据。

- 套接字是特定于第 4 层（TCP）连接的。BPF_PROG_TYPE_SOCK_OPS 允许 eBPF 程序拦截在套接字上进行的各种操作和动作，并为该套接字设置参数，如 TCP 超时值。套接字仅存在于连接的端点，而不会存在于可能经过的任何中间层。

- BPF_PROG_TYPE_SK_SKB 程序与一种特殊的映射类型一起使用，该映射类型保存了一组对套接字的引用，以提供所谓的套接字映射操作：在套接字层将流量重定向到不同的目的地。

7.5.2　流量控制

在网络栈的较低层是 "TC"（traffic control）或流量控制。Linux 内核中有一个与 TC 相关的完整子系统，查看 tc 命令的man 页可以让你了解它的复杂性以及在处理网络数据包的方式上需要深层次灵活性和配置的重要性。

eBPF 程序可以附加到提供自定义的入口和出口流量的网络数据包过滤器和分类器。这是 Cilium 项目的一个基本构建块，我将在下一章中介绍一些示例。如果你等不及的话，Quentin Monnet 的博客上有一些很好的示例。你可以通过编程实现这一点，但你也可以选择使用 tc 命令来操作这些类型的 eBPF 程序。

7.5.3　XDP

在第 3 章中，你了解了 XDP（eXpress Data Path）eBPF 程序。在那个示例中，我加载了 eBPF 程序，并使用以下命令将其附加到 eth0 接口：

```
bpftool prog load hello.bpf.o /sys/fs/bpf/hello
bpftool net attach xdp id 540 dev eth0
```

值得注意的是，XDP 程序附加到特定的接口（或虚拟接口），很可能会在不同的接口上附加不同的 XDP 程序。在第 8 章中，你将了解有关如何将 XDP 程序加载到网络卡并由网络驱动程序执行的信息。

XDP 程序是可以使用 Linux 网络工具进行管理的另一个示例，本例中使用的是iproute2 的 ip 命令的 link 子命令。加载并将程序附加到 eth0 的近似等效命令如下：

```
$ ip link set dev eth0 xdp obj hello.bpf.o sec xdp
```

此命令从 hello.bpf.o 对象中读取标记为 xdp 段的 eBPF 程序，并将其附加到 eth0 网络接口。现在，ip link show 命令对该接口包含了一些有关附加到它的 XDP 程序的信息：

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500
    xdpgeneric qdisc fq\_codel
state UP mode DEFAULT group default qlen 1000
    link/ether 52:55:55:3a:1b:a2 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 1255 tag 9d0e949f89f1a82c jited
```

使用 ip link 删除 XDP 程序可以通过以下方式完成：

```
$ ip link set dev eth0 xdp off
```

在下一章中，你将看到更多关于 XDP 程序及其应用的内容。

7.5.4 流解析器

流解析器在网络栈各处用于从数据包头中提取信息。BPF_PROG_TYPE_FLOW_DISSECTOR 类型的 eBPF 程序可以实现自定义的数据包解析。在这篇 LWN 文章中有关于使用 BPF 编写网络流解析器的详细介绍。

7.5.5 轻量级隧道

BPF_PROG_TYPE_LWT_* 程序类型系列可以在 eBPF 程序中实现网络封装。这些程序类型也可以使用 ip 命令进行操作，但这次是使用的 route 子命令。在实践中，这些并不常用。

7.5.6 资源控制组

eBPF 程序可以附加到 cgroups（控制组）。Cgroups 是 Linux 内核中的一个概念，它限制了给定进程或一组进程可以访问的资源集合。Cgroups 是将一个容器（或一个 Kubernetes

Pod）与另一个容器隔离的机制之一。将 eBPF 程序附加到 cgroup 允许自定义行为仅适用于该 cgroup 的进程。所有进程都与一个 cgroup 相关联，包括不在容器内运行的进程。

有几种与 cgroup 相关的程序类型，以及更多可以附加它们的钩子。至少在撰写本书时，几乎所有这些程序类型都与网络相关，尽管还有一个 BPF_CGROUP_SYSCTL 程序类型可以附加到影响特定 cgroup 的 sysctl 命令上。

以套接字相关的程序类型为例，有专门针对 cgroups 的 BPF_PROG_TYPE_CGROUP_SOCK 和 BPF_PROG_TYPE_CGROUP_SKB。eBPF 程序可以确定给定的 cgroup 是否被允许执行所请求的套接字操作或数据传输。这对于网络安全策略执行非常有用。套接字程序还可以欺骗调用进程，使其认为它们正在连接到特定的目标地址。

7.5.7 红外控制器

BPF_PROG_TYPE_LIRC_MODE2类型的程序可以附加到红外控制器设备的文件描述符上，以提供红外协议的解码功能。在撰写本书时，该程序类型需要 CAP_NET_ADMIN 权限，但我认为这说明了将程序类型分为与跟踪相关和与网络相关的分类并不能完全区分 eBPF 可以处理的不同应用范围。

7.6 BPF 附加类型

附加类型可以更细致地控制程序在系统中的附加位置。对于某些程序类型，它们与可附加到的钩子类型存在一对一的对应关系，因此附加类型由程序类型隐式定义。例如，XDP 程序附加到网络栈中的 XDP 钩子。对于一些程序类型，还必须指定附加类型。

附加类型涉及到决定哪些辅助函数是有效的，并且在某些情况下限制对上下文信息的访问。在前面的示例中，你可以看到验证器报告了一个无效的 bpf_context 访问错误。

你还可以在内核函数bpf_prog_load_check_attach（在bpf/syscall.c中定义）中查看哪些程序类型需要指定附加类型，以及哪些附加类型是有效的。

例如，以下是检查 CGROUP_SOCK 类型 eBPF 程序的附加类型的代码：

```
case BPF\_PROG\_TYPE\_CGROUP\_SOCK:
    switch (expected\_attach\_type) {
    case BPF\_CGROUP\_INET\_SOCK\_CREATE:
    case BPF\_CGROUP\_INET\_SOCK\_RELEASE:
    case BPF\_CGROUP\_INET4\_POST\_BIND:
    case BPF\_CGROUP\_INET6\_POST\_BIND:
        return 0;
    default:
    return -EINVAL;
}
```

这种 eBPF 程序类型可以在多个地方进行挂载：在套接字创建时、在套接字释放时，或者在 IPv4 或 IPv6 绑定完成后。

另一个可以找到程序有效挂载类型列表的地方是 libbpf文档，你还可以找到 libbpf 对每个程序和挂载类型解释得章节名称。

7.7 总结

在本章中，你了解到各种 eBPF 程序类型被用于挂载到内核的不同钩子点上。如果你想编写能够响应特定事件的代码，你需要确定适合挂载到该事件的程序类型。传递给程序的上下文取决于程序类型，并且内核可能会根据程序类型对你的程序的返回代码作出不同的响应。

本章的示例代码主要集中在与性能相关的（跟踪）事件上。在接下来的两章中，你将看到更多关于不同 eBPF 程序类型在网络和安全应用中的详细信息。

7.8 练习

本章的示例代码包括了将 kprobe、fentry、tracepoint、原始 tracepoint 和启用了 BTF 的 tracepoint 程序都挂载到同一个系统调用入口的示例。除了系统调用之外，eBPF 跟踪程序还可以挂载到其他位置。

1. 使用 strace 运行代码，捕获 bpf() 系统调用的信息，像这样：

```
strace -e bpf -o outfile ./hello
```

这将把每个 bpf() 系统调用信息记录到名为 outfile 的文件中。在该文件中查找 BPF_PROG_LOAD 指令，并观察不同程序的 prog_type 字段的变化。你可以通过跟踪中的 prog_name 字段来确定每个程序，然后将其与章节 7 中 hello.bpf.c 中的源代码进行匹配。

2. hello.c 中的用户空间代码加载了在 hello.bpf.o 中定义的所有程序对象。作为编写 libbpf 用户空间代码的练习，修改示例代码，仅加载并挂载其中一个 eBPF 程序，而不从 hello.bpf.c 中删除这些程序。

3. 编写一个 kprobe 和/或 fentry 程序，当调用其他内核函数时触发。你可以通过查看/proc/kallsyms 文件来查找你的内核版本中可用的函数。

4. 编写一个常规的、原始的或启用了 BTF 的 tracepoint 程序，将其挂载到其他内核 tracepoint 上。你可以在/sys/kernel/tracing/available_events 目录中找到可用的 tracepoint。

5. 尝试将多个 XDP 程序挂载到给定的接口，并确认你无法做到！你应该会看到类似以下的错误信息：

```
libbpf: Kernel error message: XDP program already attached
Error: interface xdpgeneric attach failed: Device or resource busy
```

第八章　eBPF 用于网络

eBPF 的动态特性让我们能够定制内核的行为。在网络世界中，取决于不同的应用场景，有许多与应用程序相关的理想行为。例如，电信运营商可能需要与特定于电信的协议（如 SRv6）进行接口交互；Kubernetes 环境可能需要与传统应用程序集成；专用硬件负载均衡器可以被运行在通用硬件上的 XDP 程序所替代。eBPF 允许程序员构建网络功能以满足特定需求，而无需强加给所有上游内核用户。

基于 eBPF 的网络工具现在被广泛使用，并且在大规模上已证明其有效性。例如，CNCF 的Cilium 项目使用 eBPF 作为 Kubernetes 网络、独立负载均衡等方面的平台，被云原生采用者在各行各业中使用。Meta 自 2017 年以来一直在大规模使用 eBPF，Facebook 的每个数据包都经过了一个 XDP 程序，Cloudflare 使用 eBPF 进行 DDoS（分布式拒绝服务）保护。

这些都是复杂的、可用于生产环境的解决方案，其细节超出了本书的范围，但通过阅读本章中的示例，你可以对如何构建这些 eBPF 网络解决方案有所了解。

本章的示例代码位于github.com/lizrice/learning-ebpf仓库的 chapter8 目录中。

8.1　丢包

有几种涉及丢弃特定传入数据包并允许其他数据包的网络安全功能。这些功能包括防火墙、DDoS 保护和减轻“致命数据包”漏洞：

- 防火墙在数据包的基础上决定是否允许它通过，根据数据源 IP 地址、目标 IP 地址和端口号进行判断。
- DDoS 保护会增加一点复杂性，例如跟踪从特定来源到达的数据包速率或检测数据包内容的某些特征，以确定攻击者或一组攻击者是否试图向接口发送大流量。
- “致命数据包”漏洞是一类内核漏洞，内核在处理特定方式构造的数据包时无法安全处理。发送具有特定格式的数据包的攻击者可以利用这个漏洞引起内核崩溃。发现这种内核漏洞时，传统做法就是升级安装修复程序。

这些功能的决策算法超出了本书的范围，但我们可以探讨一下如何通过将 eBPF 程序附加到网络接口的 XDP 钩子上来丢弃某些数据包，这是实现上述复杂方案的基础。

8.1.1 XDP 程序返回码

XDP 程序由网络数据包的到达触发。程序检查数据包，完成后，返回代码给出一个判断，指示接下来对该数据包应采取什么操作：

- XDP_PASS 表示数据包应按正常方式发送到网络栈（没有 XDP 程序时这是默认操作）。
- XDP_DROP 立即丢弃数据包。
- XDP_TX 将数据包发送回到达接口。
- XDP_REDIRECT 将数据包发送到另一个网络接口。
- XDP_ABORTED 丢弃数据包，但这时往往是出现了错误或意外情况，而不是正常丢弃数据包。

对于某些用例（如防火墙），XDP 程序只需决定是传递数据包还是丢弃它。一个决定是否丢弃数据包的 XDP 程序的示例大致如下：

```
SEC("xdp")
int hello(struct xdp_md *ctx) {
    bool drop;

    drop = <examine packet and decide whether to drop it>;

    if (drop)
        return XDP_DROP;
    else
        return XDP_PASS;
}
```

XDP 程序还可以操作数据包内容，我稍后在本章中介绍。

XDP 程序在网络数据包到达其所附加的接口时触发。ctx 参数是指向 xdp_md 结构的指针，该结构保存着有关传入数据包的元数据。让我们看看如何使用这个结构来检查数据包的内容，以便做出判断。

8.1.2 XDP 数据包解析

以下是 xdp_md 结构的定义：

```
struct xdp_md {
    __u32 data;
    __u32 data_end;
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; /* rxq->dev->ifindex */
```

```
    __u32 rx_queue_index;  /* rxq->queue_index */

    __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

不要被前三个字段的 ___u32 类型所迷惑，它们实际上是指针。data 字段指示数据包开始的内存位置，而 data_end 则指示数据包结束的位置。正如你在第 6 章中所看到的，为了通过 eBPF 验证器，你必须明确检查对数据包内容的任何读写操作是否在 data 到 data_end 的范围内。

在 data 和 data_meta 之间，数据包前面的内存区域用于存储关于该数据包的元数据。这可用于在多个 eBPF 程序之间进行协调，这些程序可能会在数据包通过网络堆栈的不同位置进行处理。

为了说明解析网络数据包的基础知识，示例代码中有一个名为 ping() 的 XDP 程序，它在检测到 ping（ICMP）数据包时会生成一行跟踪信息。以下是该程序的代码：

```
SEC("xdp")
int ping(struct xdp_md *ctx) {
    long protocol = lookup_protocol(ctx);
    if (protocol == 1) // ICMP
    {
        bpf_printk("Hello ping");
    }
    return XDP_PASS;
}
```

你可以按照以下步骤查看此程序的运行情况：

1. 在 chapter8 目录中运行 make 命令。这不仅会编译代码，还会将 XDP 程序附加到环回接口（称为 lo）上。

2. 在一个终端窗口中运行 ping localhost。

3. 在另一个终端窗口中，通过运行 cat /sys/kernel/tracing/trace_pipe 命令来观察跟踪管道中生成的输出。

你应该看到每秒钟大约两行跟踪信息输出，并且输出的内容类似于以下内容：

```
ping-26622 [000] d.s11 276880.862408: bpf_trace_printk: Hello ping
ping-26622 [000] d.s11 276880.862459: bpf_trace_printk: Hello ping
ping-26622 [000] d.s11 276881.889575: bpf_trace_printk: Hello ping
ping-26622 [000] d.s11 276881.889676: bpf_trace_printk: Hello ping
ping-26622 [000] d.s11 276882.910777: bpf_trace_printk: Hello ping
ping-26622 [000] d.s11 276882.910930: bpf_trace_printk: Hello ping
```

每秒钟会生成两行跟踪信息，这是因为环回接口同时接收到 ping 请求和 ping 响应。

如果你想通过修改代码来丢弃 ping 数据包，只需在协议匹配时添加一行代码 return XDP_DROP，如下所示：

```
if (protocol == 1) // ICMP
{
    bpf_printk("Hello ping");
    return XDP_DROP;
}
return XDP_PASS;
```

如果你尝试这样做，你会发现类似以下的输出，每秒钟只会在跟踪输出中生成一次：

```
ping-26639 [002] d.s11 277050.589356: bpf_trace_printk: Hello ping
ping-26639 [002] d.s11 277051.615329: bpf_trace_printk: Hello ping
ping-26639 [002] d.s11 277052.637708: bpf_trace_printk: Hello ping
```

环回接口接收到一个 ping 请求，而 XDP 程序将其丢弃，因此该请求无法通过网络堆栈进一步传播以获得响应。

在这个 XDP 程序中，大部分工作是在一个名为 lookup_protocol() 的函数中完成的，该函数确定了第 4 层协议类型。这只是一个示例，不是一个生产级别的网络数据包解析实现！但它足以让你了解在 eBPF 中如何进行解析数据包了。

接收到的网络数据包由一串字节组成，布局如图 8-1 所示。

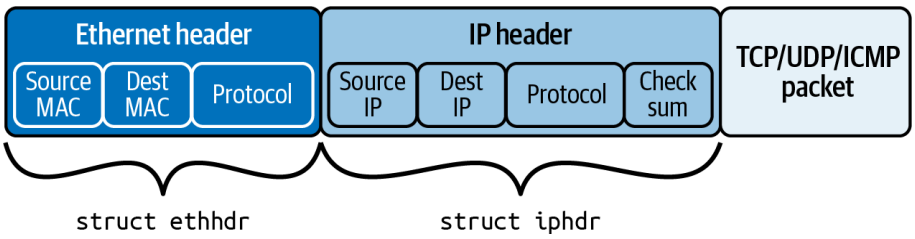

图 8.1: IP 网络数据包的布局，以太网头开始，接着是 IP 头部，然后是第 4 层的数据

lookup_protocol() 函数接收 ctx 结构体，其中包含有关网络数据包在内存中的位置信息，并返回在 IP 标头中找到的协议类型。以下是代码：

```
unsigned char lookup_protocol(struct xdp_md *ctx) {
    unsigned char protocol = 0;
    void *data = (void *)(long)ctx->data; // 1
    void *data_end = (void *)(long)ctx->data_end;
    struct ethhdr *eth = data; // 2
    if (data + sizeof(struct ethhdr) > data_end) // 3
```

```
            return 0;

        // Check that it's an IP packet
        if (bpf_ntohs(eth->h_proto) == ETH_P_IP) // 4
        {
            // Return the protocol of this packet
            // 1 = ICMP
            // 6 = TCP
            // 17 = UDP
            struct iphdr *iph = data + sizeof(struct ethhdr); //
               5
            if (data + sizeof(struct ethhdr) + sizeof(struct
               iphdr) <= data_end) // 6
                protocol = iph->protocol; // 7

        }
        return protocol;
    }
```

