Securing the world's software

自从自动化集成加入到工作流后，社区识别和发现软件的漏洞变得更快更容易了。

The 2020 State of the OCTOVERSE Vol. V

GitHub
octoverse.github.com

03

安全
全球的软件

在这个报告中，我们调查了开源的安全问题：
多少项目依赖于开源软件

存在漏洞的可能性，以及补救的最佳实践。

Finding
Balance
Productivity report ➜

//table of contents

概要

开源是许多信息经济的结缔组织。 您将很难找到一种情况，即您的数据没有通过至少一个开源软件传递。 从银行到医疗保健，我们所有人都依赖的许多服务和技术也都依赖于开源软件。 开源代码充当了全球大部分经济体的关键基础设施，因此开源软件的安全对世界至关重要。

59%

的活跃仓库在未来的一年里讲获得包生态的安全警报

作为世界上最大的开源平台，GitHub处于一个独特的位置，因而它能够分析出开源软件的依赖和依赖漏洞带来的影响，

并大规模地提醒用户解决这些问题。我们在Github内对漏洞进行报告，报警，补救，这使我们能够确定开源安全的重要趋势。

在本节的分析中，我们将开源安全和漏洞的生命周期结合在一起，找出我们作为一个社区可以提高开源安全性的关键机会。它还确定了软件团队可以集中资源去改进的领域。

17%

的漏洞明显是恶意漏洞，

但只有0.2%触发了警报

重大发现

01

GitHub上的大多数项目都依赖于开源软件。

我们看到最频繁使用开源依赖的是JavaScript(94%)、Ruby(90%)和 .Net(90%)。

02

支持包生态的活跃仓库在未来的12月中有59%可能收到安全警报。

在过去的12个月中，Ruby(81%)和JavaScript(73%)的仓库最有可能收到警告。我们的分析还将警告按严重程度划分。

03

安全漏洞通常在被披露之前的四年多里都没有被发现。

一旦确定了漏洞，包维护者和安全社区通常会在4周内作出反应，创建并发布修复补丁。这突出表现了漏洞检查在安全社区对漏洞修复发挥的重要作用。

04

大多数软件漏洞是不小心产生的，而非恶意。

对来自我们六大生态的521条漏洞进行随机抽样分析，发现17%的漏洞可以确认是恶意漏洞，如后门尝试。这些恶意漏洞通常出现在很少使用的软件包中，但仅引发0.2%的警报。恶意攻击更容易受到安全领域的关注，然而大多数漏洞都是由失误产生的。

05

自动化加速了开源供应链的安全性。

构建了依赖机器人进行自动 pull request 的仓库在修复软件漏洞的速度上比没有依赖机器人的快1.4倍或者早修复13天。可见把安全检查集成到开发工作流中，并扩大它的影响，是一个帮助团队“左移”的好方法。

行动

行动起来对抗漏洞吧！

01

定期检测你的依赖.

第一步是了解，你不能修补你不了解的东西。很少有人为私有库启用了警报，但这也导致它们面临安全威胁。通过自动警报，公司和开源项目可以随时了解安全漏洞、信息和修补程序。

02

如果你在安全团队，请加入社区

开源是关键的基础设施，我们都应该为开源软件的安全做出贡献。一种贡献方式是在您使用的开源代码中寻找安全漏洞，并向维护人员报告您私下发现的任何漏洞。另一种贡献的方式是使用 CodeQL 搜索您自己的代码中的漏洞，然后共享您的查询，以帮助其他人做同样的事情。

03

使用自动化来保证代码安全和修复漏洞的效率.

使用自动警报和补丁工具快速保护软件意味着漏洞正在变少，使攻击者更难利用。自动生成 pull request 来更新漏洞依赖的库比那些不这么做的库更新软件的速度快1.4倍。安全自动化可以帮助您的团队保护代码，因为安全自动化可以帮助你的团队像其他在社区中分享的开发者一样保护你们的代码, 消除工程中的安全隐患，并扩展你们团队的专业知识。

04

快速修复漏洞并使代码保持最新状态。

应该尽早并经常对软件进行补丁，以及使用已知的安全补救措施来保护它。延迟修复意味着您可能受到攻击，并且可能会给下一次依赖更新带来麻烦。您还应该及时将您的代码库更新到最新版本，以此得到安全更新和社区专家的支持。小的延迟更新可能会被堆积到数年后，而落后会对版本的可用性产生重大影响(最常见的依赖版本可能是最安全的，不常见的版本收到的关注会更少);维护(旧版本的开源支持较少，所以你需要自己完成);甚至招聘(没有人想为失去社区支持的过气版本工作)。

数据

本节的数据来自于 GitHub 的依赖安全功能和六大包生态。对比自2019年10月1日到2020年9月30日以及2018年到2019年的同期数据。

本节报告中的数据分析自超过45000个满足以下条件的仓库。

· 使用了六大包生态里的依赖。

- 是活跃的仓库，定义为从2018年10月到2020年9月，每个月至少有一个贡献。这意味着只有在两年内处于活跃状态的库才会纳入分析中，其中还不包括新的仓库。
- 启用依赖图，主要是公共仓库，默认情况下启用
- 非fork仓库，也不是垃圾邮箱或者 Github 官方所有

我们报告中包括的包生态及其所代表的语言：

包生态	语言
Composer	PHP
Maven	Java
npm	JavaScript
NuGet	.NET
PyPI	Python
RubyGems	Ruby

这节报告中我们提及了包生态。包生态是以统一的打包方式集成的库集合，这有利于代码复用。大多数编程语言都有一个单一的包生态，即使它们有多个包管理器。

我们的报告包括了基于现有数据列出的一系列生态系统的数据。例如，我们的分析不包括来自使用 Gradle 包管理器的 Java 仓库的数据，也不包括来自使用 Poetry 或 Conda 的 Python 仓库数据。虽然这带来了一些限制，但我们仍然可以获得关于安全性和最佳实践的有效见解。

开源安全

开发人员担心在编写代码或添加新的依赖时会引入新的安全缺陷，这是一种风险。反过来另一种风险是:过时的代码和过时的依赖意味着攻击者有时间利用每个已知的漏洞来系统地攻击系统。因此，开发人员应该采用主动检测和自动化来防止或限制 bug 在生产中的影响。为了取得成功，我们需要考虑代码中的所有漏洞:包括我们编写的代码，以及我们所依赖的开源软件。