1. 本地变量 data 和 data_end 指向网络数据包的起始和结束位置。

2. 网络数据包应以以太网头开始。

3. 但不能简单地假设这个网络数据包足够大以容纳以太网头！验证器要求你明确进行检查。

4. 以太网头包含一个 2 字节的字段，用于指示第 3 层协议。

5. 如果协议类型表示它是一个 IP 数据包，则 IP 头紧随以太网头之后。

6. 不能简单地假设网络数据包中有足够的空间来容纳该 IP 头。验证器再次要求你明确进行检查。

7.IP 头包含了该函数将返回给调用者的协议字节。

此程序中使用的 bpf_ntohs() 函数确保两个字节的顺序与主机的顺序相符。网络协议采用大端序（Big-Endian），但大多数处理器采用小端序（Little-Endian），这意味着它们以不同的顺序存储多字节值。该函数将从网络字节序转换为主机字节序。在从网络数据包的字段中提取超过一个字节的值时，应使用此函数。

这个简单的示例展示了仅用几行 eBPF 代码就可以对网络功能产生巨大的影响。很容易想象，更复杂的规则决定哪些数据包要传递，哪些数据包要丢弃，可以实现本节开头描述的各种功能：防火墙、DDoS 保护和“致命数据包”漏洞缓解。现在让我们考虑在 eBPF 程序中修改网络数据包的功能，进一步提供更多功能。

8.2 负载均衡和转发

XDP 程序不仅限于检查数据包的内容，还可以修改数据包的内容。考虑一下如果要构建一个简单的负载均衡器，将发送到特定 IP 地址的数据包转发到多个能够处理请求的后端服务器上，需要做哪些工作。

在 GitHub 仓库中有一个示例。这里的设置是一组在同一主机上运行的容器。包括客户端、负载均衡器和两个后端服务器，每个服务器运行在自己的容器中。如图 8-2 所示，负载均衡器接收来自客户端的流量并将其转发到两个后端容器之一。

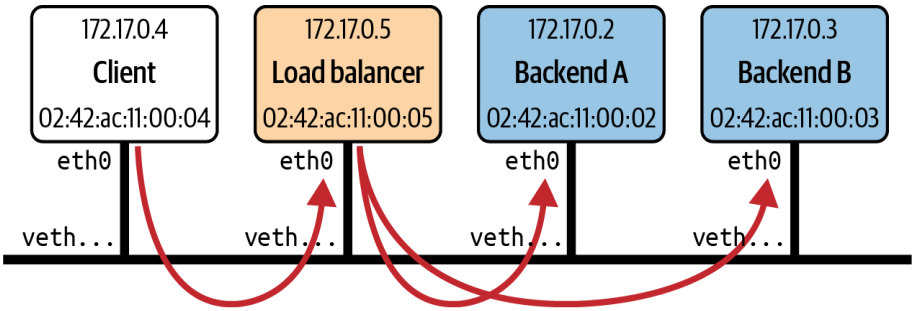

图 8.2: 负载均衡器设置示例

负载均衡功能是以一个 XDP 程序实现的，附加到负载均衡器的 eth0 网络接口上。该程序的返回码是 XDP_TX，表示数据包应该从它所进入的接口返回。但在此之前，程序必须更新数据包头中的地址信息。

虽然我认为这个示例代码对于学习很有用，但它离生产级别代码还有很长的路要走；例如，它假设 IP 地址的设置与图 8-2 中显示的完全一样，使用了硬编码地址。它还假设将接收到的唯一 TCP 流量是来自客户端的请求或响应。它还通过利用 Docker 设置虚拟 MAC 地址的方式进行了欺骗，使用每个容器的 IP 地址作为每个容器的虚拟以太网接口的 MAC 地址的最后四个字节。从容器的角度来看，该虚拟以太网接口被称为 eth0。

这是示例负载均衡器代码中的 XDP 程序：

```
SEC("xdp_lb")
int xdp_load_balancer(struct xdp_md *ctx)
{
    void *data = (void *)(long)ctx->data; // 1
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth = data;
    if (data + sizeof(struct ethhdr) > data_end)
        return XDP_ABORTED;
    if (bpf_ntohs(eth->h_proto) != ETH_P_IP)
        return XDP_PASS;
```

```
    struct iphdr *iph = data + sizeof(struct ethhdr);
    if (data + sizeof(struct ethhdr) + sizeof(struct iphdr) >
        data_end)
        return XDP_ABORTED;

    if (iph->protocol != IPPROTO_TCP) // 2
        return XDP_PASS;

    if (iph->saddr == IP_ADDRESS(CLIENT)) // 3
    {
        char be = BACKEND_A;  // 4
        if (bpf_get_prandom_u32() % 2)
        be = BACKEND_B;

        iph->daddr = IP_ADDRESS(be); // 5
        eth->h_dest[5] = be;
    } else {
        iph->daddr = IP_ADDRESS(CLIENT); // 6
        eth->h_dest[5] = CLIENT;
    }
    iph->saddr = IP_ADDRESS(LB); // 7
    eth->h_source[5] = LB;

    iph->check = iph_csum(iph);

    return XDP_TX;
}
```

1. 函数第一部分与之前示例几乎相同：它定位了网络包中的以太网头部和 IP 头部。

2. 这次它只处理 TCP 包，对于接收到的其他类型的包，原样传递给上层堆栈。

3. 这里，源 IP 地址被检查。如果包不是来自客户端，会假设它是发往客户端的响应。

4. 这段代码在 A 和 B 后端之间生成一个伪随机选择。

5. 目标 IP 和 MAC 地址将更新为与所选择的后端相匹配...

6. 如果这是来自后端的响应，目标 IP 和 MAC 地址将更新为与客户端相匹配。

7. 无论这个包要去哪里，源地址都需要更新，使得看起来像是由负载均衡器发出的包。

8.IP 头部包括对其内容计算的校验和，由于源和目标 IP 地址都已更新，因此校验和也需要重新计算并替换。

由于这本书是关于 eBPF 而不是网络的，我没有深入探讨为什么需要更新 IP 和 MAC 地址，或者如果不进行更新会发生什么。如果你感兴趣，我在 eBPF 峰会演讲的YouTube视频中对此进行了介绍。

和前面的示例一样，Makefile 包括构建代码以及使用 bpftool 加载和附加 XDP 程序到接口的指令，如下所示：

```
xdp: $(BPF_OBJ)
    bpftool net detach xdpgeneric dev eth0
    rm -f /sys/fs/bpf/$(TARGET)
    bpftool prog load $(BPF_OBJ) /sys/fs/bpf/$(TARGET)
    bpftool net attach xdpgeneric pinned /sys/fs/bpf/$(TARGET
       ) dev eth0
```

这个 make 指令需要在负载均衡器容器内部运行，以使 eth0 对应于其虚拟以太网接口。这引出了一个有趣的问题：eBPF 程序加载到内核中，而内核只有一个；然而，附加点可能在特定的网络命名空间内，只在该网络名字空间内可见。

8.3 XDP 卸载

XDP 的概念源于一次关于在网络适配器上运行 eBPF 程序以在数据包到达内核网络栈之前对其进行处理的对话。实际上，有一些支持完整 XDP 卸载功能的网络接口卡，它们可以在自己的处理器上运行 eBPF 程序来处理传入的数据包。如图 8-3 所示。

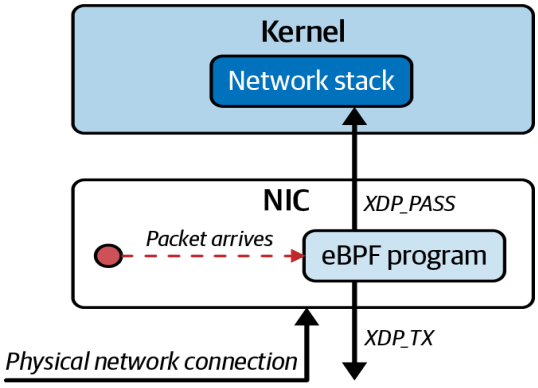

图 8.3: 支持 XDP 卸载的网络接口卡可以在不需要干预的情况下处理、丢弃和重发数据包

这意味着被丢弃或重定向回相同物理接口的数据包，不会被内核看到，主机上的 CPU 周期也不会用于处理它们，因为所有的工作都在网络适配器上完成了。

即使物理网络适配器不支持完整的 XDP 卸载，许多网络适配器驱动程序也支持XDP挂钩，可以最大限度地减少 eBPF 程序处理数据包所需的内存拷贝。这能带来显著的性能优势，并且使得像负载均衡这样的功能在通用硬件上运行得非常高效。

你已经看到了 XDP 如何用于处理传入的网络数据包，在数据包到达机器时尽早地访问它们。eBPF 还可用于在网络堆栈的其他点处理流量。下面继续思考在 TC 子系统中附加 eBPF 程序的情况。

8.4 流量控制（TC）

在前一章中，我提到了流量控制（Traffic Control）。当网络数据包达到这一点时，它将以一个sk_buff的形式存储在内核内存中。sk_buff 是一种在内核网络堆栈中广泛使用的数据结构。附加在 TC 子系统中的 eBPF 程序会将 sk_buff 结构的指针作为上下文参数进行处理。

你可能想知道为什么 XDP 程序不也使用相同的结构作为上下文。答案是 XDP 挂钩发生在网络数据到达网络堆栈之前，也就是在 sk_buff 结构被设置之前。

TC 子系统旨在进行网络流量的调度。例如，你可能希望限制每个应用程序可用的带宽，以使大家都有公平的机会。但是，当考虑调度单个数据包时，带宽并不是一个非常有意义的术语，因为它用于表示发送或接收的数据的平均量。某个应用程序可能会具有很大的突发性，另一个应用程序可能对网络延迟非常敏感，因此 TC 可以更精细地控制数据包的处理和优先级。

在 TC 中引入 eBPF 程序是为了对 TC 内部使用的算法进行自定义控制。但是，由于具备了操作、丢弃或重定向数据包的能力，附加在 TC 内部的 eBPF 程序也可以用作构建复杂网络行为的基本组件。

在网络堆栈中，给定的网络数据流向两个方向：入口（从网络接口进入）或出口（向网络接口出去）。eBPF 程序可以在任一方向上进行附加，并且只会影响该方向上的流量。与 XDP 不同，可以附加多个 eBPF 程序，它们将按顺序进行处理。

传统的流量控制分为分类器（classifiers）和动作（actions）两部分。分类器基于某些规则对数据包进行分类，而动作根据分类器的输出确定如何处理数据包。一系列分类器可以作为 qdisc（队列规则）的一部分进行定义。

eBPF 程序作为分类器附加在 TC 中，但它们也可以在同一个程序中确定要执行的动作。动作由程序的返回代码指示（其值在 linux/pkt_cls.h 中定义）：

- TC_ACT_SHOT 告诉内核丢弃该数据包。
- TC_ACT_UNSPEC 的行为就像 eBPF 程序未对此数据包进行处理一样（如果有下一个分类器，则将传递给该分类器）。
- TC_ACT_OK 告诉内核将数据包传递给堆栈中的下一层。
- TC_ACT_REDIRECT 将数据包发送到不同网络设备的入口或出口路径。

现在让我们看一些可以附加在 TC 中的简单程序示例。第一个示例仅生成一行跟踪信息，然后告诉内核丢弃该数据包：

```
int tc_drop(struct __sk_buff *skb) {
    bpf_trace_printk("[tc] dropping packet\n");
    return TC_ACT_SHOT;
```

```
}
```

考虑如何仅丢弃一部分数据包。这个示例丢弃 ICMP（ping）请求数据包，非常类似于前面看到的 XDP 示例：

```
int tc(struct __sk_buff *skb) {
    void *data = (void *)(long)skb->data;
    void *data_end = (void *)(long)skb->data_end;
    if (is_icmp_ping_request(data, data_end)) {
        struct iphdr *iph = data + sizeof(struct ethhdr);
        struct icmphdr *icmp = data + sizeof(struct ethhdr) +
            sizeof(struct iphdr);
        bpf_trace_printk("[tc] ICMP request for %x type %x\n
           ", iph->daddr,
        icmp->type);
        return TC_ACT_SHOT;
    }
    return TC_ACT_OK;
}
```

sk_buff 结构体具有指向数据包数据起始和结束的指针，非常类似于 xdp_md 结构体，数据包解析的过程也非常相似。同样，为了验证通过，你必须明确检查对数据的任何访问是否在 data 和 data_end 之间的范围内。

你可能想知道为什么在 XDP 中已经实现有类似功能的情况下，还要在 TC 层实现类似的功能。一个很好的原因是，你可以将 TC 程序用于出口流量，而 XDP 只能处理入口流量。另一个原因是，因为 XDP 是在数据包到达时立即触发的，此时与数据包相关的 sk_buff 内核数据结构还不存在。如果 eBPF 程序对 sk_buff 感兴趣或希望操作内核为该数据包创建的 sk_buff，TC 附加点才是合适的选择。

为了更好地了解 XDP 和 TC eBPF 程序之间的差异，请阅读 Cilium 项目中的 “Program Types” 部分的BPF 和 XDP 参考指南。

现在来看一个不仅丢弃特定数据包的示例。这个示例识别收到的 ping 请求，并以 ping 响应进行回应：

```
int tc_pingpong(struct __sk_buff *skb) {
    void *data = (void *)(long)skb->data;
    void *data_end = (void *)(long)skb->data_end;
    if (!is_icmp_ping_request(data, data_end)) { // 1
    return TC_ACT_OK;
    }
```

```
        struct iphdr *iph = data + sizeof(struct ethhdr);
        struct icmphdr *icmp = data + sizeof(struct ethhdr) +
            sizeof(struct iphdr);

        swap_mac_addresses(skb); // 2
        swap_ip_addresses(skb);

        // Change the type of the ICMP packet to 0 (ICMP Echo
            Reply)
        // (was 8 for ICMP Echo request)
        update_icmp_type(skb, 8, 0); // 3

        // Redirecting a clone of the modified skb back to the
            interface
        // it arrived on
        bpf_clone_redirect(skb, skb->ifindex, 0); // 4

        return TC_ACT_SHOT; // 5
    }
```

1. is_icmp_ping_request() 函数解析数据包并检查是否为 ICMP 消息，且具体为回显（ping）请求。

2. 由于该函数将向发送者发送响应，因此需要交换数据包中的源地址和目标地址，还包括更新 IP 头部的校验和。

3. 通过更改 ICMP 头部中的类型字段，将数据包转换为回显响应。

4. 辅助函数 bpf_clone_redirect() 通过接收数据包的接口（skb->ifindex）发送数据包的克隆。

5. 由于辅助函数在发送响应前克隆了数据包，原始数据包应该被丢弃。

在正常情况下，ping 请求将由内核的网络栈稍后处理，但这个简单的示例演示了如何用 eBPF 实现替代网络功能。

如今，许多网络功能都由用户空间服务处理，但只要可以用 eBPF 程序替代，就有可能获得更高的性能。在内核中处理的数据包无需完成其在网络栈中的整个传输过程；它无需转入用户空间进行处理，响应也无需再次返回内核。而且，这两者可以并行运行，eBPF 程序可以对任何需要复杂处理而无法处理的数据包返回 TC_ACT_OK，以便将其作为常规方式传递给用户空间程序。

对我来说，这是在 eBPF 中实现网络功能的一个重要方面。随着 eBPF 平台的发展（例如，允许一百万条指令的较新内核），可以在内核中实现越来越复杂的网络功能。尚未在 eBPF 中实现的部分功能仍然可以由内核中的传统栈或用户空间来处理。随着时间的推

移，越来越多的功能可以从用户空间移入内核，而 eBPF 的灵活性和动态性意味着你无需等待它们成为内核的一部分，你可以立即加载 eBPF 实现。

首先，来看下 eBPF 能实现的另一个功能：检查加密流量。

8.5 数据包加密和解密

如果一个应用程序使用加密来保护发送或接收的数据，那么在加密前或解密后的某个时刻，数据将处于明文状态。回想一下，eBPF 可以在机器上的几乎任何地方附加程序，因此如果你可以钩入数据传递的点，并且数据尚未加密，或者刚刚解密完成，那么你的 eBPF 程序就可以观察到明文数据。与传统的 SSL 检查工具不同，你不需要提供任何证书来解密。

在许多情况下，应用程序会使用像 OpenSSL 或 BoringSSL 这样的用户空间库来加密数据。在这种情况下，当数据到达套接字时，它已经被加密，而套接字是用户空间和内核之间的边界。如果你想要追踪这些数据得未加密形式，可以在用户空间代码的适当位置附加一个 eBPF 程序。

8.5.1 用户空间的 SSL 库

追踪加密数据包解密内容的一种常见方法是钩入对用户空间库（如 OpenSSL 或 BoringSSL）的调用。使用 OpenSSL 的应用程序通过调用一个名为 SSL_write() 的函数发送要加密的数据，并使用 SSL_read() 函数从网络接收到的加密数据中检索明文数据。通过使用 uprobes 将 eBPF 程序钩入到这些函数中，eBPF 程序可以在明文状态下观察使用此共享库的任何应用程序的数据，无论是在加密之前还是解密之后。而且不需要任何密钥，因为应用程序已经提供了明文。

Pixie 项目中有一个相当简单的示例，名为 openssl-tracer(这篇文章也是讲的这个话题)，在其中的一个名为 openssl_tracer_bpf_funcs.c 的文件中包含了 eBPF 程序的代码。以下是该代码的一部分，用于使用性能缓冲区将数据发送到用户空间：

```
static int process_SSL_data(struct pt_regs* ctx, uint64_t id,
    enum
ssl_data_event_type type, const char* buf) {
...
  bpf_probe_read(event->data, event->data_len, buf);
  tls_events.perf_submit(ctx, event, sizeof(struct
     ssl_data_event_t));

  return 0;
}
```

可以看到，使用辅助函数 bpf_probe_read() 将 buf 中的数据读入到一个事件结构中，然后将该事件结构提交到一个性能缓冲区。

如果这些数据被发送到用户空间，可以合理地假设这些数据是未加密的格式。那么这个数据缓冲区是从哪里获取的呢？通过查看 process_SSL_data() 函数的调用位置可以找到答案。这个函数在两个地方被调用：一个是用于读取数据，另一个是用于写入数据。图 8-4 说明了在读取以加密形式到达该机器的数据时发生了什么。

在读取数据时，你向 SSL_read() 函数提供一个指向缓冲区的指针，当函数返回时，该缓冲区将包含未加密的数据。与 kprobes 类似，函数的输入参数（包括缓冲区指针）只在附加到入口点的 uprobe 中可用，因为它们所在的寄存器在函数执行期间可能会被覆盖。但是在函数退出之前，数据在缓冲区中是不可用的，你可以使用 uretprobe 在函数退出时读取它。

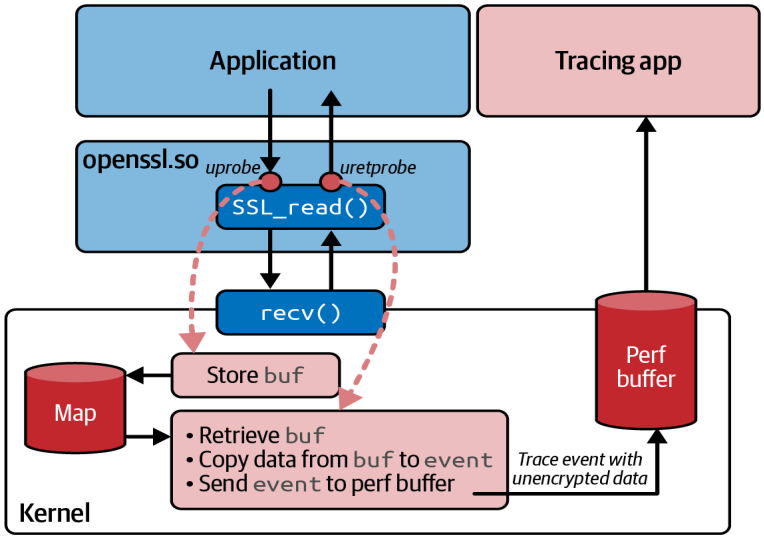

图 8.4: eBPF 程序通过 uprobes 探针挂钩到 SSL_read() 函数的入口和出口，以便从缓冲指针中读取未加密数据

因此，这个示例遵循了 kprobes 和 uprobes 的常见模式，如图 8-4 所示，入口探针使用一个映射临时存储输入参数，退出探针可以从中检索这些参数。让我们看一下附加到 SSL_read() 入口的 eBPF 程序的代码。

```
// Function signature being probed:
// int SSL_read(SSL *s, void *buf, int num)
int probe_entry_SSL_read(struct pt_regs* ctx) {
    uint64_t current_pid_tgid = bpf_get_current_pid_tgid();
    ...
    const char* buf = (const char*)PT_REGS_PARM2(ctx); // 1
    active_ssl_read_args_map.update(&current_pid_tgid, &buf); // 2
    return 0;
}
```

1. 如注释所述，缓冲区指针是作为第二个参数传递给 SSL_read() 函数的，而这个指针将附加到该函数上。PT_REGS_PARM2 宏从上下文中获取该参数。

2. 缓冲区指针存储在一个哈希映射中，其中键是当前进程和线程 ID，在开始时使用辅助函数 bpf_get_current_pid_tgid() 获取。

下面是相应的退出探针的程序：

```
int probe_ret_SSL_read(struct pt_regs* ctx) {
    uint64_t current_pid_tgid = bpf_get_current_pid_tgid();
    ...
    const char** buf = active_ssl_read_args_map.lookup(&
       current_pid_tgid); // 1
    if (buf != NULL) {
        process_SSL_data(ctx, current_pid_tgid, kSSLRead, *
           buf); // 2
    }
    active_ssl_read_args_map.delete(&current_pid_tgid); // 3
    return 0;
}
```

1. 获取当前进程或线程 ID 后，使用它作为键从哈希映射中检索缓冲区指针。

2. 如果不是空指针，则调用 process_SSL_data() 函数，该函数会将缓冲区中的数据通过 perf 缓冲区发送到用户空间。

3. 清理哈希映射中的条目，因为每个进入调用都应该与一个退出调用配对。

这个示例展示了如何追踪用户空间应用程序发送和接收的加密数据的明文。追踪本身是附加到用户空间库上的，但并不保证每个应用程序都使用特定的 SSL 库。BCC 项目中还包括一个名为sslsniff的实用工具，它还支持 GnuTLS 和 NSS。但如果某个应用程序使用其他加密库（甚至，选择自己编写加密算法），uprobes 就无法找到正确的挂钩位置，这些跟踪工具也就无法工作。

甚至有更常见的原因导致基于 uprobes 的方法可能无法成功。与内核不同（每台虚拟机只有一个内核），用户空间库代码可以有多个副本。如果使用容器，每个容器很可能有自己的库依赖。你可以在这些库中挂接 uprobes，但必须为要追踪的特定容器标识出正确的副本。另一种可能是，应用程序可能使用静态链接，而不是共享的动态链接库，这样它就是一个独立的可执行文件。

8.6 eBPF 和 Kubernetes 网络

虽然本书不涉及 Kubernetes，但 eBPF 在 Kubernetes 网络中的广泛应用是展示如何使用该平台定制网络堆栈的绝佳示例。

在 Kubernetes 环境中，应用程序被部署在 Pod 中。每个 Pod 是一个由一个或多个容

器组成的组，它们共享内核命名空间和控制组，将 Pod 与其他 Pod 和运行它们的主机隔离开来。

特别是对于本章的目的，每个 Pod 通常有自己的网络命名空间和 IP 地址。这意味着内核为该命名空间维护了一组网络堆栈结构，与主机和其他 Pod 隔离开来。如图 8-5 所示，Pod 通过虚拟以太网连接与主机相连，并分配了自己的 IP 地址。

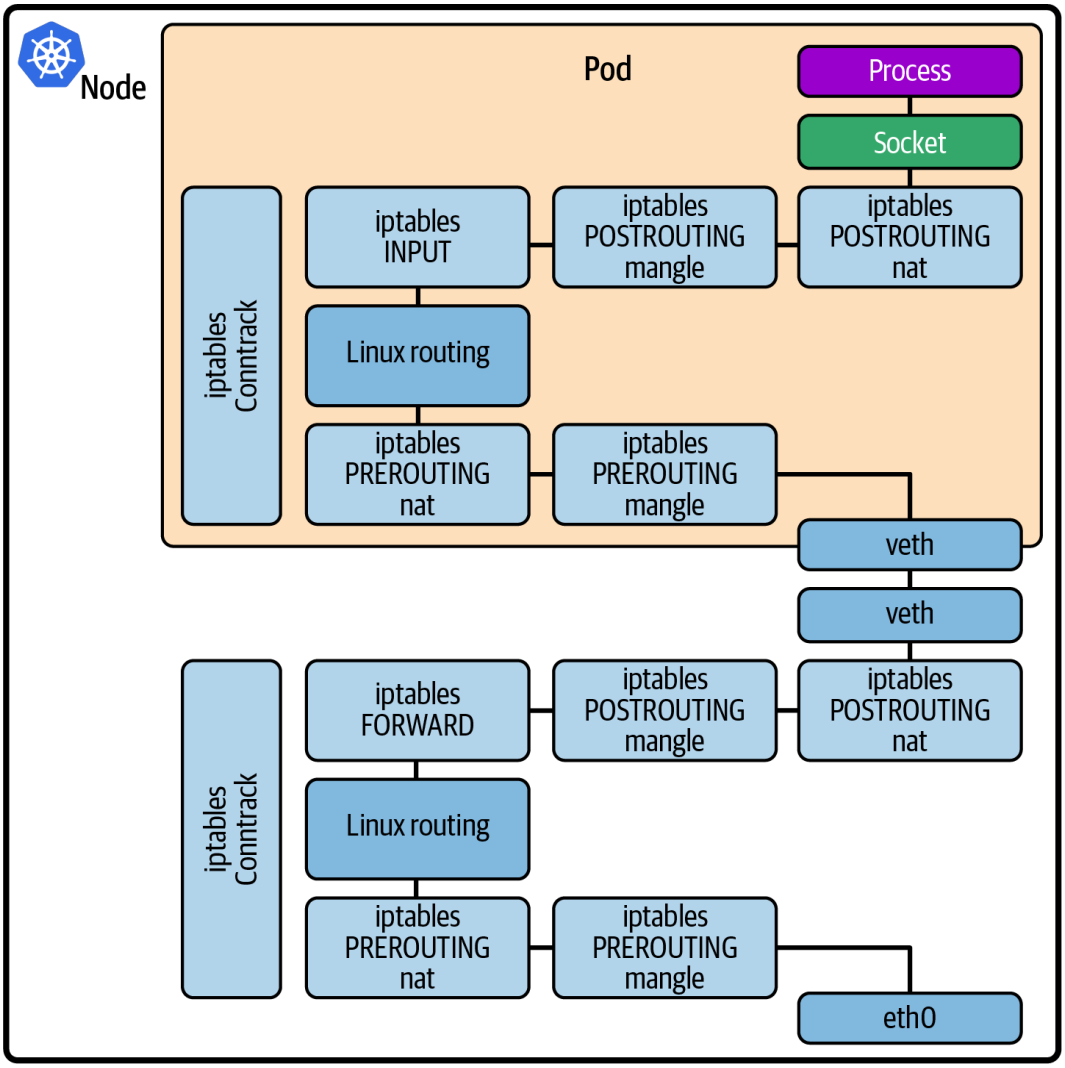

图 8.5: Kubernetes 中的网络路径

从图 8-5 可以看出，从外部机器发送到应用程序 Pod 的数据包必须通过主机上的网络堆栈，跨越虚拟以太网连接，进入 Pod 的网络命名空间，然后再次通过网络堆栈到达应用程序。

这两个网络堆栈在同一个内核中运行，因此数据包实际上会经过相同的处理两次。网络数据包经过的代码越多，延迟就越高，因此如果可能缩短网络路径，很可能会带来性能改进。

像 Cilium 这样基于 eBPF 的网络解决方案可以跨接到网络堆栈，覆盖内核的网络行为，如图 8-6 所示。

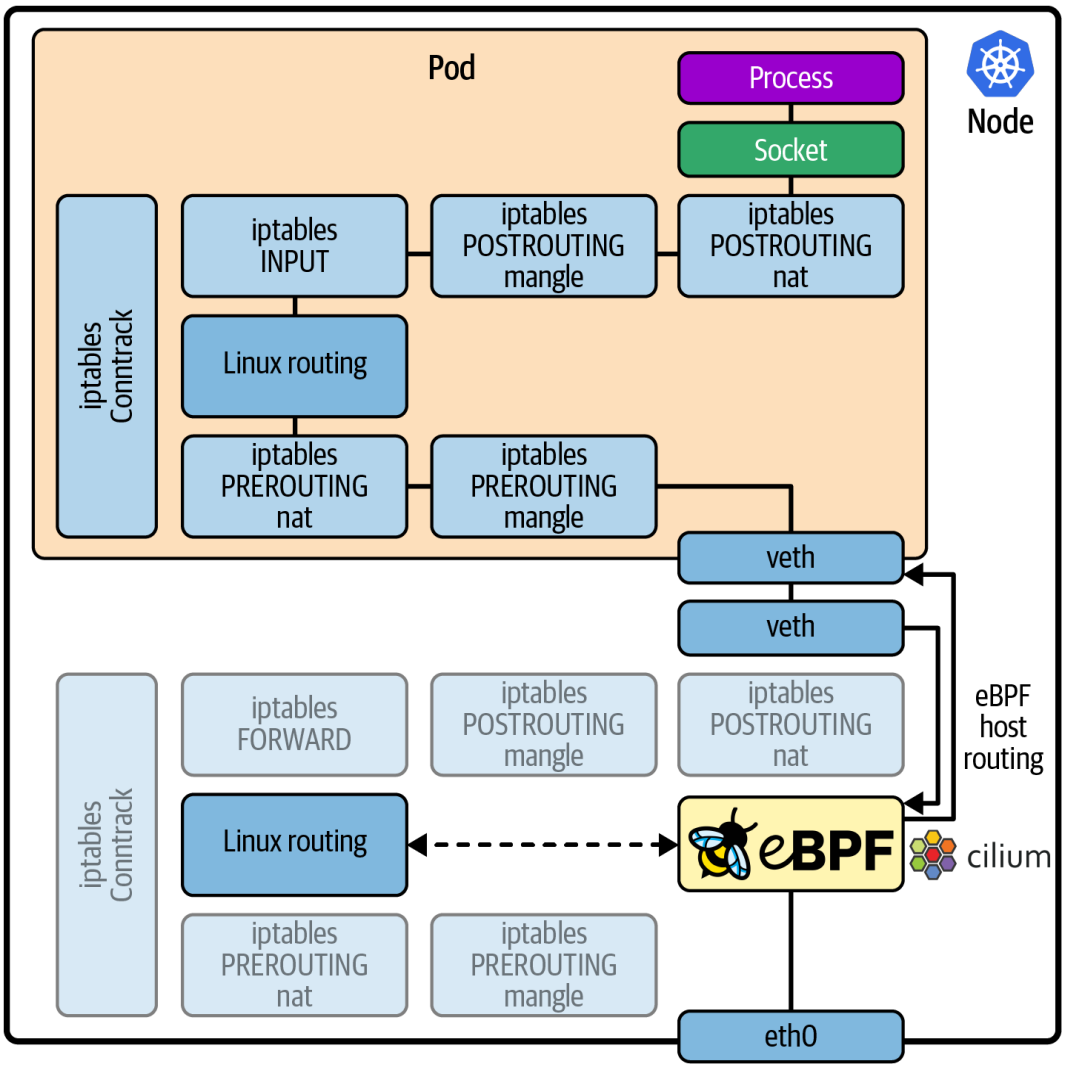

图 8.6: 使用 eBPF 绕过 iptables 和 conntrack 处理

特别是，eBPF 使得可以用更高效的方案来替代 iptables 和 conntrack 并管理网络规则和连接跟踪。现在来讨论下为什么这会在 Kubernetes 中带来显著的性能提升。

8.6.1 避免 iptables

Kubernetes 中的 kube-proxy 组件实现了负载均衡行为，允许多个 pod 处理对服务的请求。这是通过使用 iptables 规则实现的。Kubernetes 通过容器网络接口（CNI）为用户提供了选择网络解决方案的权利。一些 CNI 插件使用 iptables 规则来在 Kubernetes 中实现 L3/L4 网络策略，即 iptables 规则指示是否丢弃不符合网络策略的数据包。

虽然 iptables 在传统的（容器之前）网络中是有效的，但在 Kubernetes 中使用时存在缺点。在容器中，pod 及其 IP 地址动态创建和销毁，每添加或移除一个 pod，iptables 规则必须完全重写，这对性能产生了极大得影响（Haibin Xie 和 Quinton Hoole 在 2017 年的 KubeCon 上的一次演讲描述了为 2 万个服务的 iptables 规则进行单个规则更新可能需要五个小时）。

iptables 更新不是唯一的性能问题：查找需要在表中进行线性搜索，这是一个 O(n) 的操作，随着规则数量的增加而线性增长。

Cilium 使用 eBPF 哈希表映射来存储网络策略规则、连接跟踪和负载均衡器查找表，可以替代 kube-proxy 中的 iptables。在哈希表中查找条目和插入新条目都是近似 O(1) 的操作，规模上更加高效。你可以在 Cilium博客中阅读有关这方面的经过基准测试的性能提升。在同一篇文章中，你将看到另一个 CNI 插件 Calico eBPF 模式下也实现了更好的性能。eBPF 为可扩展、动态的 Kubernetes 部署提供了实现性能最佳的方案。

8.6.2 协调的网络程序

Cilium 的网络实现是复杂的，不能以一个单独的 eBPF 程序来完成。如图 8-7 所示，它提供了多个不同的 eBPF 程序，钩入到内核及其网络栈的不同部分。