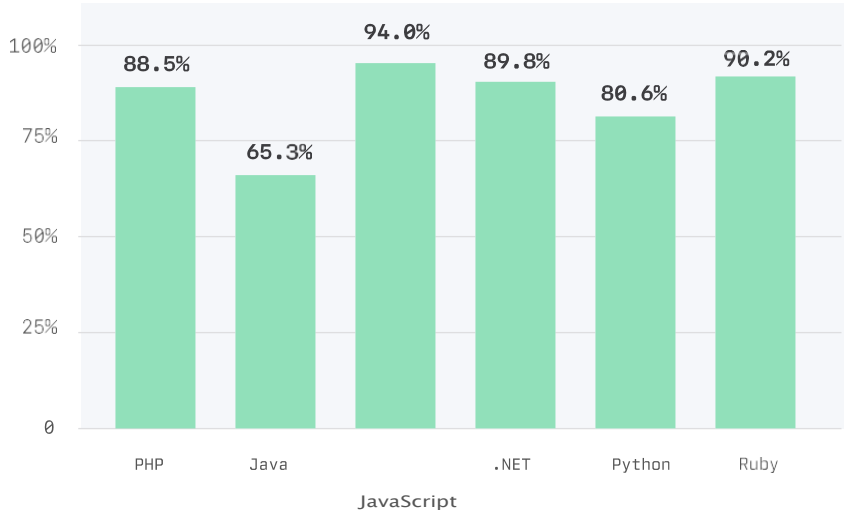
接触面

安全漏洞可以直接或间接通过其依赖（为了让软件包顺利运行而引用和使用的代码）来影响软件。也就是说，代码可能因为包含漏洞，或者因为它依赖于包含漏洞的依赖而易受攻击。在现代软件中，大多数应用程序中有超过80%的代码来自依赖，因此我们研究了包生态系统及其典型的依赖特性。

简单来说，漏洞是攻击者可以利用的任何弱点，包括内部控制、安全程序、程序运行以及计算机系统中的缺陷。为了进行分析，我们将重点放在可以通过软件利用的漏洞上

首先，我们说一下引用至少一个开源依赖的库的百分比。我们看到最频繁使用开源的是 JavaScript(94%)、Ruby(90%)和。net(90%)。可以看到 Java 很可能在我们的数据集中更低，因为使用 Gradle 作为包管理器的库中的依赖信息对我们来说是不可用。我们期望可以得到这些程序的编写和绑定方式。

使用开源软件的活跃仓库的百分比



在每个库中，我们检查每个包生态的依赖数量。在检查直接依赖关系时，我们发现，在所有库中，JavaScript 的依赖关系中位数最高(10)，其次是 Ruby 和 PHP (9)和 Java (8)，而 net 和 Python 的依赖关系中位数最少 (6)。这显示了语言之间直接依赖的中间值有一些变化，但变化不大。

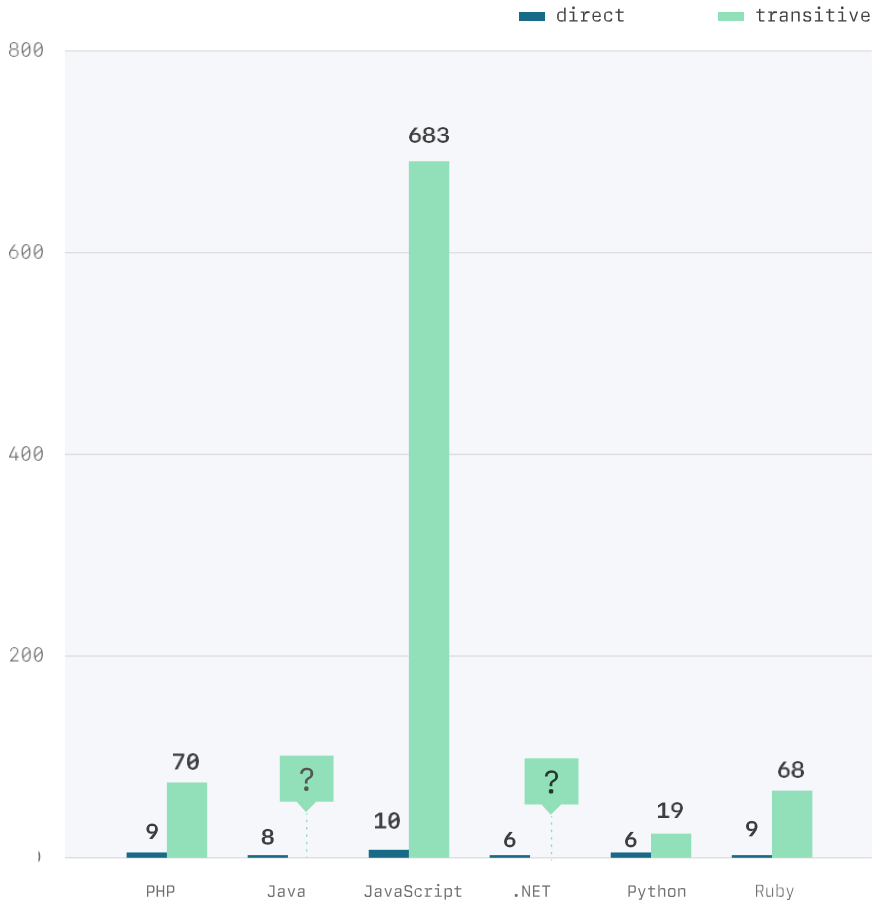
但直接依赖并不是全部。每个直接依赖项本身可以有自己的依赖，而依赖又可以有进一步的依赖，依此类推。我们将任何不是“直接”的依赖称为“传递依赖”。对于那些在 lockfiles 中包含传递依赖细节的语言，以及带有 lockfiles 的库，JavaScript 的中间依赖数最高，为 683，其次是 PHP (70)，Ruby (68)，和 Python(19)。

[1]我们通过查找该生态系统的非锁定清单文件（Gemfile，packages.json，pom.xml等）来收集直接依赖。 我们通过查找锁定文件（匹配形式为“.lock”或“-lock.json”）来收集传递依赖)

[2]其中包括JavaScript（npm），Ruby（RubyGems），PHP（Composer）和 Python（使用 Pipenv 的库）的数据，但不包括 Java（Maven）或 .NET（NuGet）生态的数据。

[3] JavaScript 依赖数与其他语言之间存在数量级差异可能是由 npm 的“微包”哲学（甚至将一线函数作为依赖包装）和 JavaScript 标准库的小尺寸以及 JavaScript 所处的复杂环境（通常在浏览器中使用）共同驱动的。 微包很少在应用程序中使用（即，直接依赖），而通常会在库中使用（显然这是传递依赖）。

按包生态系统划分，每个包的直接依赖中位数和传递依赖中位数



请注意，我们没有有关Java和.NET的传递依赖数据（如图中的“？”所示），并且传递依赖项栏仅表示包含在锁定文件的库的中位数

不会是失误的恶意: 窃听门和后门

后门是故意植入软件中以利于利用的软件漏洞。 错误门是一种特殊的后门，伪装成方便利用但难以发现的错误，与引入明确的恶意行为相反。

后门的模棱两可性质使它们很难被界定，而确认意图可能会特别具有挑战性。 一个隐秘的后门程序与正常的编程错误是无法区分的。 因此，我们需要依靠其他指标来确定可疑的后门事件的意图。

最为明显的后门迹象是，攻击者通常会通过帐户劫持（例如 2018 年的ESLint 攻击）获得对软件包源代码库的提交访问权限，该攻击使用了受感染的软件包窃取而来的用户 npm 软件包注册表凭据。 防范这些后门尝试的最后一道防线是在开发流中进行仔细的代码评审（尤其是对新提交者所做的更改） 。许多成熟的项目都进行了认真的代码评审。 攻击者已经意识到了这一点，因此他们经常试图在发行版本之外控制软件，或者通过例如对软件包名称进行域名抢注来诱使人们获取代码的恶意版本。