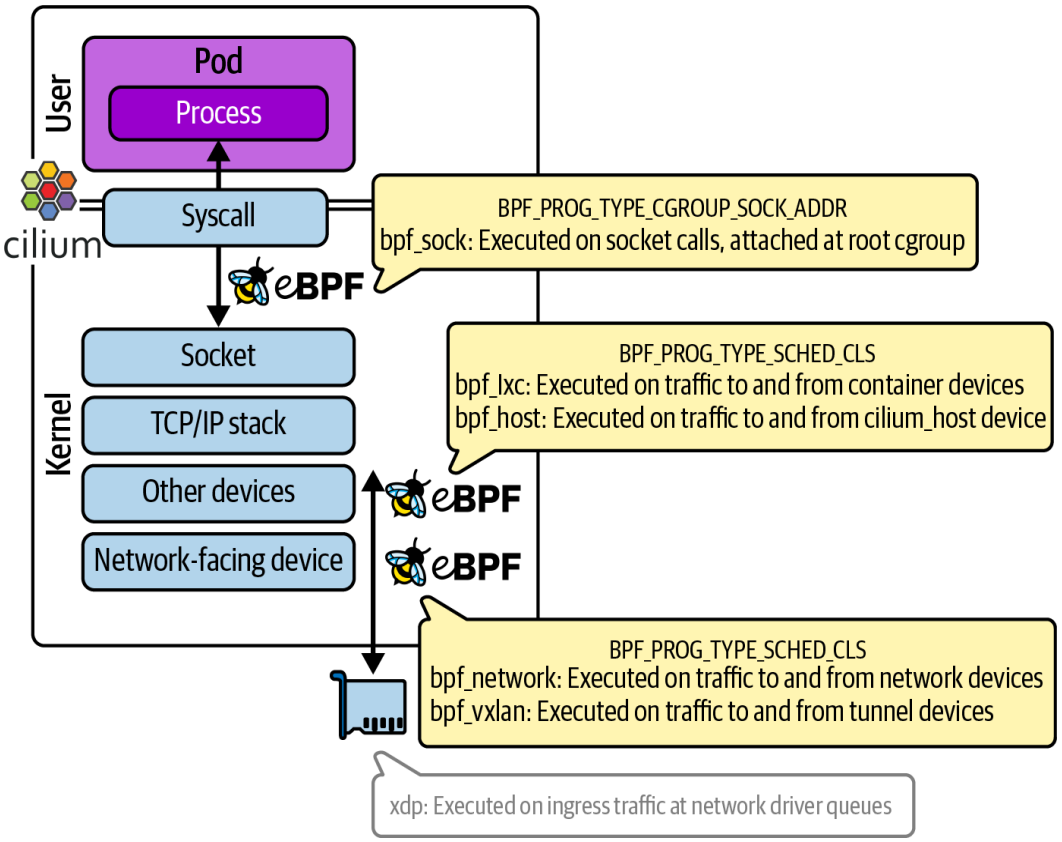

图 8.7: Cilium 由多个协调的 eBPF 程序组成，这些程序钩入内核的不同点

作为一个通用原则，Cilium 尽早拦截流量，以缩短每个数据包的处理路径。从应用程序 pod 流出的消息在套接字层被拦截，尽可能靠近应用程序。来自外部网络的入站数据包使用 XDP 进行拦截。但是其他附加的连接点呢?

Cilium 支持适用于不同环境的不同网络模式。详细描述超出了本书的范围（你可以在Cilium.io上找到更多信息），但我在这里简要概述一下，以便你了解为什么会有这么多不同的 eBPF 程序!

有一种简单的扁平网络模式，Cilium 从相同的 CIDR 中为集群中的所有 pod 分配 IP 地址，并直接在它们之间路由流量。还有几种不同的隧道模式，其中针对不同节点上的 pod 的流量被封装在一个寻址到目标节点 IP 地址的消息中，并在目标节点上解封装以进行最后一跳到 pod。根据数据包是否路由到本地容器、本地主机、本网络上的另一主机或隧道，调用不同的 eBPF 程序来处理流量。

在图 8-7 中，你可以看到多个 TC 程序，用于处理与不同设备之间的流量。这些设备代表可能的不同真实和虚拟网络接口，数据包可在其中流动：

- pod 网络的接口（pod 与主机间的虚拟以太网连接的一端）
- 网络隧道的接口
- 主机上的物理网络设备的接口
- 主机自己的网络接口

如果你对数据包在 Cilium 中的流动方式感兴趣，Arthur Chiao 撰写了一篇详细且有趣的博文：“Cilium 中的数据包生命周期：发现 Pod 到 Service 的流量路径和 BPF 处理逻辑”。内核中的不同 eBPF 程序通过使用 eBPF 映射和附加到网络数据包的元数据进行通信。这些程序不仅是将数据包路由到目的地，还根据网络策略丢弃数据包，就像你之前看到的示例一样。

8.6.3 网络策略执行

在本章的开头，你看到了 eBPF 程序可以丢弃数据包，这意味着它们将无法到达目的地。这是网络策略执行的基础，无论我们是考虑传统环境还是云原生防火墙，概念上基本是一样的。一个策略根据数据包的源或目的地信息决定是否丢弃该数据包。

在传统环境中，IP 地址长期分配给特定的服务器，但在 Kubernetes 中，IP 地址是动态变化的，今天分配给特定应用 Pod 的地址明天可能被完全不同的应用重用。这就是为什么传统防火墙在云原生环境中并不十分有效的原因。每次 IP 地址发生变化时，手动重新定义防火墙规则是不切实际的。

相反，Kubernetes 支持 NetworkPolicy 资源的概念，该资源基于应用于特定 Pod 的标签定义防火墙规则，而不是基于其 IP 地址。虽然资源类型是 Kubernetes 的本地资源，但它并非由 Kubernetes 本身实现。相反，这个功能委托给所使用的 CNI 插件。如果选择的 CNI 不支持 NetworkPolicy 资源，那么配置的任何规则都将忽略。另一方面，CNIs 可以自由配置自定义资源，允许进行比原生 Kubernetes 定义更复杂的网络策略配置。例如，Cilium 支持基于 DNS 的网络策略规则，因此可以根据 DNS 名称而不是 IP 地址来定义

是否允许流量通过。还可以为各种第 7 层协议定义策略，例如允许或拒绝对特定 URL 的 HTTP GET 调用，但不允许 POST 调用。

Isovalent 的免费实践实验室"Getting Started with Cilium" 将引导你在第 3/4 层和第 7 层定义网络策略。另一个非常有用的资源是networkpolicy.io上的网络策略编辑器，它以可视化方式呈现网络策略的效果。

正如我在本章前面讨论的那样，可以使用 iptables 规则来丢弃流量，这是一些 CNI 用于实现 Kubernetes NetworkPolicy 规则的方法。Cilium 使用 eBPF 程序来丢弃不符合当前规则的流量。你在本章前面见过丢弃数据包，相信你已经有了一个大致的思维模型，了解其工作原理。

Cilium 使用 Kubernetes 身份来确定特定网络策略规则是否适用。就像标签在 Kubernetes 中定义哪些 Pod 属于一个服务一样，标签也定义了 Cilium 对 Pod 的安全标识。通过使用这些服务标识作为索引，eBPF 哈希表可以实现非常高效的规则查找。

8.6.4 加密连接

许多组织需要在应用程序之间加密流量以保护其部署和用户数据的要求。可以在每个应用程序中编写代码来实现这一目标，以确保建立安全连接，通常是使用互信的传输层安全性（mTLS）作为 HTTP 或 gRPC 连接的基础。建立连接需要首先确定连接两端的应用程序身份（通过交换证书来实现），然后加密之间的数据。

在 Kubernetes 中，可以将这个要求从应用程序转移到服务网格层或底层网络本身。本书无法全面讨论服务网格，但你可能会对我在 The New Stack 上撰写的一篇文章《How eBPF Streamlines the Service Mesh》感兴趣。现在，让我们集中讨论网络层以及如何通过 eBPF 将加密要求推入内核。

在 Kubernetes 集群中，确保流量在集群内部加密的最简单方法是使用透明加密。称为" 透明"，因为它完全在网络层进行，从运营的角度来看非常轻量级。应用程序本身不需要知道加密的存在，也不需要建立 HTTPS 连接；而且这种方法不需要在 Kubernetes 上运行任何额外的基础设施组件。

常用的两种内核加密协议是 IPsec 和 WireGuard(R)，它们都得到了 Cilium 和 Calico CNI 在 Kubernetes 网络中的支持。讨论这两种协议的差异超出了本书的范围，但关键是它们在两台机器之间建立了一个安全隧道。CNI 可以选择通过这个安全隧道连接 Pod 的 eBPF 端点。

Cilium 博客上有一篇很好的文章，介绍了 Cilium 如何使用 WireGuard(R) 和 IPsec 在节点之间加密流量，还简要概述了两种协议的性能特点。

安全隧道是使用两端节点的身份进行设置的。这些身份由 Kubernetes 进行管理，因此运营人员的管理负担很小。对于许多情况来说，这已经足够了，因为它确保了集群中所有网络流量都是加密的。透明加密也可以与使用 Kubernetes 身份来管理集群中不同端点间流量的 NetworkPolicy 一起使用，而无需进行修改。

一些组织在多租户环境中运行，需要强大的多租户边界，且每个应用程序端点的身份验证是必不可少的。在每个应用程序内部处理这点是一个很重得负担，因此最近已将其转

移到服务网格层，但这需要部署一整套额外的组件，增加资源消耗、延迟和操作复杂性。

现在，eBPF 正在启用一种新方法，它建立在透明加密的基础上，但使用 TLS 进行初始证书交换和端点认证，以便身份可以代表单个应用程序，而不仅仅是它们运行的节点，如图 8-8 所示。

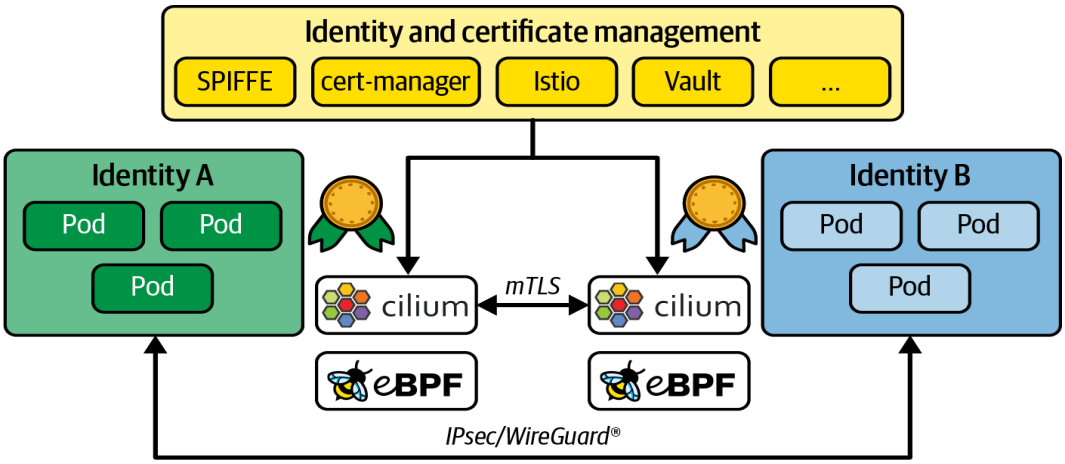

图 8.8: 认证的应用程序身份间的透明加密

一旦进行了身份验证步骤，内核中的 IPsec 或 WireGuard(R) 就可用于加密这些应用程序间的流量。这样做有许多优点，它允许第三方证书和身份管理工具（如 cert-manager 或 SPIFFE/SPIRE）处理身份验证，而网络负责加密，因此对应用程序来说，这一切都是完全透明的。Cilium 支持通过 SPIFFE ID 而不仅仅是 Kubernetes 标签来指定端点的 NetworkPolicy 定义。而且，最重要的是，这种方法可用于在 IP 数据包中传输的任何协议。这是对 mTLS 的重大扩展，后者仅适用于基于 TCP 的连接。

本书中的篇幅不足以深入探讨 Cilium 的所有内部机制，但我希望本节能帮助你理解 eBPF 是构建复杂网络功能的强大平台，如功能齐全的 Kubernetes CNI。

8.7 总结

在本章中，你看到了 eBPF 程序连接到网络栈的各种不同层。我展示了一些基本的数据包处理示例，希望这些示例让你了解了 eBPF 如何创建强大的网络功能。你还看到了一些实际的网络功能示例，包括负载均衡、防火墙、安全缓解和 Kubernetes 网络。

8.8 练习和更多阅读材料

以下是一些了解 eBPF 的各种网络功能的方法：

1. 修改示例 XDP 程序 ping()，使其针对 ping 响应和 ping 请求生成不同的跟踪消息。ICMP 头在网络数据包中紧随 IP 头（就像 IP 头紧随以太网头一样）。你可能需

要使用 linux/icmp.h 中的 struct icmphdr，并查看类型字段是否显示 ICMP_ECHO 或 ICMP_ECHOREPLY。

2. 如果想深入研究 XDP 编程，推荐使用 xdp-project 的xdp-tutorial。
3. 使用 BCC 项目的sslsniff查看加密流量的内容。
4. 使用 Cilium 网站链接的教程和实验室来探索Cilium。
5. 使用networkpolicy.io上的编辑器来可视化 Kubernetes 部署中网络策略的影响。

第九章　eBPF 用于安全

在前面的章节中，你已经了解到 eBPF 如何用于观测系统事件并向用户空间报告。在本章中，你将了解如何基于事件检测的概念构建基于 eBPF 的安全工具，用于检测甚至防止恶意活动。我将从帮助你理解安全与其他类型的观测性工具之间的区别开始。本章代码在 GitHub仓库的 chapter9 目录中。

9.1　安全可观测性需要策略和上下文

安全工具与仅报告事件的观测性工具间的区别在于，安全工具需要区分正常事件和可能发生恶意活动的事件，它本身具有某种区分策略。例如，一个程序将数据写入普通文件是正常工作，如/home/<username>/<filename>，这种活动从安全角度来说并不是你要关注的。然而，如果该程序将数据往 Linux 敏感文件写，你肯定希望收到通知。例如，它不太可能需要修改/etc/passwd 中的密码，所以也不应该往这个文件写。

策略必须考虑到系统在完全正常运行时的行为以及预期的错误行为。例如，如果磁盘空间已满，程序可能会开始发送告警。这些告警不应被视为安全事件，尽管告警是不寻常的，但它们的出现并不可疑。考虑到各种错误行为以创建有效的策略是十分困难的。

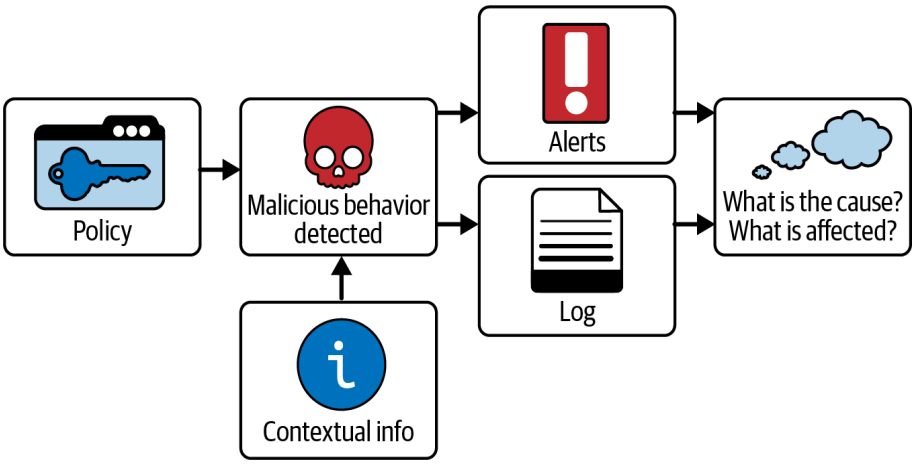

图 9.1: 在安全观测性中，除了超出策略事件检测外，还需要上下文信息

定义预期行为和非预期行为就是策略的工作。安全工具将活动与策略进行比较，并在

活动超出策略范围时采取一些安全措施，使其变得可疑。安全措施通常包括生成安全事件日志，并发送到安全信息事件管理（SIEM）平台，还可能导致向负责调查员发出告警。

调查员可以获得的上下文信息越多，他们就越有可能找到事件的根因，并确定是否是一次攻击，受影响的组件有哪些，攻击发生的方式、时间以及谁负责。如图 9-1 所示，能够回答此类问题将使工具从仅仅是记录日志升级为“安全观测性”的工具。

让我们探索一些使用 eBPF 程序来检测和执行安全事件的方法。正如你所了解的，eBPF 程序可以附加到各种事件上，而多年来一组常用于安全的事件是系统调用（syscalls）。我们将从 syscalls 开始，尽管 syscalls 可能不是使用 eBPF 实现安全工具的最有效方法。在本章后面，我们将看到一些更新且更复杂的方法。

9.2 使用系统调用进行安全事件监控

系统调用（syscalls）是用户空间应用程序与内核之间的接口。如果限制应用程序可用的系统调用集合，那么它的操作范围将受到限制。例如，阻止应用程序进行 open*() 系列的系统调用，那么它将无法打开文件。如果有一个程序，你不希望它打开文件，你可能就希望创建这种限制，以便即使程序被入侵，它也无法恶意访问文件。如果你在过去几年中使用过 Docker 或 Kubernetes，那么很可能你已经遇到过使用 BPF 来限制系统调用的安全工具：seccomp。

9.2.1 Seccomp

seccomp 这个名字是“SECure COMPuting”的缩写。在其原始或“严格”的形式中，seccomp 用于将进程可用的系统调用集合限制为非常小的子集：read()、write()、_exit() 和 sigreturn()。严格模式的目的是允许用户运行不受信任的代码（可能是从互联网上下载来的程序）而不会导致该代码执行恶意操作。

严格模式非常严格，许多程序需要使用更大范围的系统调用集合，但这并不意味着它们需要系统内 400 多个系统调用。因此，允许更灵活的方法来限制程序可用的系统调用集合是十分有价值的。这就是我们在容器领域内遇到 seccomp 版本的理论依据，更准确地称为 seccomp-bpf。与允许的系统调用子集不同，这种 seccomp 模式使用 BPF 代码来过滤允许和禁止的系统调用。

在 seccomp-bpf 中，一组 BPF 指令被加载为过滤器。每次系统调用时，都会触发此过滤器。过滤器代码可以访问传递给系统调用的参数，以便根据系统调用本身和传递给它的参数做出决策。结果可能是以下几种操作：

- 允许系统调用继续执行
- 向用户空间的程序返回错误代码
- 终止此线程
- 通知用户空间的程序（seccomp-unotify）（从内核版本 5.0 开始）

如果你想尝试编写自己的 BPF 过滤器代码，Michael Kerrisk 的网站上有一些很好的例子，网址：https://oreil.ly/cJ6HL。

一些传递给系统调用的参数是指针，而 seccomp-bpf 中的 BPF 代码无法解引用这些指针。这限制了 seccomp 配置文件的灵活性，因为它只能在进程启动时应用，并且无法修改应用于给定应用程序进程的配置文件。

你可能已经在不编写 BPF 代码的情况下使用过 seccomp-bpf，因为代码通常是从可读性较好的 seccomp 配置文件派生而来。Docker 的默认配置文件就是一个很好的例子。这是一个通用配置文件，可用于几乎任何正常的容器。这意味着它允许大多数系统调用，只禁止了一些在任何应用程序中都不太可能出现的系统调用，比如 reboot()。

根据 Aqua Security 的说法，大多数容器使用的 syscall 数量在 40 到 70 之间。为了提高安全性，最好使用更受限制的配置文件，针对每个特定程序设置其可用的 syscall。

9.2.2 生成 Seccomp 配置文件

如果你问一般的程序员它们的程序使用了哪些 syscall，他们的反应可能是一片茫然。这并不是要你去耍笑他们，只是因为大多数程序员使用的编程语言提供了高层抽象，远离 syscall 的细节。例如，他们可能知道应用程序打开了哪些文件，但很少能告诉你是使用了 open() 还是 openat() 系统调用。因此，如果要求程序员自己手工创建合适的 seccomp 配置文件，可能就不太可能。

自动化是未来的发展方向：应该用工具来记录程序使用的 syscall 集合。在早期，seccomp 配置文件通常使用 strace 编译，以收集程序调用的 syscall 集合。然而，在云原生时代，将 strace 指向特定的容器或 Kubernetes pod 并不容易。而且，生成的配置文件最好不仅仅是 syscall 列表，还应采用 Kubernetes 和 OCI 兼容的容器运行时所接受的 JSON 格式。有一些工具可以做到这点，它们使用 eBPF 来收集所有被调用的 syscall 的信息

- Inspektor Gadget包含一个 seccomp 分析器，可以为 Kubernetes pod 中的容器自定义生成seccomp 配置文件。
- Red Hat 创建了一个OCI 运行时钩子，作为 seccomp 分析器。

使用这些分析器，你需要运行程序一段任意时间，以生成包含所有可能合法调用的 syscall 的配置文件。如本章前面讨论的，此列表需要包括错误路径。如果程序由于所需的 syscall 被阻塞而无法在错误条件下正确运行，可能会导致更大的问题。而且，由于 seccomp 配置文件涉及的抽象级别低于大多数开发人员熟悉的层级，手动检查它们是否覆盖了所有正常的情况是困难的。

以 OCI 运行时钩子为例，将一个 eBPF 程序附加到 syscall_enter 原始跟踪点，并维护一个 eBPF 映射以跟踪已触发的 syscall。这个工具的用户空间部分是用 Go 语言编写的，并使用iovisor/gobpf库。

下面是 OCI 运行时钩子中加载 eBPF 程序并将其附加到跟踪点的代码：

```
src := strings.Replace(source, "$PARENT_PID", strconv.Itoa(
    pid), -1) // 1
m := bcc.NewModule(src, []string{})
defer m.Close()
```

```
...
enterTrace, err := m.LoadTracepoint("enter_trace") // 2
...
if err := m.AttachTracepoint("raw_syscalls:sys_enter",
   enterTrace); err != nil // 3
    {
        return fmt.Errorf("error attaching to tracepoint: %v
           ", err)
}
```

1. 这行代码做了一件有趣的事：它将 eBPF 源代码中的名为 $PARENT_PID 的变量替换为一个数字进程 ID。这是一种常见的模式，表明该工具将为每个被检测的进程加载单独的 eBPF 程序。

2. 这里，一个名为 enter_trace 的 eBPF 程序被加载到内核中。

3. enter_trace 程序被附加到跟踪点 raw_syscalls:sys_enter。这是在任何 syscall 进入时的跟踪点，你在之前的示例中已经见过。每当用户空间代码进行 syscall 时，都会触发这个跟踪点。

这个分析器使用附加到 sys_enter 的 eBPF 代码来跟踪已使用的 syscall 集合，并生成一个用于 seccomp 的 seccomp 配置文件。接下来我们将考虑的一类 eBPF 工具也会附加到 sys_enter，但它们使用 syscall 来跟踪程序的行为并将其与安全策略进行比较。

9.2.3 跟踪系统调用的安全工具

系统调用跟踪安全工具中最知名的工具是 CNCF 的Falco，它提供安全告警功能。默认情况下，Falco 被安装为一个内核模块，但也有一个基于 eBPF 的版本。用户可以定义规则来确定哪些事件与安全相关，并且当事件发生时，如果不符合规则中定义的策略，Falco 可以多种格式生成告警。

无论是内核模块驱动程序还是基于 eBPF 的驱动程序，它们都会附加到系统调用。如果你在GitHub 上查看 Falco的 eBPF 程序，你会看到类似下面的代码，它们将探针附加到原始系统调用的进入和退出点上（还有一些其他事件，如页面错误）：

```
BPF_PROBE("raw_syscalls/", sys_enter, sys_enter_args)

BPF_PROBE("raw_syscalls/", sys_exit, sys_exit_args)
```

由于 eBPF 程序可以动态加载并可检测到进程触发的事件，因此像 Falco 这样的工具可以将策略应用于正在运行的线上应用。用户可以修改应用的规则集，而无需修改程序或其配置。这与 seccomp 配置文件不同，后者必须在启动应用程序进程时应用。

不幸的是，使用系统调用入口点进行安全工具处理存在一个问题：“检查-使用时间窗口”（Time Of Check to Time Of Use，TOCTOU）问题。

eBPF 程序在系统调用的入口点触发时，它可以访问用户空间传递给系统调用的参数。如果这些参数是指针，内核需要将指向的数据复制到自己的数据结构中，然后再对该数据进行操作。如图 9-2 所示，在 eBPF 程序检查数据后，内核复制数据之前，存在一个攻击者修改数据的时间窗口。因此，实际操作的数据可能与 eBPF 程序捕获的数据不同（这对 Falco 产生了积极影响）。

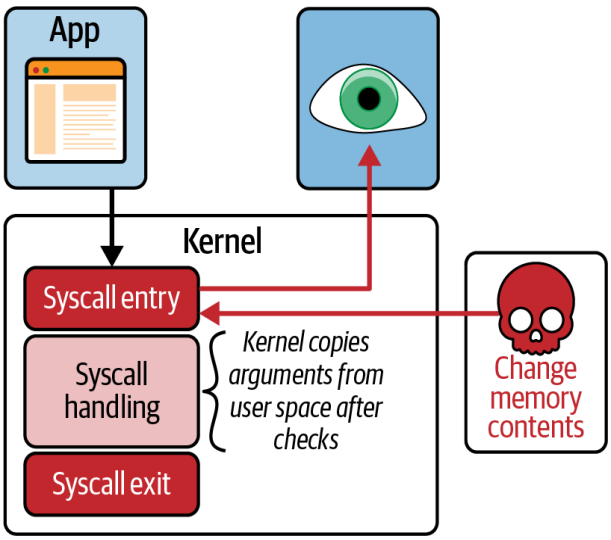

图 9.2: 在被内核访问之前，攻击者可以更改系统调用参数

如果不是因为 seccomp-bpf 中的一个事实，即程序不允许对用户空间指针进行解引用，那么同样的时间窗口也适用于 seccomp-bpf。

TOCTOU 问题适用于 seccomp_unotify，这是 seccomp 的一种最近添加的模式，可以向用户空间报告违规情况。seccomp_unotify 的 man 页明确指出 seccomp 用户空间通知机制不能用于实施安全策略！”

系统调用的入口点对于可观察性目的可能非常方便，但对于安全工具来说，这点不够。Sysmon for Linux工具通过附加到系统调用的入口和出口点来解决 TOCTOU 问题。一旦调用完成，它会查看内核的数据结构以获取准确的视图。例如，如果系统调用返回一个文件描述符，附加到退出点的 eBPF 程序可以查看相关进程的文件描述符表来获取文件描述符对应对象的正确信息。尽管这种方法可以产生与安全相关活动的准确记录，但它无法阻止危险操作发生，因为在进行检查时，系统调用已经完成。

为确保检查的是内核将要操作的数据，eBPF 程序应该附加到参数被复制到内核后发生的事件上。不幸的是，内核中没有一个通用的位置可以办到这点，因为数据在特定于系统调用的代码中以不同的方式处理。然而，存在一个定义明确的接口，eBPF 程序可以安全地附加到其中：Linux 安全模块（LSM）API。这需要一个相对较新的 eBPF 功能：BPF LSM。

9.3 BPF Linux 安全模块 (BPF LSM)

LSM 接口提供了一组钩子，钩子在内核即将对内核数据结构进行操作之前发生。由钩子调用的函数可以决定是否允许该操作继续进行。最初提供此接口是为了允许以内核模块的形式实现安全工具；BPF LSM通过扩展此接口使得 eBPF 程序可以附加到相同的钩子点，如图 9-3 所示。

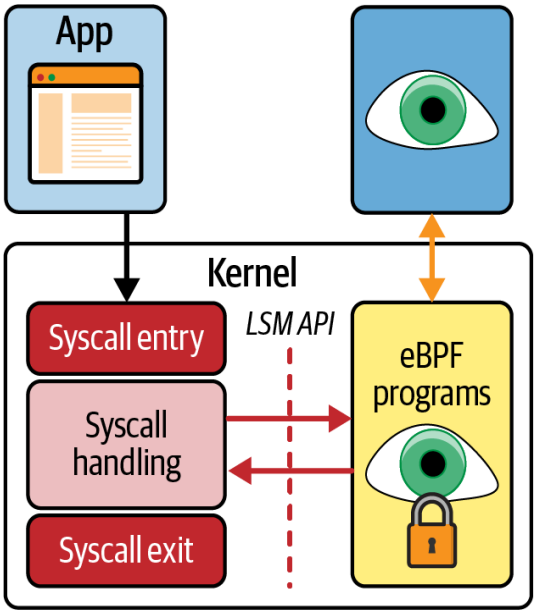

图 9.3: 使用 LSM BPF，eBPF 程序可以通过 LSM 钩子事件触发执行

LSM 钩子有数百个，并且在内核源代码中有很好的文档。需要明确的是，系统调用与 LSM 钩子间并没有一对一的关系，但是如果系统调用在安全上有潜力，处理该系统调用将触发一个或多个钩子。

以下是附加到 LSM 钩子的一个示例。此示例在处理 chmod 命令时调用（"chmod" 代表"change modes"，主要用于更改文件的访问权限）：

```
SEC("lsm/path_chmod")
int BPF_PROG(path_chmod, const struct path *path, umode_t mode)
{
    bpf_printk("Change mode of file name %s\n", path->dentry->d_iname);
    return 0;
}
```

该示例仅仅跟踪文件名并始终返回 0，但真实的实现中会利用这些参数来决定是否允许此模式更改。返回非零值将拒绝进行此更改的权限，因此内核将不会执行该更改。值得注意的是，在内核内部进行此类策略检查非常高效。

BPF_PROG() 函数的 path 参数是表示文件的内核数据结构，而 mode 参数是所需的新模式值。你可以从 path->dentry->d_iname 字段中看到正在访问的文件名。

LSM BPF 是在内核版本 5.7 中引入的，这意味着（至少在撰写本文时）它还没有在许多支持的 Linux 发行版中得到广泛使用，但我预计在未来几年内，许多厂商将开发利用这个接口的安全工具。在 LSM BPF 被广泛应用之前，还有另一种可能的方法，即 Cilium Tetragon 的开发者所使用的方法。

9.4 Cilium Tetragon

Tetragon是 Cilium 项目的一部分（也是 CNCF 的一部分）。与附加到 LSM API 钩子不同，Tetragon 的方法是构建一个框架，用于将 eBPF 程序附加到 Linux 内核中的任意函数。

Tetragon 专为在 Kubernetes 环境中使用而设计，该项目定义了一种名为 TracingPolicy 的自定义 Kubernetes 资源类型。它用于定义一组事件，将 eBPF 程序附加到这些事件上，并定义需要由 eBPF 代码检查的条件以及条件满足时要执行的操作。以下是一个示例 TracingPolicy 的摘录：

```
spec:
kprobes:
- call: "fd_install"
...
    matchArgs:
    - index: 1
      operator: "Prefix"
      values:
      - "/etc/"
...
```

这个策略定义了一组要附加程序的 kprobe，其中第一个是内核函数 fd_install。这是内核中的一个内部函数。让我们探讨一下为什么会选择附加到这样一个函数上

9.4.1 附加到内核内部函数

系统调用接口和 LSM 接口被定义为 Linux 内核中的稳定接口，也就是说它们不会出现不兼容的情况。如果你今天编写使用这些接口函数的代码，在将来的内核版本中也能用。这些接口只占据了 Linux 内核这 3000 万行代码中的一小部分。尽管没有正式声明它们为稳定接口，但内核的很多代码实际上都是稳定的，长期以来就没有变化过，并且未来也不太可能发生变化。

编写 eBPF 程序将其附加到稳定的内核函数上，并期望其在相当长一段时间内保持正常工作是完全合理的。此外，考虑到新的内核版本通常需要几年时间才能得到广泛部署，我

们可以认为在处理可能出现的不兼容性问题时有足够的时间。

Tetragon 的贡献者包括一些内核开发人员，他们利用对内核的了解，确定了一些位置–这些位置适合附加 eBPF 程序。有几个示例的 TracingPolicy 定义利用了这些知识。这些示例监控安全事件，包括文件操作、网络活动、程序执行和权限更改等恶意行为可能涉及的各种操作。

让我们回到那个附加到 fd_install 策略定义的例子。”fd” 代表” 文件描述符”，源代码中的注释告诉我们，这个函数的作用是” 在 fd 数组中安装一个文件指针”。这发生在文件打开时，文件的数据结构在内核中填充后才会调用该函数。这是一个检查文件名称的安全地方，正如前面的 TracingPolicy 示例所示，只有文件名以”/etc/” 开头时才感兴趣。

与 LSM BPF 程序类似，Tetragon 的 eBPF 程序可以访问上下文信息，以在内核中进行安全决策。与其将给定类型的所有事件报告给用户空间，安全相关的事件可以在内核中进行过滤，只有违反策略的事件才报告给用户空间。

9.4.2 预防性安全

大多数基于 eBPF 的安全工具使用 eBPF 程序来检测恶意事件，并通知用户空间程序采取行动。如图 9-4 所示，用户空间程序采取的任何行动都是异步执行的，这时可能为时已晚了——可能已经泄露了数据，或者攻击者可能已经将恶意代码持久化到磁盘上。

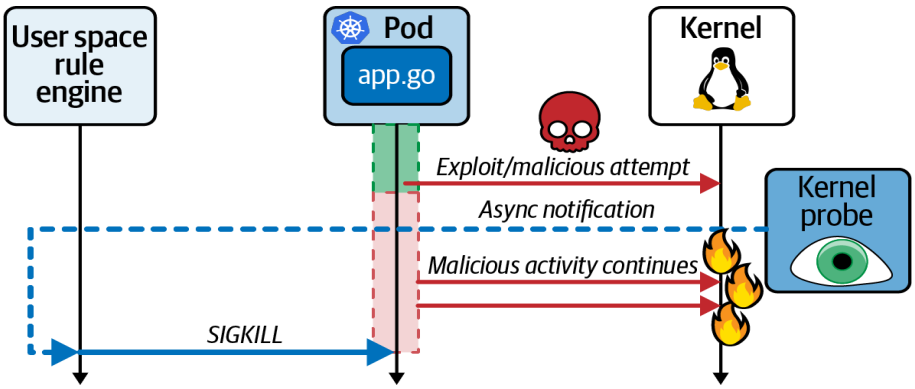

图 9.4: 从内核到用户空间的异步通知可能导致攻击继续进行一段时间

在内核版本 5.3 及更高版本中，存在一个名为 bpf_send_signal() 的 BPF 辅助函数。Tetragon 使用这个函数来实现预防性安全措施。如果策略定义了 Sigkill 操作，任何匹配的事件都会导致 Tetragon 的 eBPF 代码生成一个 SIGKILL 信号，终止正在尝试执行违反策略操作的进程。如图 9-5 所示，这是同步发生的，也就是说，内核正在执行的被 eBPF 代码判断为违反策略的活动将被阻止。