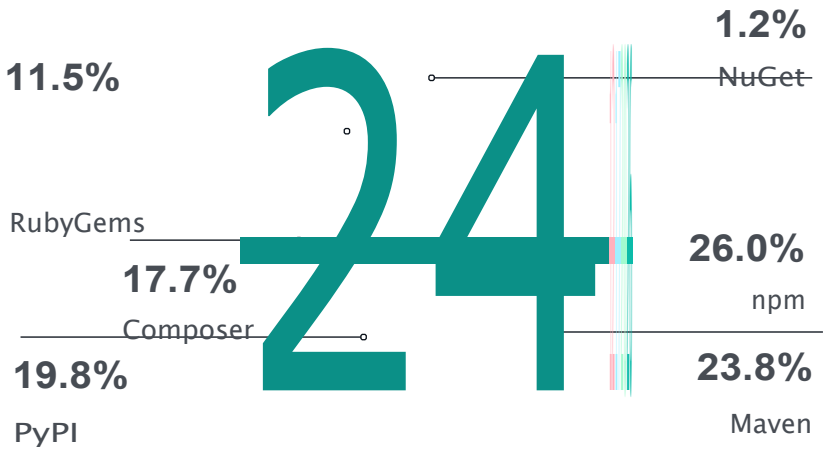
对来自我们六个生态系统的 521 个漏洞的随机样本进行的分析发现，其中17%的漏洞与诸如后门尝试之类的明显恶意行为有关。 在这17%中，绝大多数来自 npm 生态系统。 尽管17%的恶意攻击将在安全圈中引起关注，但错误引入的漏洞可能具有破坏性，并且更有可能影响受欢迎的项目。 在 GitHub 发送给开发人员的通知中可以看到，依赖存在的漏洞，只有 0.2% 与明显的恶意活动有关。 也就是说，大多数漏洞仅是由失误引起的。

维护开源代码的可信度所面临的挑战很大一部分是，让志愿者可以正常地提交代码以确保下游使用者的代码完整性和一致性。 这需要更好地了解项目的贡献图，一致的代码评审，提交和发布签名以及通过多因素身份验证（MFA）确保帐户的安全性。

New advisories

在分析的这个阶段，我们包含了一个额外的数据来源:GitHub 漏洞数据库，该数据库包含了一个安全漏洞列表，这些漏洞被映射到 GitHub 依赖图跟踪的包上。

包生态中的漏洞



这里我们可以看到，npm 和 Maven 在 GitHub 的漏洞数据库中所占比例最高，分别为 26% 和 23.8%，NuGet 最少(1.2%)。但从严重性来看，并不是所有的漏洞都是一样的。

[4] 我们最初的分析将大量 npm 漏洞导入漏洞数据库，这是因为在GitHub 收购 npm 之后将 npm 安全数据库合并到 GitHub 漏洞数据库中去。 此次导入包含 738 个漏洞，占数据库的24%，这使 npm 主导了我们正在探索的趋势数据。我们在分析中排除了这一重要因素，以确保报告可以反映出观察到的趋势，但是请注意，npm 在我们的分析中呈现出不同的模式。 无需大量 npm 数据导入即可显示按软件包生态系统分布的漏洞分布。

额外的数据: 安全漏洞

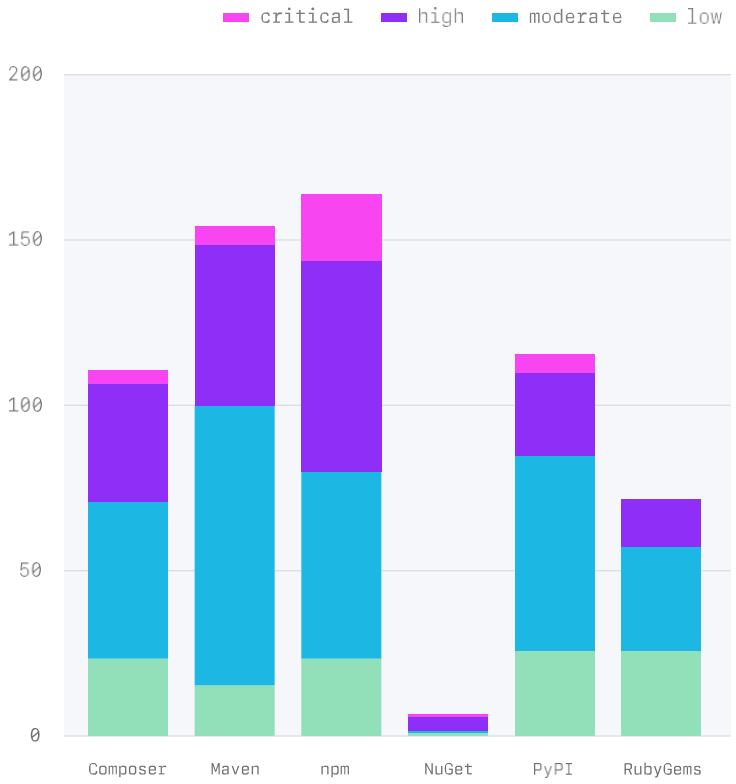
在分析的这个阶段，我们包含了一个额外的数据来源:GitHub 漏洞数据库，该数据库包含了一个安全漏洞列表，这些漏洞被映射到 GitHub 依赖图跟踪的包上。

本报告中的漏洞来自两个来源:外部生态系统的安全漏洞和 GitHub 维护人员报告的安全漏洞，前者占我们分析的54%，后者自2019年5月推出以来，已经占了剩下的46%。

外部生态系统：包括国家漏洞数据库、RubySec、FriendsOfPHP 和其他一些偶尔使用的资源。GitHub 会仔细地验证第三方的 feed 建议以及任何维护人员发布的漏洞，以便将这些漏洞涵盖在漏洞数据库中。我们评估严重程度，确认受影响的版本范围，并寻求补救的办法。

GitHub 安全漏洞：允许维护人员直接在 GitHub 上描述、修复以及公布他们代码中的漏洞。GitHub 会审查所有发布的安全漏洞，并在适当的时候为这些漏洞发布常见修改建议和暴露 (CVE) id。这使得它们被公布在国家漏洞数据库中，从而被全球软件界广泛获得。我们可以这样做，因为我们是 CVE 编号机构，同时也是 CNA 机构。

各大生态的漏洞分布以及漏洞严重性



这里我们看到 npm 有最严重（n=23）和最多（n=66）的漏洞，Maven有最多的中等漏洞（n=86）。RubyGems 没有致命性的漏洞，总体而言，NuGet 的漏洞最少。

尽管其他生态系统有详细的、由社区管理的生态系统资源来提供它们的建议，但这些资源对于 NuGet 来说仍然相当有限，而且关键的是，它们不能被机器读取。因此，尽管 NuGet 的建议似乎比其他生态系统少，但这并不意味着它更安全。

漏洞是怎么被评分的

除了已知在野外被积极利用的相对较少的漏洞，“严重性”是一个有点主观的概念。通常，安全专家会查看可用信息并做出判断，从而为严重性评分。

通用漏洞评分系统（CVSS）是帮助标准化的常用工具。 可从国家漏洞数据库（NVD）获得在线CVSS计算器。 通用漏洞评分系统（CVSS 3.1）中定义了四个级别：低，中，高和严重。 漏洞的级别取决于多种因素，例如，利用难度如何以及成功利用漏洞的影响可能有多大。

当漏洞存在于广泛使用的应用程序中并且可以使用有效的概念验证漏洞（PoC）时，分配准确的分数相对容易。 但是，当没有可用的 PoC 或库中存在漏洞时，评分变得更加主观，这意味着可利用性取决于库的使用方式。 像所有CVE编号颁发机构一样，GitHub在分配严重性时会努力客观地遵循CVSS。 我们还与NVD积极合作进行CVMAP流程，该流程使用美国政府安全研究人员客观评估所有提交给NVD的漏洞的严重性评分。

Security alerts

漏洞修复过程的一个重要部分是了解和跟踪软件清单，与安全建议匹配，然后在出现相关漏洞时发出警报。这涉及到识别易受攻击的组件和相应的漏洞，以便团队能够采取适当的措施来确保他们的代码安全。

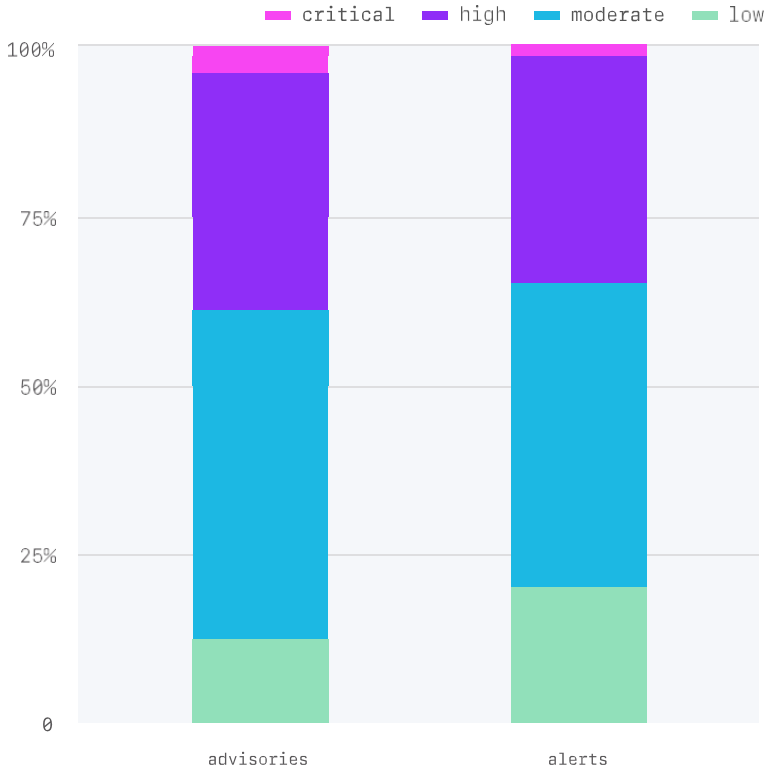
Additional data: alerting

现在，我们在分析中添加了一个额外的数据源：reliabot。默认情况下，Reliadot会提醒开发人员公共库中存在易受攻击的依赖，开发人员可能会选择退出。相反，私有库是可选的，开发人员必须在个人或组织级别启用可靠警报。因此，并非所有库都会收到警报。对于此分析，我们捕获发送给开发人员的警报。

接收到依赖机器人警报的活跃存储库的百分比

总的来说，具有59%支持包生态系统的活跃库获得安全警报。按照包生态系统进行细分，我们发现库最有可能得到警告的是 RubyGems（81%）和 npm（73%）。

严重性级别的漏洞和严重性级别的自动警报（通过依赖机器人）



在这里，我们可以看到按严重程度分类的告警和按严重程度分类的自动警报。注意，它们与我们在漏洞数据库中看到的漏洞的比例有所不同。最大的区别在于低严重程度(警报的24%，高于漏洞数据库中的警告比例)和严重程度(警报的2%，低于漏洞数据库中的警告比例)。这意味着，用户收到的针对较低级别漏洞的警报数量不成比例，尽管这可能意味着，如果没有充分的区分，罕见的、更严重的警报会淹没在噪音中。这还意味着最关键的漏洞不会出现在广泛存在的组件中，因此从一开始就影响到更少的用户。

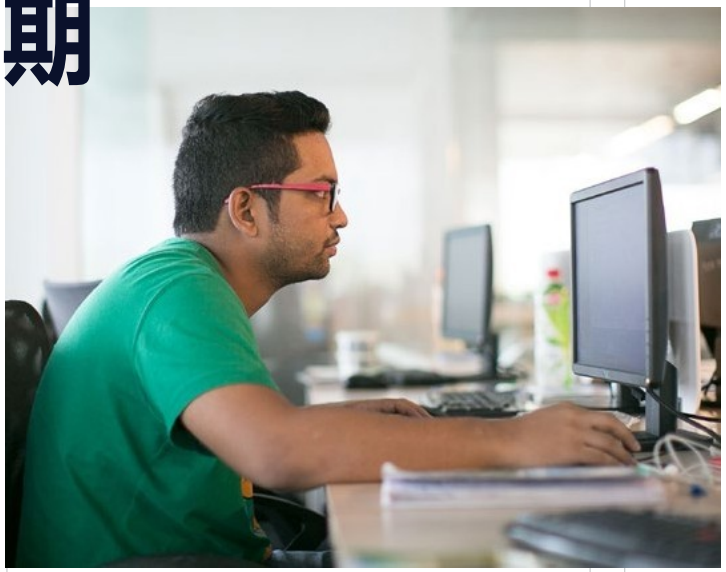
为了了解安全警报是如何在跨包生态的情况下分发，以及它们与漏洞数据库中已知的漏洞有何不同，我们将展示这两种分发版。

每个包的生态系统警报

我们对上一年每个包的生态系统警报进行了细分，
发现依赖漏洞的严重程度与使用该依赖的人数的关系不大。

在过去的12个月里，在最常用的npm包中发现的漏洞的严重性为低或中等。因此我们很容易得出这样的结论：漏洞严重性与其流行程度呈负相关，这可能为“许多眼睛使所有缺陷都变得肤浅”的理论提供了可信度，这意味着最关键的漏洞会在代码审查中被发现。然而，快速查看 Composer 警报的漏洞分布就足以打消我们这种误解。在其最大的软件包中也存在严重的漏洞。事实似乎是，严重的漏洞和轻度的漏洞一样容易通过代码审查。