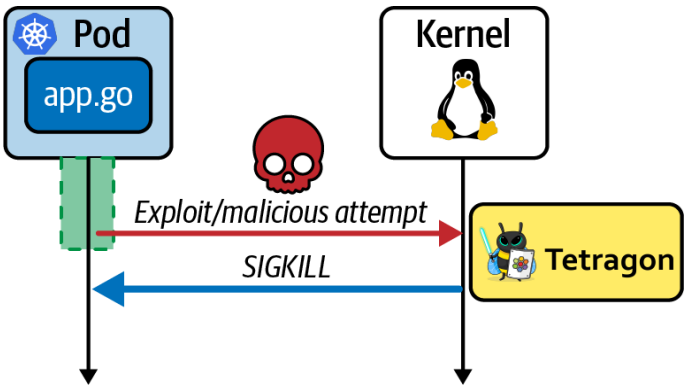

图 9.5: Tetragon 通过从内核发送 SIGKILL 信号同步地终止恶意进程

需要谨慎使用 Sigkill 策略，因为配置不正确的策略可能会终止程序，但这也是 eBPF 在安全领域非常强大的应用。你可以先以“审计”模式运行，生成安全事件但不应用 SIGKILL 强制执行，直到你确信该策略不会破坏任何东西。

如果你有兴趣了解如何使用 Cilium Tetragon 来检测安全事件，可以阅读 Natália Réka Ivánkó 和 Jed Salazar 撰写的《使用 eBPF 进行安全可观测性》的报告，其中有更详细的内容。

9.5 网络安全

本章讨论了如何使用 eBPF 来非常高效地实现网络安全机制。总结如下：

- 防火墙和 DDoS 保护非常适合通过 eBPF 程序来完成，只要将 eBPF 程序附加到网络数据包接收事件，并且将 XDP 程序卸载到硬件上，恶意数据包甚至可能都根本不会到达 CPU！
- 对于更复杂的网络策略，例如 Kubernetes 策略确定允许哪些服务彼此通信，eBPF 程序可以附加到网络堆栈中的特定点来实现，如果确定数据包违反了策略就可以丢弃该数据包。

网络安全工具通常在预防模式下使用，不仅仅会审计恶意活动还会丢弃数据包。这是因为恶意攻击者很容易发动网络攻击；如果你的设备对外暴露了公共 IP 地址，不久之后你就会开始看到可疑的流量，并因此被迫采取预防措施。

相反，许多组织将入侵检测工具设置为审计模式，并依赖取证来确定可疑事件是否真是恶意的，以及需要采取什么补救措施。如果某个安全工具过于粗糙，容易产生误报，那么它需要在审计模式下运行而不是预防模式下运行也就不足为奇了。我相信 eBPF 正在为更精密的安全工具提供细粒度、准确的控制。就像我们今天认为防火墙在预防模式下的准确性足够用一样，我们将看到越来越多的预防性工具将作用于其他非网络事件。甚至可能基于 eBPF 的控制代码会作为程序的一部分打包，以便提供自己的运行时安全能力。

9.6 总结

在本章中，你了解了 eBPF 安全领域的应用是如何从对系统调用的低级检查逐渐演变为更复杂的安全策略检查、内核事件过滤和运行时检查。

还有很多项目在使用 eBPF 进行安全工作，我相信未来几年，将会看到这一领域的工具不断发展，并得到广泛使用。

第十章　eBPF 编程

在本书的前面章节中，你已经学到了很多关于 eBPF 的知识，并看到了它在各种应用中的应用。但是如果你想基于 eBPF 实现自己的想法，该怎么办呢？本章将讨论如何编写自己的 eBPF 工具。

正如你在阅读本书时所了解的，eBPF 编程包括两个部分：

- 编写在内核中运行的 eBPF 程序。
- 编写管理和与 eBPF 程序交互的用户空间代码。

本章将介绍的大多数库和语言都要求作为程序员的你同时处理这两个部分，并意识到在哪里该处理什么。bpftrace 是一种简单的 eBPF 编程语言，它为程序员省去了这种负担，便于开发。

10.1　Bpftrace

如项目的 README 页面所述，“bpftrace 是一种用于 Linux eBPF 的高级追踪语言...受到 awk 和 C 的启发，以及先前的跟踪工具，如 DTrace 和 SystemTap 的启发。”

bpftrace命令行工具将用这种高级语言编写的程序转换为 eBPF 内核代码，并在终端中提供一些输出格式化的功能。作为用户，你不需要真正考虑内核和用户空间的分离。

在项目文档中，你会找到几个有用的命令示例，包括一个漂亮的教程，从编写一个简单的“Hello World”脚本到编写从内核数据结构中读取的数据的更复杂的脚本。

你可以从 Brendan Gregg 的bpftrace 速查表中了解 bpftrace 提供的各种功能。或者，如果你想深入了解 bpftrace 和 BCC，可以阅读他的书《BPF Performance Tools》。

正如其名称所示，bpftrace 可以附加到跟踪（也称为性能相关）事件，包括 kprobes、uprobes 和 tracepoints。例如，你可以使用-l 选项列出机器上可用的 tracepoints 和 kprobes，如下所示：

```
$ bpftrace -l "*execve*"
tracepoint:syscalls:sys_enter_execve
tracepoint:syscalls:sys_exit_execve
...
kprobe:do_execve_file
kprobe:do_execve
```

```
kprobe:__ia32_sys_execve
kprobe:__x64_sys_execve
...
```

这个例子找到了所有包含"execve" 的可附加点。从输出中可以看出，可以附加到一个名为 do_execve 的 kprobe 上。下面是一个附加到该事件的 bpftrace 脚本：

```
bpftrace -e 'kprobe:do_execve { @[comm] = count(); }'
Attaching 1 probe...
^C
@[node]: 6
@[sh]: 6
@[cpuUsage.sh]: 18
```

脚本的 @[comm] = count(); 部分是附加到该事件的代码。这个例子跟踪了不同可执行文件触发该事件的次数。

bpftrace 脚本可以协调附加到不同事件的多个 eBPF 程序。例如，一个名为opensnoop.bt的脚本可用于报告文件的打开情况。以下是一个摘录：

```
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat
{
    @filename[tid] = args->filename;
}

tracepoint:syscalls:sys_exit_open,
tracepoint:syscalls:sys_exit_openat
/@filename[tid]/

{
    $ret = args->ret;
    $fd = $ret > 0 ? $ret : -1;
    $errno = $ret > 0 ? 0 : - $ret;

    printf("%-6d %-16s %4d %3d %s\n", pid, comm, $fd, $errno,
        str(@filename[tid]));
    delete(@filename[tid]);
}
```

这个脚本定义了两个不同的 eBPF 程序，分别附加到两个不同的内核跟踪点，即 open() 和 openat() 系统调用的进入和退出点。这两个系统调用都用于打开文件，并接受文件名作

为输入参数。无论是哪种系统调用的进入点触发的程序，都会将文件名缓存在一个映射中，其中键是当前线程 ID。当触发退出跟踪点时，脚本中的/@filename[tid]/行会从该映射中检索出缓存的文件名。

运行这个脚本会生成如下内容：

```
./opensnoop.bt
Attaching 6 probes...
Tracing open syscalls... Hit Ctrl-C to end.
PID     COMM               FD ERR PATH
297388  node               30   0 /home/liz/.vscode-server/data/User/
                                  workspaceStorage/73ace3ed015
297360 node                23   0 /proc/307224/cmdline
297360 node                23   0 /proc/305897/cmdline
297360 node                23   0 /proc/307224/cmdline
```

我刚刚告诉你，有四个 eBPF 程序附加到跟踪点，那么为什么输出中说有六个探针？答案是有两个用于 BEGIN 和 END 子句的“特殊探针”，完整版本的程序包括用它们来初始化和清理脚本（类似于 awk 语言）。出于简洁起见，我在这里省略了这些子句，但你可以在源代码中找到它们。

如果你使用的是 bpftrace，你不需要了解底层的程序和映射情况，但对于那些已经阅读过本书前几章的读者来说，这些概念现在应该已经相当熟悉了。如果你有兴趣查看运行 bpftrace 程序时加载到内核中的程序和映射，你可以像第 3 章中展示的那样用 bpftool 来实现。下面是我在运行 opensnoop.bt 时得到的输出：

```
$ bpftool prog list
...
494: tracepoint name sys_enter_open tag 6f08c3c150c4ce6e gpl
        loaded_at 2022-11-18T12:44:05+0000 uid 0
        xlated 128B jited 93B memlock 4096B map_ids 254
495: tracepoint name sys_enter_opena tag 26c093d1d907ce74 gpl
        loaded_at 2022-11-18T12:44:05+0000 uid 0
        xlated 128B jited 93B memlock 4096B map_ids 254
496: tracepoint name sys_exit_open tag 0484b911472301f7 gpl
        loaded_at 2022-11-18T12:44:05+0000 uid 0
        xlated 936B jited 565B memlock 4096B map_ids 254,255
497: tracepoint name sys_exit_openat tag 0484b911472301f7 gpl
        loaded_at 2022-11-18T12:44:05+0000 uid 0
        xlated 936B jited 565B memlock 4096B map_ids 254,255

$ bpftool map list
```

```
254: hash flags 0x0
        key 8B value 8B max_entries 4096 memlock 331776B
255: perf_event_array name printf flags 0x0
        key 4B value 4B max_entries 2 memlock 4096B
```

你可以明显地看到这四个跟踪点程序，以及用于缓存文件名的哈希映射和用于将输出数据从内核传递到用户空间的 perf_event_array。

bpftrace 工具是构建在 BCC 之上的，你在本书的其他地方已经了解过 BCC，并且我稍后在本章中还会介绍它。bpftrace 脚本被转换为 BCC 程序，然后使用 LLVM/Clang 工具链在运行时进行编译。

如果你想要基于 eBPF 的性能测量的工具，很可能会发现bpftrace可以满足你的需求。尽管 bpftrace 可以作为一个使用 eBPF 进行跟踪的强大工具，但它并没有完全利用 eBPF 的全部潜力。

要发挥 eBPF 的全部潜力，你需要直接为内核编写 eBPF 程序，并处理用户空间部分。这两个方面通常可以使用完全不同的语言来开发。

10.2 eBPF 编程语言选择

eBPF 程序可以直接使用 eBPF 字节码编写，但实际上，大多数 eBPF 程序是从 C 或 Rust 编译成字节码的。这些语言的编译器支持将代码编程成 eBPF 字节码。

并非所有编译语言都适用于 eBPF 字节码。如果语言涉及运行时组件（例如 Go 或 Java 的虚拟机），它很可能与 eBPF 的验证器不兼容。例如，很难想象内存垃圾回收与验证器对内存安全使用的检查相协调。同样，eBPF 程序需要是单线程的，因此无法使用并发。

有一个有趣的项目XDPLua，虽不是 eBPF，但它提出使用 Lua 脚本编写在内核中直接运行的 XDP 程序。然而，该项目的初步研究表明，eBPF 可能更具优。特别是随着内核版本中 eBPF 功能的增强（现在已经支持循环），除了一些人喜欢用 Lua 脚本编写代码之外，很难说 Lua 有多大的优势。

我猜想，大多数使用 Rust 编写 eBPF 内核代码的人也会选择相同的语言编写用户空间代码，因为数据结构一致，就无需重写一套。但这并非强制要求，你可以将 eBPF 内核代码与任何用户空间语言混用。

那些选择使用 C 写 eBPF 内核代码的人也可以选择用 C 来写用户空间代码。但 C 是一种相当底层的语言，需要程序员自己处理许多细节，尤其是内存管理。虽然有些人习惯这样干，但很多人更愿意使用其他更高级的语言写用户空间代码。无论你偏好哪种语言，你都希望有一个提供 eBPF 支持的库，这样你就不必直接编写第三章中看到的那些系统调用接口。本章的其余部分，我们将讨论不同语言中流行的 eBPF 库。

10.3 BCC Python/Lua/C++

在第 2 章中，我给你展示的第一个“Hello World”示例是用 BCC 库写的 Python 程序。该项目包含了许多使用相同库实现的性能测量工具（以及我即将介绍的基于 libbpf 的新实现）。

除了文档中介绍如何使用提供的 BCC 工具进行性能测量外，BCC 还包括参考指南和Python 编程教程，帮助你在这个框架中开发自己的 eBPF 工具。

第 5 章讨论了 BCC 的可移植性方法，即在运行时将 eBPF 代码编译，以确保与目标机器的内核数据结构兼容。在 BCC 中，你将内核端的 eBPF 程序代码定义为字符串（或者是 BCC 读取为字符串的文件内容）。这个字符串被传递给 Clang 进行编译，但在此之前，BCC 对字符串进行了一些预处理。这使它能够为程序员提供快捷方式，其中一些你在本书中已经见过。例如，下面是第 2 章中示例代码 hello_map.py 中的一些相关行:

```
#!/usr/bin/python3                  // 1
from bcc import BPF

program = """                       // 2
BPF_RINGBUF_OUTPUT(output, 1); // 3
...
int hello(void *ctx) {
    ...
    output.ringbuf_output(&data, sizeof(data), 0); // 4
    return 0;
}
"""

b = BPF(text=program) // 5
...

b["output"].open_ring_buffer(print_event) // 6
...
```

1. 这是一个运行在用户空间的 Python 程序。

2. 程序字符串保存了要编译并加载到内核中的 eBPF 程序。

3. BPF_RINGBUF_OUTPUT 是一个 BCC 宏，定义了一个名为 output 的环形缓冲区。这是程序字符串的一部分，所以可以假设它是从内核的角度定义缓冲区的。

4. 这行看起来像是在一个名为 object 的对象上调用 ringbuf_output() 方法。但等一下，对象的方法不是 C 语言的一部分！BCC 在这里做了一些重要的工作，将这些方法展开为底层的 BPF 辅助函数 bpf_ringbuf_output()。

5. 这是程序字符串被重写为 Clang 可以编译的 BPF C 代码的地方。这行还将生成的程序加载到内核中。

6. 代码中没有其他地方定义名为 output 的环形缓冲区，然而在这里的 Python 用户空间代码中可以访问它。BCC 在处理第 3 步的预处理时发挥了双重作用，它为用户空间和内核空间都定义了环形缓冲区。

正如这个示例所示，BCC 本质上提供了自己的类似于 C 语言的语言来进行 BPF 编程。它简化了程序员的工作，处理了内核和用户空间的共享结构定义，并提供了便捷的方式来包装 BPF 辅助函数。这意味着如果你对 eBPF 编程还不熟悉，尤其是如果你已经熟悉 Python，BCC 是一个进入 eBPF 编程的方式。

如果你想探索 BCC 编程，这个针对 Python 程序员的教程是一个很好的材料，它涵盖了比本书中更多的 BCC 功能。

文档并没有很清楚地说明，除了支持 Python 作为 eBPF 工具的用户空间部分的语言外，BCC 还支持使用 Lua 和 C++ 来编写工具。在示例中有 lua 和 cpp 目录，你可以以此为基础编写自己的代码。

BCC 对程序员来说可能很方便，但由于将编译器工具链与实用程序一起分发的效率低下，如果你希望编写并分发的生产质量级别的工具，我建议你考虑下本章后面讨论的其他库。

10.4 C 和 Libbpf

在本书中，你已经看到了许多使用 C 编写的 eBPF 程序并使用 LLVM 工具链将其编译为 eBPF 字节码。你还看到了添加了 BTF 和 CO-RE 支持的扩展。许多 C 程序员还熟悉另一个主要的 C 编译器 GCC，并且乐意知道，从版本 10开始，GCC 也支持将 eBPF 作为目标进行编译。然而，与 LLVM 提供的功能相比，GCC 仍然存在差距。

正如第 5 章中所见，CO-RE 和 libbpf 支持了一种可移植的 eBPF 编程方法，不需要为每个 eBPF 工具提供编译器工具链。BCC 项目利用了这一点，除了最初的一组基于 BCC 的性能跟踪工具之外，现在还有使用 libbpf 重写的版本。普遍的共识是，基于 libbpf 重写的 BCC 工具版本是更好的选择，因为它们的内存占用较低，并且不会在编译时出现启动延迟。

如果你习惯使用 C 进行编程，使用 libbpf 是很有意义的。你已经在本书中看到了很多这方面的示例。

要在 C 中编写自己的 libbpf 程序，最好的起点是libbpf-bootstrap。阅读 Andrii Nakryiko 的关于此项目的博客文章，了解背后的动机是一个很好的入门介绍。

还有一个名为libxdp的库，它在 libbpf 的基础上构建，以便更容易开发和管理 XDP 程序。这是 xdp-tools 的一部分，它还提供了我最喜欢的用于 eBPF 编程的学习资源之一：XDP 教程(视频)。