开源代码漏洞的生命周期



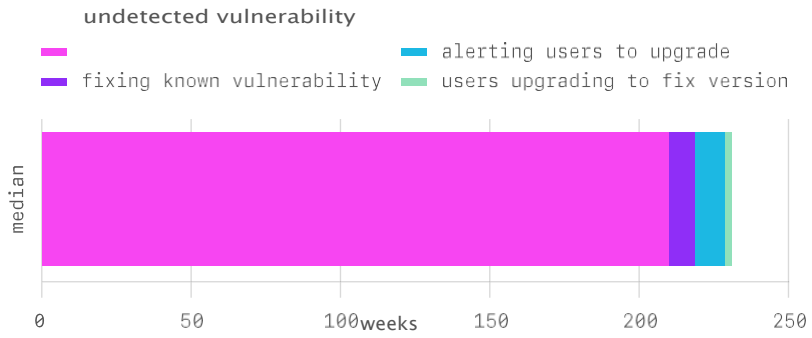
安全漏洞是软件开发和交付的重要组成部分，应用程序的专业安全人员帮助团队和组织保护他们的代码和系统。在本节中，我们将研究漏洞的生命周期，并展示最佳实践如何帮助更快地修复漏洞，从而生成更安全可靠的软件。

开源漏洞修复的四个步骤是

1. 漏洞被识别和报告
2. 维护者修复了漏洞并发布了新版本。
3. 安全工具向最终用户发出安全更新警报。
4. 开发者更新修复后的版本

开源社区中的任何人都可以识别漏洞（第一步），尽管通常是安全研究人员。 然后，维护人员将带头创建修复程序并发布安全更新（第二步）。 在维护者或安全研究人员要求CVE将该漏洞和安全工具将其添加到他们的数据库之后，将通知依赖项的最终用户（第三步）。 然后，这些最终用户将其代码更新为使用新发布的固定版本（第四步）。

一个漏洞完整的生命周期



在发现漏洞之前，通常要经过218周（仅在四年内）才检测到该漏洞。从那里开始，社区通常需要4.4周的时间来识别并发布该漏洞的修复程序，然后需要10周的时间来警告该漏洞的可用性。 我们发现，对于确实应用了此修复程序的库，通常需要一个星期才能解决。通过集中精力及时发现漏洞，可以缩短漏洞的寿命。 这突出了两件事：集中精力发现漏洞对缩短检测时间的重要性，以及当今我们的开源软件中可能存在大量未发现的漏洞。 如果我们的开发工作以恒定的速度引入它们，发现的速度将大大落后于引入的速度。

软件包生态，漏洞数据库以及安全团队警报和补救的方法有所不同。 由于掌握了大量数据，我们将对每个阶段进行更详细的研究，我们的分析重点放在 RubyGems 和 npm 上。

[6] 四年似乎是一个很长的时间。与我们自己对所有漏洞的分析不同的是，兰德报告说的零日漏洞——除了黑客之外，任何人都不知道的漏洞——通常在五年内都不会被发现。

[7] 用于代表漏洞被发现的时间轴的线很可能是倾斜的。因为修复程序通常应用于“X版本或之前”的代码，所以我们捕获了所有潜在受影响的版本的时间线。虽然漏洞经常在更接近于固定版本的提交中引入，但在没有根本原因分析的情况下识别漏洞是不可行的，而且不适合本报告的目的。
因为修复程序通常应用于“X版本或之前”的代码，所以我们捕获了所有那些潜在受影响的版本的时间线。虽然漏洞经常在更接近于固定版本的提交中引入，但在没有根本原因分析的情况下识别漏洞是不可行的，而且不适合本报告的目的。

[8] 十周才能收到警报是由许多其他因素导致的，包括跨几个社区的导入和管理时间。

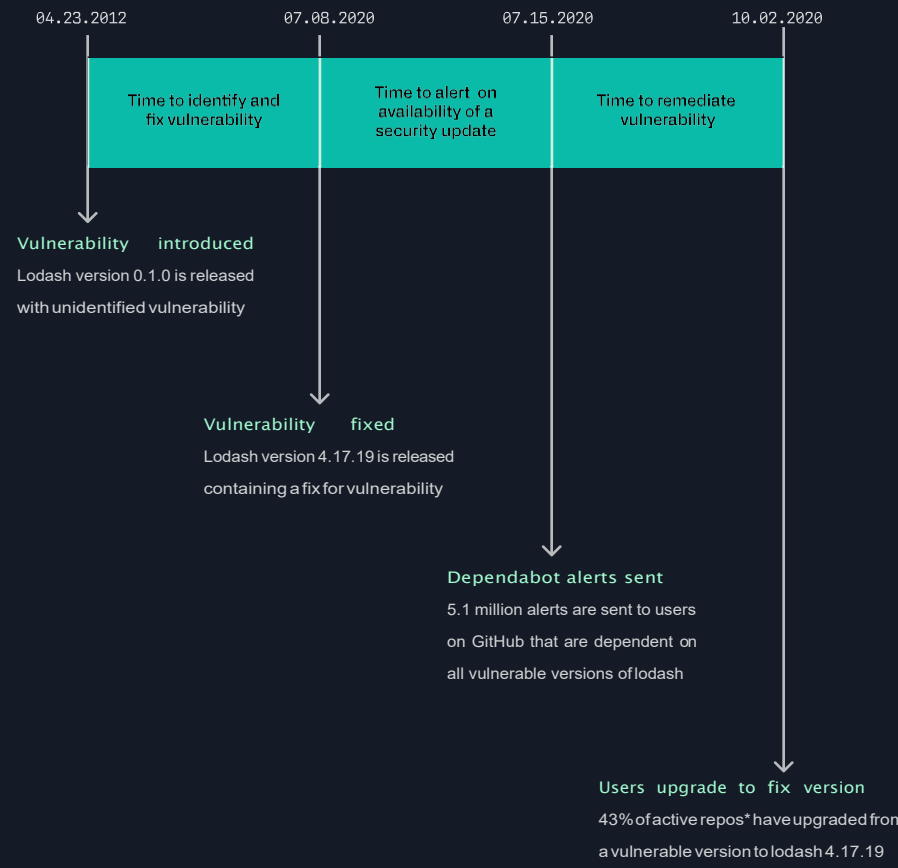
[9] 对于这一分析，我们针对的是前25%对软件进行修复的活跃库;可以从中推断出打算更新的库的典型时间线。

2020最严重的漏洞

截至2020年11月，哪个漏洞最严重？ 这取决于您如何定义“最严重”。 一些明显的候选对象是CVE-2020-0601（又名Curveball），CVE-2020-0796（又名SMBGhost）和CVE-2020-1472（又名Zerologon）。 就它们影响的开发人员数量以及它们对易受攻击的网络和端点的潜在影响而言，这些漏洞非常严重。 这些可能是最糟糕的，因为它们是严重的漏洞，需要系统管理员紧急关注。

但是，最严重的另一个定义是对项目维护者影响最大的漏洞。 根据此定义，CVE-2020-8203（lodash中的原型污染）是一年中最有影响力的漏洞的强有力候选者。 这个漏洞由一手负责造成超过500万个依赖机器人的警报。 那是因为lodash是使用最广泛的 npm 软件包之一。 此外，原型污染是一个潜在的严重漏洞，在最坏的情况下，如果使用 zipObjectDeep 方法，则可能导致远程执行代码。 强烈建议开发人员升级loadash 到最新版本。

loash漏洞的时间线



* An active repo is defined as one with a push in the week before the Dependabot alerts were sent out