但是 C 是一种相当具有挑战性的底层语言。C 程序员必须对内存管理和缓冲处理等问题负责，很容易写出存在安全漏洞的代码，更不用说由于处理指针不当而导致的崩溃了。

eBPF 验证器在内核方面提供了保护，但对于用户空间代码来说，没有类似的保护。

好消息是，其他编程语言也有与 libbpf 进行接口或提供类似重定位功能的库，以实现可移植的 eBPF 程序。以下是一些最受欢迎的选项。

10.4.1 Go

Go 语言已经广泛应用于基础设施和云原生工具，因此自然而然地应该有基于 Go 编写 eBPF 代码库。

10.4.2 Gobpf

可能第一个正式的 Golang 实现是gobpf项目，它作为 Iovisor 的一部分与 BCC 并列。然而，该项目已经有一段时间没有维护了，而且在我撰写本书时，有一些关于将其弃用的讨论，因此在选择库时请注意这一点。

10.4.3 Ebpf-go

作为 Cilium 项目的一部分，广泛使用的eBPF Go库（在 GitHub 上大约 10,000 个引用，获得了近 4,000 个 star）。它提供了方便的函数来管理和加载 eBPF 程序和映射，包括 CO-RE 支持，纯用 Go 语言实现。

使用该库，你可以选择将你的 eBPF 程序编译为字节码，并使用名为bpf2go的工具将字节码嵌入到 Go 源码中。在构建过程中，你需要 LLVM/Clang 编译器来生成字节码。一旦 Go 代码编译完成，你就会得到一个单独的 Go 二进制文件，其中包含了 eBPF 字节码。该二进制文件对不同的内核是可移植的，除了 Linux 内核本身之外没有任何依赖项。

cilium/ebpf 库还支持加载和管理以独立 ELF 文件形式构建的 eBPF 程序（就像本书中看到的 *.bpf.o 示例一样）。

撰写本文时，cilium/ebpf 库支持性能事件跟踪，包括相对较新的 fentry 事件，以及广泛的网络程序类型，如 XDP 和 cgroup 套接字附加。

在 cilium/ebpf 的示例目录下，你会看到内核程序的 C 代码与相应的 Go 用户空间代码位于同一目录中：

- C 文件以// +build ignore 开头，告诉 Go 编译器忽略它们。撰写本书时，正在进行更新，以改为使用更新的//go:build 构建标签风格。
- 用户空间文件包括类似以下的行，告诉 Go 编译器在 C 文件上调用 bpf2go 工具：

```
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc
   $BPF_CLANG
    -cflags $BPF_CFLAGS bpf <C filename> -- -I../headers
```

运行 go:generate 命令会重新构建 eBPF 程序并生成项目骨架代码。

类似于第 5 章中介绍的 bpftool gen skeleton，bpf2go 生成用于操作 eBPF 对象的骨架代码，最大限度减少需要用户编写的用户空间代码（只是生成的是 Go 代码而不是 C 代码）。输出文件还包括包含字节码的.o 对象文件。

事实上，bpf2go 为大端和小端架构生成了两个版本的字节码.o 文件。相应地生成了两个.go 文件，并且在编译时会根据目标平台使用正确的版本。以 cilium/ebpf 中的kprobe 示例为例，自动生成的文件包括：

- 包含 eBPF 字节码的 bpf_bpfeb.o 和 bpf_bpfel.o ELF 文件。
- bpf_bpfeb.go 和 bpf_bpfel.go 文件，定义与该字节码中映射、程序和链接相对应的 Go 结构和函数。

你可以将自动生成的 Go 代码中定义的对象与生成它的 C 代码关联起来。以下是该 kprobe 示例中 C 代码中定义的对象：

```
struct bpf_map_def SEC("maps") kprobe_map = {
...
};

SEC("kprobe/sys_execve")
int kprobe_execve() {
...
}
```

自动生成的 Go 代码包括表示所有映射和程序的结构体：

```
type bpfMaps struct {
    KprobeMap *ebpf.Map ebpf:"kprobe_map"
}
type bpfPrograms struct {
    KprobeExecve *ebpf.Program ebpf:"kprobe_execve"
}
```

名称”KprobeMap” 和”KprobeExecve” 是从 C 代码中使用的映射和程序名称派生而来的。这些对象被分组到一个名为 bpfObjects 的结构体中，表示要加载到内核中的内容：

```
type bpfObjects struct {
    bpfPrograms
    bpfMaps
}
```

然后，你可以在用户空间的 Go 代码中使用这些对象定义和自动生成的函数。为了给你一个启示，以下是基于相同kprobe 示例的 main 函数的摘录（出于简洁起见省略了错误处理）：

```
objs := bpfObjects{}
loadBpfObjects(&objs, nil) // 1
defer objs.Close()

kp, _ := link.Kprobe("sys_execve", objs.KprobeExecve, nil)// 2
defer kp.Close()

ticker := time.NewTicker(1 * time.Second) // 3
defer ticker.Stop()

for range ticker.C {
    var value uint64
    objs.KprobeMap.Lookup(mapKey, &value) // 4
    log.Printf("%s called %d times\n", fn, value)
}
```

1. 从字节码形式的对象文件中加载所有 BPF 对象到自动生成的代码定义的 bpfObjects 中。

2. 将程序附加到 sys_execve kprobe。

3. 设置一个定时器，以便代码每秒轮询一次映射。从映射中读取一个条目。

cilium/ebpf 目录中还有其他几个示例供你参考和借鉴。

10.4.4 Libbpfgo

由 Aqua Security 开发的libbpfgo项目在 libbpf 的 C 代码基础上实现了 Go 的包装器，提供了加载和附加程序的工具，以及使用 Go 本地特性（如通道）接收事件。因为它是构建在 libbpf 之上，所以支持 CO-RE。

以下是来自 libbpfgo 的 README 示例的摘录：

```
bpfModule := bpf.NewModuleFromFile(bpfObjectPath) // 1
bpfModule.BPFLoadObject()                         // 2
mymap, _ := bpfModule.GetMap("mymap")             // 3

mymap.Update(key, value)
rb, _ := bpfModule.InitRingBuffer("events", eventsChannel,
   buffSize)
rb.Start()
e := <-eventsChannel                              // 4
```

1. 从一个对象文件中读取 eBPF 字节码。
2. 将该字节码加载到内核中。
3. 操作 eBPF 映射中的一个条目。
4. Go 程序员喜欢通过通道接收来自环形缓冲区或 perf 缓冲区的数据，这是一种处理异步事件的语言特性。

这个库是为 Aqua 的Tracee安全项目创建的，也被其他项目使用，比如 Polar Signals 的Parca，它提供了基于 eBPF 的 CPU 分析。对于这个项目的方法，唯一的担忧是 libbpfC 代码和 Go 之间的 CGo 边界，它可能导致性能和其他问题。

虽然 Go 语言成为构建基础设施工具的常用语言已有约十年的时间，但最近出现了越来越多开发者喜欢用的 Rust。

10.5 Rust

Rust 语言在构建基础设施工具方面的使用越来越广泛。它允许像 C 语言一样进行低级别访问，但又具有内存安全的优势。事实上，Linus Torvalds 在 2022 年确认，Linux 内核本身将开始整合 Rust 代码，并且最近的 6.1 版本已经具备了一些初步的Rust 支持。

正如我在本章前面讨论的那样，Rust 可以编译成 eBPF 字节码，这意味着（在有适当的库支持的情况下），可以使用 Rust 编写 eBPF 工具的用户空间和内核代码。

Rust eBPF 开发中有几个选择：libbpf-rs、Redbpf 和 Aya。

10.5.1 Libbpf-rs

Libbpf-rs是 libbpf 项目的一部分，它提供了对 libbpf C 代码的 Rust 封装，使你可以使用 Rust 编写 eBPF 代码的用户空间部分。从该项目的示例中可以看出，eBPF 程序本身还是用 C 编写的。

在libbpf-bootstrap项目中还有一些 Rust 的示例，旨在帮助你开始构建自己的代码。

这个 crate 对于将 eBPF 程序整合到基于 Rust 的项目中非常有帮助，但它不能满足许多人希望在内核层面也使用 Rust 编写代码的愿望。

10.5.2 Redbpf

Redbpf是一组 Rust crate 与 libbpf 接口，作为基于 eBPF 的安全监控代理foniod的一部分开发而来。Redbpf 出现的时候，Rust 还不能直接编译为 eBPF 字节码，因此它使用了多步编译过程，包括从 Rust 编译为 LLVM 位码，然后使用 LLVM 工具链生成 ELF 格式的 eBPF 字节码。Redbpf 支持多种 eBPF 程序类型，包括跟踪点、kprobe 和 uprobe、XDP、TC 和一些套接字事件。

随着 Rust 编译器 rustc 具备直接生成 eBPF 字节码的能力，出现了一个名为 Aya 的项目。在撰写本文时，根据ebpf.io 社区网站的说法，Aya 被认为是一个” 新兴” 项目，而 Redbpf 被列为一个重要项目，但我个人的观点是，Aya 发展得更好。

10.5.3 Aya

Aya是直接使用 Rust 构建的，达到了系统调用级别，因此它不依赖于 libbpf（或者 BCC 或 LLVM 工具链）。但它支持 BTF 格式，与 libbpf 一样支持重定位，因此它提供了相同的 CO-RE 功能，可以编译一次并在其他内核上运行。在撰写本文时，它支持比 Redbpf 更广泛的 eBPF 程序类型，包括跟踪/性能相关事件、XDP 和 TC、cgroups 和 LSM 附加。

正如我之前提到的，Rust 编译器还支持将代码编译为 eBPF 字节码，因此可以在内核和用户空间中都使用这种语言进行 eBPF 编程。

在 Rust 中能够原生地编写内核和用户空间代码，而无需依赖 LLVM 的能力吸引了 Rust 程序员。GitHub 上有一个有趣的讨论，讨论了lockc项目开发人员（一个使用 LSM 钩子增强容器工作负载安全性的 eBPF 项目）为什么决定将其项目从 libbpf-rs 迁移到 Aya。

该项目包括aya-tool，一个实用程序，用于生成与内核数据结构匹配的 Rust 结构定义。

Aya 项目非常注重开发者体验，使新手可以轻松入门。为此，”Aya book” 是一本非常易读的介绍书，其中包含一些很好的示例，并附带有解释。

为了让你对 Rust 中的 eBPF 代码有一个简要的了解，以下是 Aya XDP 示例的摘录，该示例允许所有流量通过：

```
#[xdp(name="myapp")]                    // 1
pub fn myapp(ctx: XdpContext) -> u32 {
    match unsafe { try_myapp(ctx) } {  // 2
        Ok(ret) => ret,
        Err(_) => xdp_action::XDP_ABORTED,
    }
}


unsafe fn try_myapp(ctx: XdpContext) -> Result<u32, u32> { //3
    info!(&ctx, "received a packet");
    Ok(xdp_action::XDP_PASS)
}
```

1. 这一行定义了节名，相当于 C 语言中的 SEC(”xdp/myapp”)。

2. 名为 myapp 的 eBPF 程序调用函数 try_myapp 来处理在 XDP 层接收到的网络数据包。

3. 函数 try_myapp 记录接收到数据包的情况，并始终返回 XDP_PASS 值，告诉内核继续正常处理该数据包。

正如我们在本书中看到的基于 C 语言的示例一样，eBPF 程序被编译成一个 ELF 目标文件。不同之处在于，Aya 使用 Rust 编译器而不是 Clang 来生成这个文件。

Aya 还生成了用于将 eBPF 程序加载到内核并将其附加到事件的用户空间代码。下面是同一个示例的用户空间代码的关键部分：

```
let mut bpf = Bpf::load(include_bytes_aligned!(      // 1
    "../../target/bpfel-unknown-none/release/myapp"
))?;

let program: &mut Xdp = bpf.program_mut("myapp").unwrap().
   try_into()?; // 2

program.load()?;  // 3
program.attach(&opt.iface, XdpFlags::default()) // 4
```

1. 从编译器生成的 ELF 目标文件中读取 eBPF 字节码。
2. 在字节码中找到名为 myapp 的程序。
3. 将其加载到内核。
4. 将其附加到指定网络接口的 XDP 事件上。

如果你是 Rust 程序员，我强烈推荐你详细了解”Aya book” 中的其他示例。还有一篇 Kong 的博文，介绍了如何使用 Aya 编写一个 XDP 负载均衡器。

Aya 的维护者 Dave Tucker 和 Alessandro Decina 在 “eBPF and Cilium Office Hours” 直播的第25 集中与我一起，演示并介绍了使用 Aya 进行 eBPF 编程的内容。

10.5.4 Rust-bcc

Rust-bcc提供了 Rust 绑定，模仿了 BCC 项目的 Python 绑定，并提供了一些 BCC 跟踪工具的 Rust 实现。

10.6 测试 BPF 程序

有一个 bpf() 命令，BPF_PROG_RUN，可以从用户空间运行 eBPF 程序进行测试。BPF_PROG_RUN（目前）仅适用于一些主要与网络相关的 BPF 程序类型。

你还可以使用一些内置的统计信息获取有关 eBPF 程序性能的信息。运行以下命令以启用它：

```
$ sysctl -w kernel.bpf_stats_enabled=1
```

这将在 bpftool 的输出中显示有关程序的额外信息，例如：

```
$ bpftool prog list
...
2179: raw_tracepoint name raw_tp_exec tag 7f6d182e48b7ed38 gpl
    run_time_ns 316876 run_cnt 4
    loaded_at 2023-01-09T11:07:31+0000 uid 0
```

```
        xlated 216B jited 264B memlock 4096B map_ids 780,777
        btf_id 953
        pids hello(19173)
```

额外的统计信息以粗体显示，在这里显示该程序已运行四次，总共花费了约 300 微秒的时间。

可以从 Quentin Monnet 在 2020 年 FOSDEM 上的演讲 “Tools and mechanisms to debug BPF programs.” 中了解更多信息。

10.7 多个 eBPF 程序

eBPF 程序是附加到内核事件的函数。许多应用程序需要跟踪多个事件来实现其目标。一个简单的例子就是 opensnoop [2]。前面的章节中，我介绍了 bpftrace 版本的 opensnoop，你看到它将 BPF 程序附加到了四个不同的系统调用跟踪点上：

- syscall__enter__open
- syscall__exit__open
- syscall__enter__openat
- syscall__exit__openat

这些是内核处理 open() 和 openat() 系统调用的入口和出口点。这两个系统调用用于打开文件，而 opensnoop 工具跟踪这两个调用。

但是为什么它需要同时跟踪这些系统调用的入口和出口呢？入口点被用于获取系统调用的参数，包括文件名和传递给 open[at] 系统调用的任何标志。但在这个阶段，还不能确定文件是否成功打开。这就解释了为什么需要将 eBPF 程序附加到出口点的原因。

如果你查看 opensnoop 的 libbpf-tools版本，你会发现只有一个用户空间程序，它会将所有四个 eBPF 程序加载到内核中，并将它们附加到相应的事件上。这些 eBPF 程序本质上是独立的，但它们使用 eBPF 映射在之间进行协调。

一个复杂的应用程序甚至可能需要在长时间内动态添加和删除 eBPF 程序。对于任何给定的应用程序，可能没有固定数量的 eBPF 程序。例如，Cilium 会将 eBPF 程序附加到每个虚拟网络接口上，在 Kubernetes 环境中，这些接口的创建和销毁取决于运行的 Pod 数量。

本章介绍的大多数库都会自动处理这种多个 eBPF 程序的情况。例如，libbpf 和 ebpf-go 会生成骨架代码，可以通过一个函数调用从对象文件或缓冲区中的字节码加载所有程序和映射。它们还生成了更细粒度的函数，以便可以单独操作程序和映射。

10.8 总结

大多数使用基于 eBPF 的工具的人不需要自己编写 eBPF 代码，但如果你确实想要自己实现一些功能，你有很多选择。eBPF 是一个变化迅速的领域，所以很可能在你阅读本

章时，会有新的语言库和框架出现。你可以在 ebpf.io 的重要项目列表中的基础设施页面上找到关于 eBPF 的主要语言项目的最新列表。

对于快速收集跟踪系统信息，bpftrace 是一个非常有价值的选择。

如果你需要更灵活和可控的选项，并且熟悉 Python，那么 BCC 是一种快速构建 eBPF 工具的方式，前提是你不关心运行时的编译步骤。

如果你要编写在不同内核版本间广泛分发和移植的 eBPF 代码，你可能会想利用 CO-RE。目前支持 CO-RE 的用户空间框架包括 C 语言的 libbpf，Go 语言的 cilium/ebpf 和 libbpfgo，以及 Rust 语言的 Aya。

如果需要进一步的建议，我强烈推荐加入eBPF Slack，并在那里讨论你的问题。在这个社区中，你很可能会遇到许多这些语言库的维护者。

10.9 练习

如果你想尝试本章中讨论的一个或多个库，”Hello World” 总是一个很好的起点：

1. 使用你选择的一个或多个库，编写一个简单的”Hello World” 程序，输出一个简单的跟踪消息。

2. 使用 llvm-objdump 来比较与第 3 章中的”Hello World” 示例生成的字节码。你会发现很多相似之处！

3. 正如你在第 4 章中看到的，你可以使用 strace -e bpf 来查看何时进行 bpf() 系统调用。尝试对你的”Hello World” 程序使用这个命令，看看它的行为是否符合你的预期。