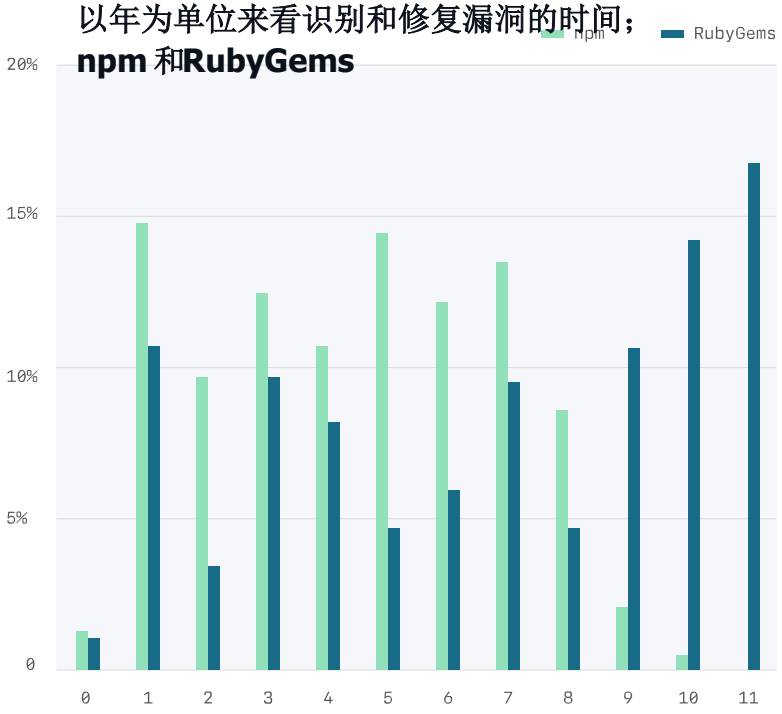
FOR SECURITY RESEARCHERS AND MAINTAINERS

识别和修复漏洞

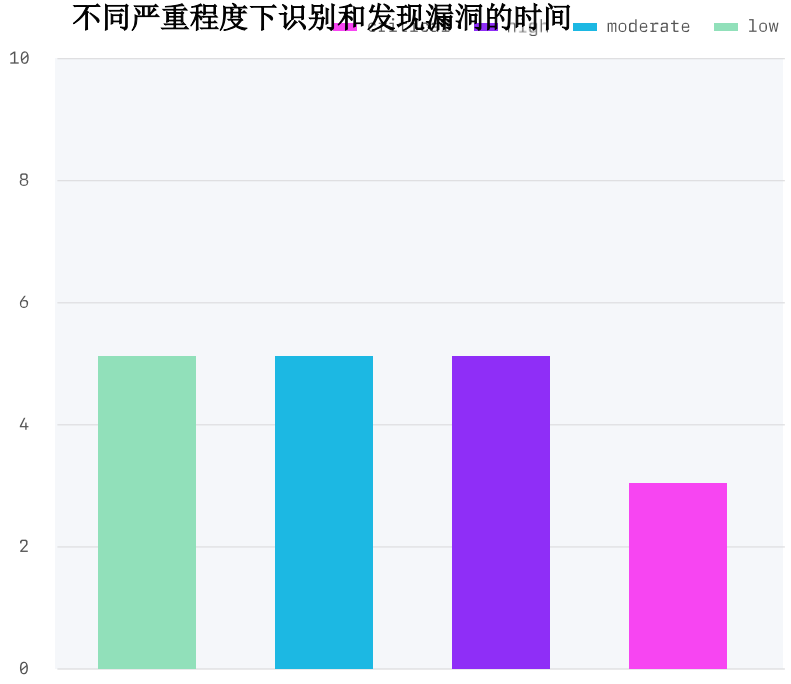
从将漏洞引入生态系统到安全研究人员和维护人员确定修复程序之间的时间，对于RubyGems通常为7年，对于npm为5年。 这是因为软件漏洞通常不会被察觉和发现。 此外，许多团队可能缺乏专业知识（或者只是时间），无法在其代码中查找漏洞，而是专注于开发核心功能。

从严重性来看，我们发现关键漏洞的披露速度更快。 这显然是个好消息，但目前尚不清楚是什么因素推动了这一更快的时间表，值得进行更多研究。

以年为单位来看识别和修复漏洞的时间；**npm 和RubyGems**



不同严重程度下识别和发现漏洞的时间



years

FOR MAINTAINERS ON GITHUB

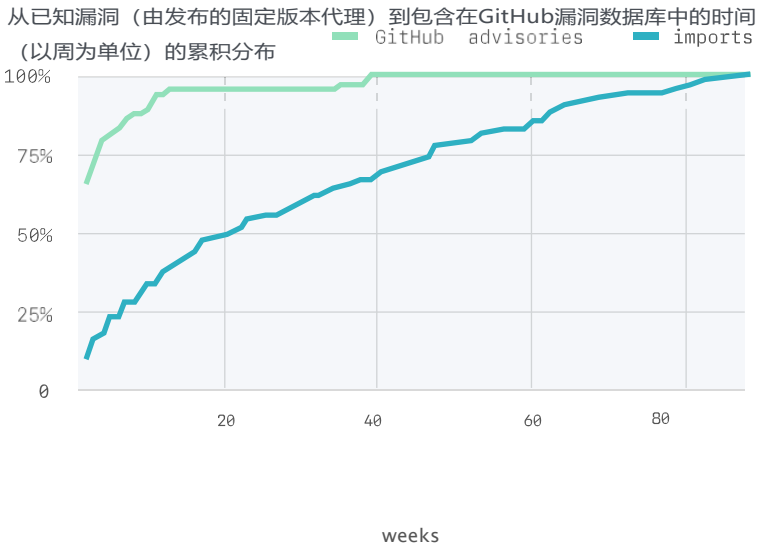
有效的安全更新警报

对于任何安全专业人员，以及更广泛的 DevOps 团队来说，警报时间都是一个重要组成部分。

除了自己发现漏洞之外，接收到关于漏洞依赖的警报后，团队必须第一时间做出响应。

一旦修复版本发布，团队就可以马上修补易受攻击的代码并升级受影响的系统。

Weeks from fix version to Dependabot alerts sent



我们用于分析的警报机制和数据来自GitHub漏洞数据库和依赖机器人的警报。

提示的时间根据提示来源的不同而不同: 从外部源导入或直接提交到 GitHub 漏洞数据库。这种差异来自于提交漏洞的流程。因为 GitHub 的漏洞数据库直接接收提交，所以维护人员甚至可以在修复准备就绪之前起草一个漏洞，并直接从 GitHub 获得CVE。一旦发布，它可以更快地发出警报，通常在一周之内。相比之下，导入的通知在最终进入中央库之前必须通过其他渠道，这可能会导致延迟。这些其他通道可能需要依次通过几个中介，获得CVE，发布修复，并提交给NVD。警报的分布表明，库告警倾向于快速告警，而导入的告警通常在20周后发出警报，并且尾部较长。