第十一章　eBPF 发展趋势

eBPF 技术尚未完全成熟！和大多数软件一样，它在 Linux 内核中不断演变发展，并且也逐步添加到了 Windows 操作系统中。在本章中，我们将探讨 ePBF 未来的发展方向。

自从在 Linux 内核中引入 BPF 以来，它已经发展成为拥有自己的子系统、邮件列表和维护人员的独立模块。随着 eBPF 的普及和 Linux 内核社区外对其产生的浓厚兴趣，创立一个中立的机构来协调各方的合作自是应有之义，这个机构就是 eBPF 基金会。

11.1　eBPF 基金会

eBPF 基金会于 2021 年由 Google、Isovalent、Meta（原 Facebook）、Microsoft 和 Netflix 在 Linux 基金会的指导下成立。该基金会作为一个中立机构，可以持有资金和知识产权，以便各商业公司间进行合作。

这并不意味着要改变 Linux 内核社区和 Linux BPF 子系统的贡献者们开发 eBPF 技术的方式。基金会的活动由 BPF Steering Committee（BPF 指导委员会）负责，该委员会完全由技术专家组成，包括 Linux 内核 BPF 模块维护人员和其他核心 eBPF 项目的代表。

eBPF 基金会专注于让 eBPF 作为一种技术平台以及支持 eBPF 开发的生态系统工具。构建在 eBPF 之上且寻求所有权中立的项目可能更适合归属到其他基金会。例如，Cilium、Pixie 和 Falco 都属于 CNCF（云原生计算基金会），这是有道理的，因为它们都用于云原生环境。

Microsoft 对在 Windows 操作系统中开发 eBPF 很感兴趣。这引发了一个需求，即定义 eBPF 的标准，以便在一个操作系统上编写的程序可以在另一个操作系统上使用。这项工作是在 eBPF 基金会的指导下进行的。

11.2　Windows eBPF

微软一直在为Windows 支持 eBPF做工作。我写这篇文章的时候，已经有功能性的demo展示了在 Windows 上运行的 Cilium 第 4 层负载均衡和基于 eBPF 的连接跟踪。

我之前说过，eBPF 编程是内核编程，乍一看，编写在 Linux 内核中运行并访问 Linux 内核数据结构的程序如何能在不同的操作系统中运行呢？但实际上，特别是在网络方面，所有操作系统间都有很多共同之处。无论是在 Windows 还是 Linux 机器上创建的网络数据

包都具有相同的结构，网络栈的层次结构也必须以相同的方式处理。

你还记得 eBPF 程序是由一组字节码指令组成吗？这些指令由内核中实现的虚拟机（VM）处理。这个虚拟机也可以在 Windows 中实现！

图 11-1 展示了该项目 GitHub仓库的 Windows eBPF 架构概述。从这个图可以看出，eBPF for Windows 重用了现有 eBPF 生态系统中的一些开源组件，例如 libbpf 和 Clang 对于生成 eBPF 字节码的支持。Linux 内核是根据 GPL 许可的，而 Windows 是私有的，因此 Windows 项目无法重用 Linux 内核实现的验证器。相反，它使用PREVAIL 验证器和uBPF JIT 编译器（两者都具有宽松的许可，以便更广泛地用于各种项目和组织）。

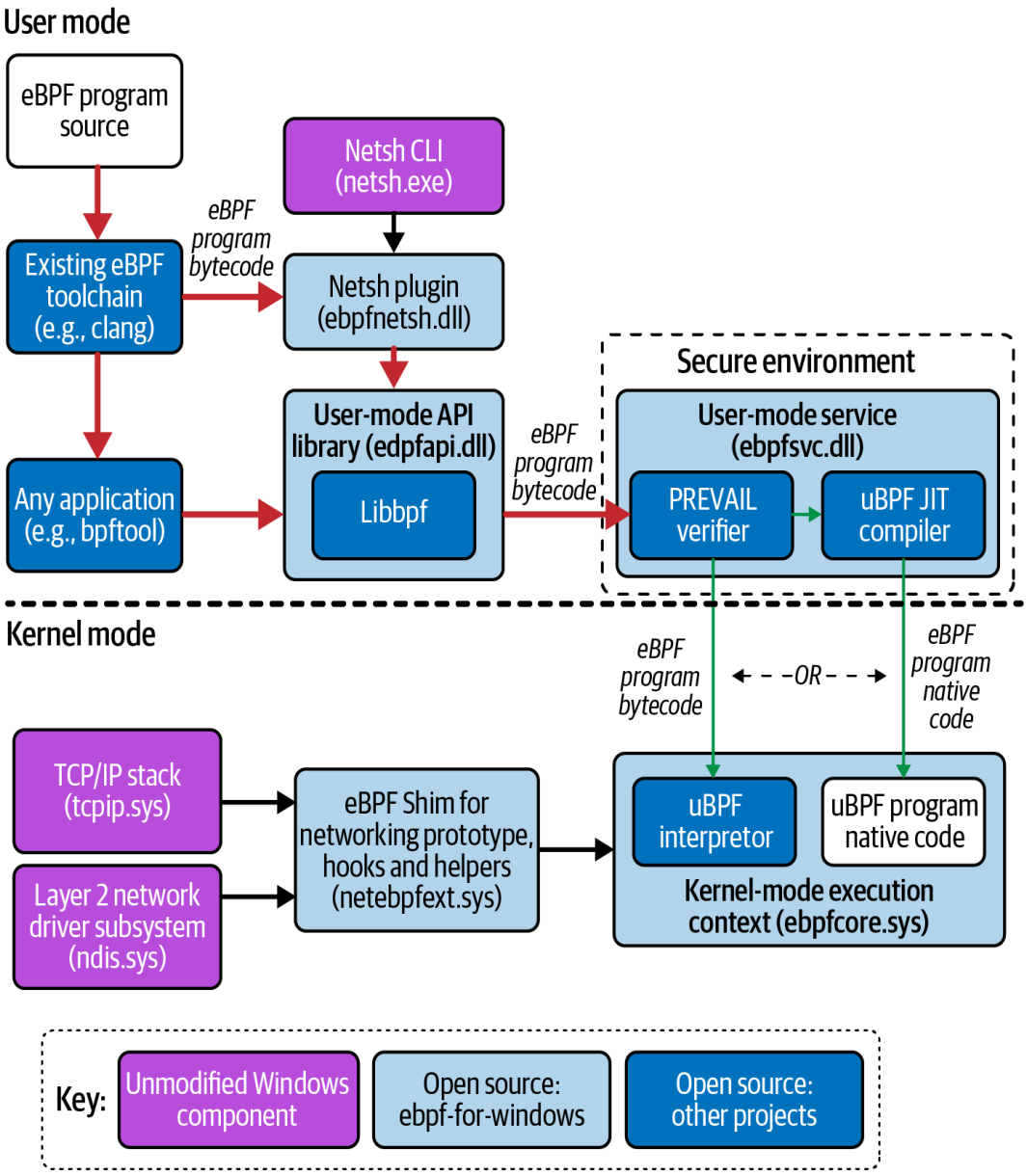

图 11.1: eBPF for Windows 架构概述

有趣的是，在 Windows 中，eBPF 代码是在用户空间的安全环境中进行验证和 JIT 编译的，而不是在内核中（在图 11-1 中内核显示的 uBPF 解释器仅用于调试构建而非生产环境）。

期望 Linux 下运行的 eBPF 程序都能在 Windows 上运行是现实的。这其实和让 eBPF 程序在不同的 Linux 内核版本上运行遭遇的挑战没什么不同：即使有了 CO-RE 支持，在不同内核版本之间，内部内核数据结构也可能发生变化、添加或删除。eBPF 程序员的任务是优雅地处理解决这些问题。

那么，在未来几年，我们该期待在 eBPF 中看到哪些变化呢?

11.3 Linux 下 eBPF 的演进

自从 3.15 版以来，几乎随着每个内核版本发布，eBPF 的功能都得到了改进。如果你想知道某个特定版本中有哪些功能可用，可以参考这个 BCC 项目维护的列表。我肯定在接下来的几年来会有更多内容添加进去。

预测将来的变化最好的方式就是听那些从事相关工作的人的意见。例如，在 2022 年的 Linux Plumbers Conference 上，eBPF 维护者 Alexei Starovoitov 发表了一个演讲，讨论了他对 eBPF 程序中使用的 C 语言的发展预期。我们已经看到 eBPF 从仅支持几千条指令演变到几乎无限复杂性，新增了对循环的支持和越来越多的 BPF 辅助函数。随着对所支持的 C 语言功能的进一步扩展和验证器的支持，eBPF C 语言可能会演变成适用于所有内核模块开发，同时具备 eBPF 的安全性和动态加载特性。

还有一些正在讨论和开发的关于新的 eBPF 功能的点子，包括：

- 签名的 eBPF 程序

过去几年，软件供应链安全一直是一个热门话题，其中一个关键因素是要能够检查程序是否来自合法的来源并且没有被篡改。实现这一目标的一种方式是验证程序的加密签名。你可能认为这是该内核干的事情，甚至只是作为验证步骤的一部分，但不幸的是，并没有那么简单！正如你在本书中看到的，用户空间加载程序会根据 Maps 的位置信息动态调整程序，并且出于 CO-RE 目的，在签名的角度看来，很难区分是否有恶意修改。这是 eBPF 社区渴望找到解决方案的一个问题。

- 长期有效的内核指针

eBPF 程序可以使用辅助函数或 kfunc 检索指向内核对象的指针，但指针仅在程序执行期间有效。无法将指针存储在 Maps 中以供后续检索。具有类型指针支持的想法将在这个问题上提供更多的灵活性。

- 内存分配

eBPF 程序不能简单地调用像 kmalloc() 这样的内存分配函数，但有一个提议建议提供一个针对 eBPF 的特定替代方案。

作为终端用户，你能够使用的 eBPF 功能取决于你使用的内核版本。正如我在第 1 章中提到的，内核版本需要经过数年时间才能在稳定的 Linux 发行版中得到支持。作为个人用户，你可以选择使用最新的内核版本，但绝大多数部署服务器的组织更倾向于使用稳定

且受支持的内核版本。eBPF 程序员必须考虑到，如果他们编写的代码依赖于内核的最新功能，这些功能在大多数生产环境中可能要数年时间才能使用得上。然而，一些组织可能有急切需求，会更快地升级内核版本以提前用上新的 eBPF 功能。

举个例子，Daniel Borkmann 在一个展望未来网络的演讲中讨论了一个名为 Big TCP 的功能。Big TCP 在 Linux 的 5.19 版本中添加，通过将网络数据包进行批处理以在内核中进行处理，实现了 100 GBit/s 及更高速度的网络传输。大多数 Linux 发行版在数年内可能不会采用这个最新的内核版本，但对于处理大量网络流量的专业组织来说，可能就要提前升级内核版本了。将 Big TCP 支持添加到 eBPF 和 Cilium 中意味着这些功能可以在大范围内得到使用，即使需要一段时间。

由于 eBPF 允许动态调整内核代码，自然地，可以预计它将被用于解决” 实际场景中” 的问题。在第 9 章中，您知道了使用 eBPF 来减轻内核漏洞的情况；同时，也正在进行让 eBPF 支持硬件设备（例如鼠标、键盘和游戏控制器）的工作，这是在第 7 章中提到的红外控制器协议解码的基础上进行的拓展。

11.4 eBPF 是一个平台，而不仅仅是一种功能

将近十年前，容器技术成为了炙手可热的新技术，似乎每个人都在谈论容器以及它带来的优势。如今，eBPF 也处于类似的阶段，许多会议演讲和博客文章（其中几篇在本书中也有提及）夸奖了 eBPF 的好处。如今，对许多开发人员来说，容器已经成为日常工作的一部分，无论是在本地使用 Docker 或其他容器运行时来运行代码，还是在 K8s 中部署代码。那么，eBPF 将来会成为每个人日常工作的工具吗?

我的答案是否定的，或者至少不是直接的。大多数用户不会去直接编写 eBPF 程序，也不会用 bpftool 这样的工具手动操作。但他们将经常使用 eBPF 构建的工具，无论是性能观测、调试、网络、安全、跟踪还是其他许多尚未使用 eBPF 实现的功能。用户可能不知道他们在用 eBPF，就像他们用容器时也不知道自己在使用命名空间 (namespaces) 和控制组 (cgroups) 等内核功能一样。

如今，具有 eBPF 的项目和供应商强调他们在使用 ePBF，因为 eBPF 是如此的非常强大，有许多优势。随着基于 eBPF 的项目和产品获得越来越多的认可和市场份额，eBPF，事实上会默认成为各种基础设施的技术平台。

掌握 eBPF 编程是一项要靠个人驱动且相对罕见的能力，就像如今内核开发比起开发业务应用或游戏等更不常见一样。如果你喜欢深入系统的底层并构建基础设施，那么掌握 eBPF 将对你大有裨益。希望本书在你的 eBPF 编程之旅中能提供一些帮助!

- 进一步阅读

在本书中，我提供了一些特定的文章和文档页面的参考资料。以下是一些其他的资源，可以帮助你在 eBPF 编程之旅中更进一步：

- eBPF 社区网站 ebpf.io。
- Cilium 文档中的 BPF 和 XDP 参考
- Linux内核文档中关于 BPF 的内容

- Brendan Gregg 关于使用 eBPF 进行性能和可观察性的网站
- Andrii Nakryiko 的网站，特别是关于 CO-RE 和 libbpf 的更多信息
- Lwn.net，一个关于 Linux 内核更新的好资源，包括 BPF 子系统的内容
- Elixir.bootlin.com，你可以浏览 Linux 源代码的网站
- eCHO，一个周播节目，涵盖了 eBPF 和 Cilium 社区的各种话题（本书的作者经常是主讲人）

11.5 结论

恭喜你读完了本书！

我希望《eBPF 指南》一书让你深入了解了 eBPF 的强大之处。也许它激发了你自己编写 eBPF 代码或尝试一些我讨论的工具的兴趣。如果你决定进行 eBPF 编程，我希望本书能够让你对如何入门有所信心。如果你在阅读过程中完成了练习，那就更棒了！

如果你对 eBPF 很感兴趣，也可以参与到社区活动里来。最好的起点是 ebpf.io 网站，它将为你提供最新的新闻、项目、活动以及有关 eBPF 的 Slack 频道，你可以在那里找到具备专业知识的人来回答你的问题。

欢迎你的反馈、评论以及对本书的任何纠错意见。你可以通过本书附带的 GitHub 仓库（github.com/lizrice/learning-ebpf）提供意见。我也很乐意直接听取你的意见。在网上的诸多地方，你都能找到我的身影，我的用户名是 @lizrice。

感谢您的阅读和支持！祝您在 eBPF 的旅程中取得更多的成果！

附录

作者信息

Isovalent 是 Cilium 云原生网络、安全和可观察性项目的创建者，Liz Rice 是 Isovalent eBPF 专家团队的首席开源官。她还是云原生计算基金会（CNCF, Cloud Native Computing Foundation）治理委员会和 OpenUK 委员会的成员。她曾担任 2019-2022 年 CNCF 技术监督委员会主席，2018 KubeCon + CloudNativeCon 联合主席。她也是 O'Reilly 出版的《容器安全》一书的作者。

Liz 在网络协议、分布式系统及视频点播、音乐和 VoIP 等数字技术领域拥有丰富的软件开发、团队和产品管理经验。平时，Liz 喜欢在她家乡伦敦之外，天气更好的地方骑自行车，参加 Zwift 虚拟比赛，并以 Insider Nine 的艺名制作音乐。

跋

《eBPF 指南》封面上的动物是早熊蜂（Bombus pratorum），一种遍布欧洲大部分地区（尤其是英国）和亚洲部分地区的熊蜂物种。

早熊蜂在田野、公园和森林等地建造巢穴，甚至利用废弃的鸟巢或啮齿动物巢穴。早熊蜂出现在一年的早期，通常从 3 月到 7 月，但在英国南部，工蜂可能早于 2 月出现，因此一年内可能有两次蜂群。

这种熊蜂品种体型相对较小。虽然蜂王、工蜂和雄性雄蜂有些许变化，但早熊蜂通常呈黑色，具有黄色领圈、腹部带黄色条纹，尾部呈红或暗橙色。

早熊蜂会由蜂王和工蜂组成蜂群，但与其他物种不同的是，蜂王通过攻击性行为而不是信息素来维持统治地位，她会用下颚猛撞最强壮的工蜂，以保持对蜂群的控制。工蜂负责采集白三叶草、蓟、鼠尾草、薰衣草等开花植物的花蜜和花粉；雄蜂则在蜂巢周期的后期生成，离开巢穴寻找新的蜂王。

O'Reilly 封面上的许多动物都处于濒危状态，但它们对世界至关重要。本书封面插图由 Karen Montgomery 绘制，基于《动物王国插图集》中的一幅古老线刻画。封面字体为 Gilroy Semibold 和 Guardian Sans。正文字体为 Adobe Minion Pro；标题字体为 Adobe Myriad Condensed；代码字体为 Dalton Maag 的 Ubuntu Mono。

参考文献

[1] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. *Winter USENIX conference*, page 25–39, January 1993.

[2] Liz Rice. What is ebpf. Website, 2022.

[3] Yujuan Jiang, Bram Adams, and Daniel German. Will my patch make it? and how fast? case study on the linux kernel. pages 101–110, 05 2013.

[4] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.