纠正安全更新

在理想的情况下，遇到漏洞团队会得到警告，安装补丁，他们的系统就会安全。但是软件是复杂的，知道有补丁可用通常只是补救的开始。尽早打补丁通常是最好的安全策略，但并不总是可以解决问题。团队可能需要确保他们的操作不会中断，可能有太多补丁不能一次合并，或者可能不得不围绕还不支持补丁的功能或遗留平台工作。

知道为安全漏洞打补丁并不简单，我们调查了补救的时间。
此分析包括为所有已解析的库警报的时间。

依赖机器人报警后 npm 和 RubyGems 漏洞被修复的百分比随时间的变化

days after alert was sent

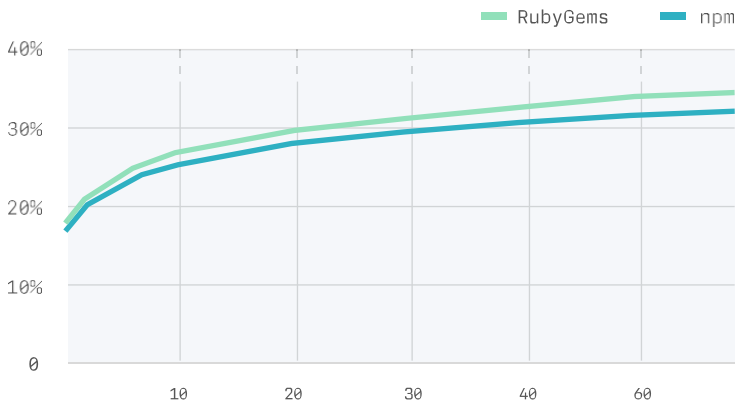
在所有的库中，RubyGems 和 npm 警报后在一天内有接近 20% 的修复率。这个比率会随着时间的推移稳步上升，在发出警报后的一个月内达到 30%。

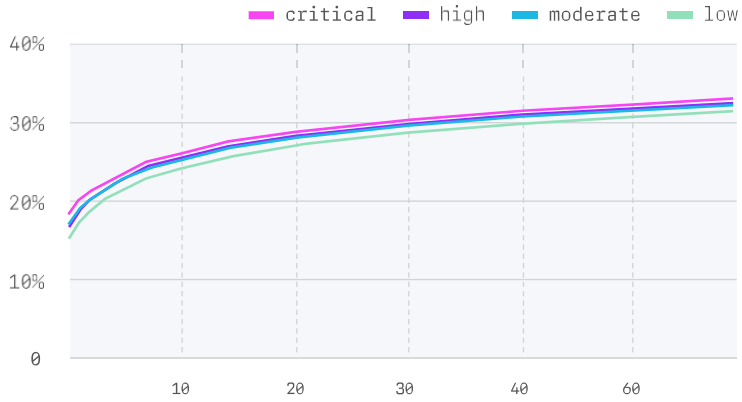
不同严重程度的漏洞在依赖机器人报警后的修复率随时间的变化

days after alert was sent

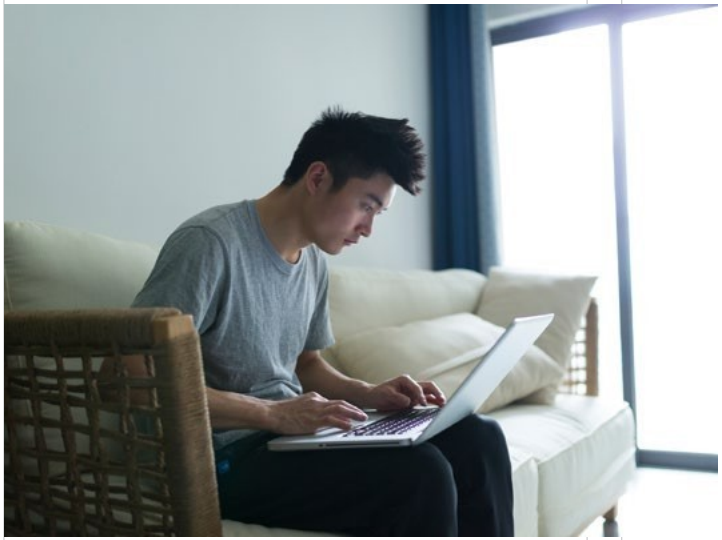
开发人员对更严重的问题反应更快，但差距并不大——所有严重问题的解决率在一天内接近20%。这个比率会随着时间的推移稳步上升，在发出警报后的一个月内达到30%。

22


RubyGems
npm
40%
30%
20%
10%
0
10
20
30
40
60


critical
high
moderate
low
40%
30%
20%
10%
0
10
20
30
40
60

更多的安全自动化



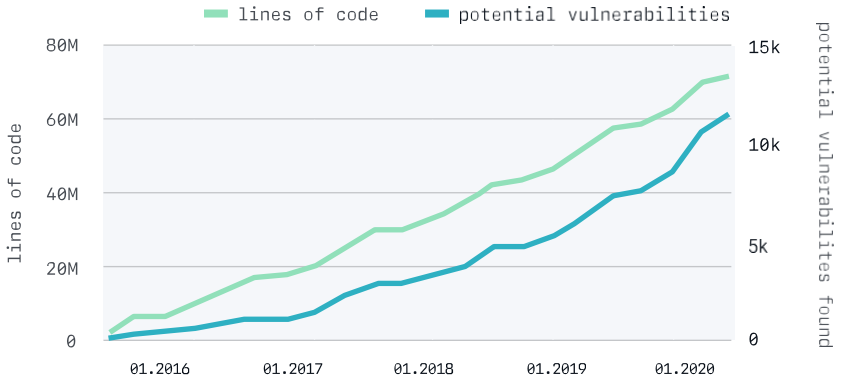
软件变得更安全了吗?我们在发出警报方面做得更好了吗?团队在解决发现的漏洞方面做得更好了吗?这些问题很难回答，部分原因是软件正在成长和变化。随着我们构建新特性和维护基础设施，软件和系统也在不断发展，这意味着我们的漏洞也在不断更新。这促使攻击者保持活跃并时刻准备着新一轮的攻击，因为他们以前利用的漏洞可能随时被新特性替换或修补。与此同时，新的人员加入了我们的团队和项目，并学习如何保护软件和系统。

自动化，比如使用许多开发人员使用开放源码来更快地创建和构建项目，DORA的研究发现，优秀的执行者是
广泛使用开源软件的可能性是低绩效者的1.75倍。

虽然有些人担心开源代码可能有看不见的依赖和漏洞，但在使用软件时，安全性始终是一个需要关注的问题。我们的分析表明，潜在的漏洞随代码行数的增加而增加。开源的力量和承诺在于社区的力量。通过与数百万开发人员合作，不仅可以构建软件包，还可以识别并修复漏洞，我们可以更快、更安全地构建软件。关键是可以利用自动警报和补丁工具来快速地保护您的软件。

这些潜在的安全漏洞是静态分析出来的。分析对每个提交应用进行相同的静态分析和查询，以查看警报数量如何随时间变化。这代表了一个相对公正的漏洞代理。

源代码中发现的潜在漏洞随着代码行数的增加而增加



我们现在编写的代码中引入的漏洞是否比过去少?对开放源码库提交的分析表明不是这样的。

通过对项目的历史提交运行静态分析，我们可以看到什么时候引入了新的潜在漏洞。我们在五年内对数千个流行的开放源码项目的每次提交运行了 CodeQL，这是 GitHub 的静态分析安全工具，以查看引入漏洞的速度是否随时间而改变。结果如上图所示，这表明在 2020 年编写的一行代码与在 2016 年编写的一行代码一样有可能安全漏洞。

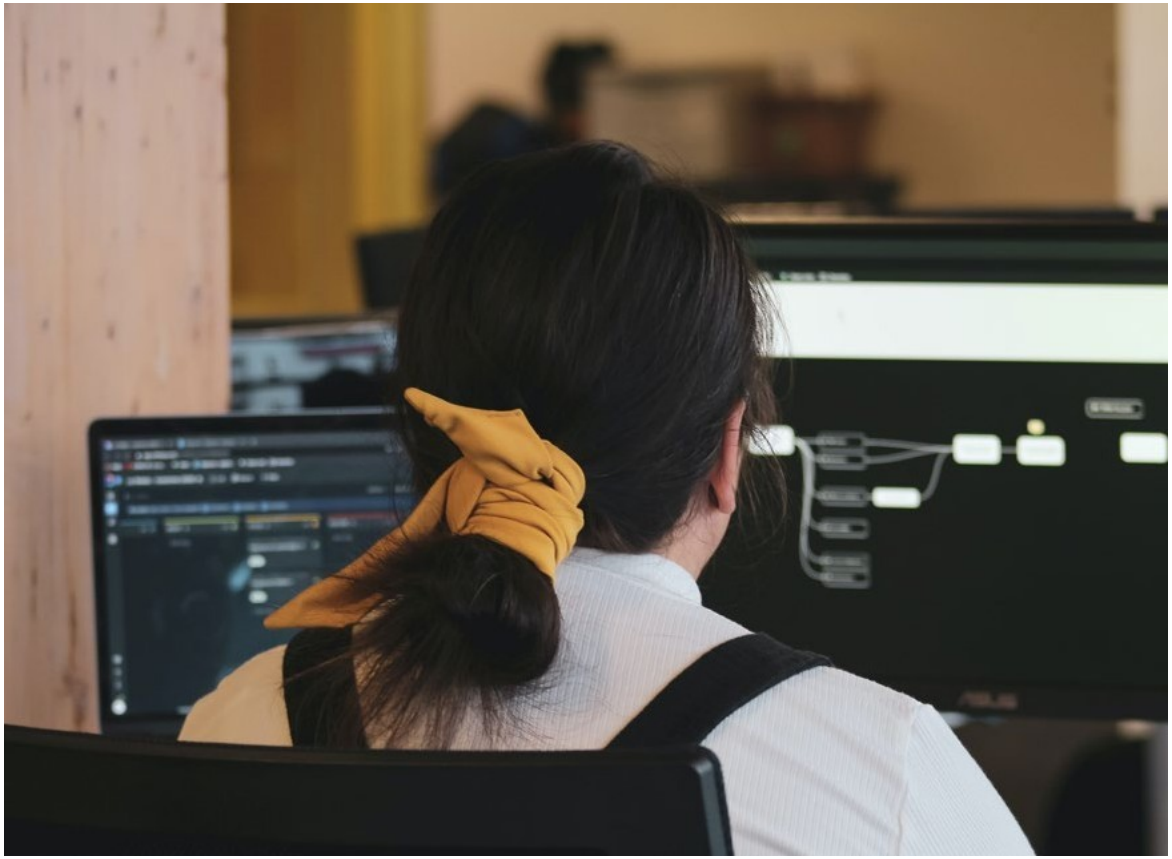
自动化漏洞的补救:左移

DevSecOps专业人士宣称“左移”是一种超级力量，并表示将安全性构建到开发过程中可以增强infosec专业人士的专业技能。但是，这些团队如何“左移”并构建安全的软件呢?

DORA的研究指出，自动化高性能的执行效率可以轻松地让团队开发更安全。我们自己的分析发现，自动生成pull request 以更新到固定版本的库，将在33天内就修复了他们的软件，比那些没有生成 pull request 的库快13天，或者说快1.4倍。使用自动化是一项重要的最佳实践:自动化 pull request 并对安全补丁进行广泛持续集成检查的团队报告，这些对快速更新至关重要。

Sonatype还发现，在高效的软件开发团队成功更新依赖和无损修复漏洞可能比过去快了4.9倍。

随时间变化，解决安全问题的百分比



保护好开源软件的安全是关键

软件安全是每个人的工作。而且这些努力是值得的:拥有良好的自动化和补丁实践可以使我们的开发工作变得更加容易和安全。

更多我们的工作:

Finding balance

Productivity report ➜

Empowering communities

Community report ➜

词汇表

报告中用于分析的方法和名词

27

Active repository 活跃的库

活跃的库是指在分析期间每个月至少有一个贡献的库。

Dependency graph 依赖关系图

该特性列出了仓库的依赖，可以帮助我们识别出一些已知的漏洞。

Developers 开发者

开发者是GitHub上的个人账户，不管他们在做什么

GitHub Advisory Database漏洞数据库

漏洞数据库包含所有策划的CVE和安全漏洞，这些漏洞已映射到GitHub依赖关系图所跟踪的软件包中。

Location 地理位置

开发人员的国家/地区信息是基于其已知的最后位置。对于组织，我们从组织资料中获取最知名的地理位置，或者取自组织成员最活跃的地区.我们仅用总体的地理信息来观察特定国家或地区的增长趋势等信息。地理信息的粒度不应该小于国家级。

Open source projects 开源项目

开放源码项目是具有开放源码许可证的公共库。

Organizations 组织

组织账户代表了GitHub上的用户集合。这些组织账户既有付费的，也有免费的，大的，小的，商业的或者非盈利的都有。

Projects and repositories 项目和仓库

我们可以互换地使用项目和库，尽管我们理解有时一个较大的项目可以跨越多个库。

Vulnerabilities 漏洞

这是代码中的一个问题，它可能会破坏项目或使用其代码的其他项目的机密性、完整性或可用性。漏洞的类型、严重性和攻击方法各不相同。

// 感谢

非常感谢我们的数据科学家、贡献者和审稿人。

按贡献类型的字母顺序列出。

- -

作者: Nicole Forsgren

with contributions from Bas Alberts,

Kevin Backhouse, Grey Baker

数据科学家 Bas Alberts, Grey Baker,

Greg Cecarelli, Derek Jedamski, Scot Kelly,

Clair Sullivan

审稿人: Grey Baker, Dino Dai Zovi,

Denae Ford, Maya Kaczorowski, Alex

Mullans, Kelly Shortridge

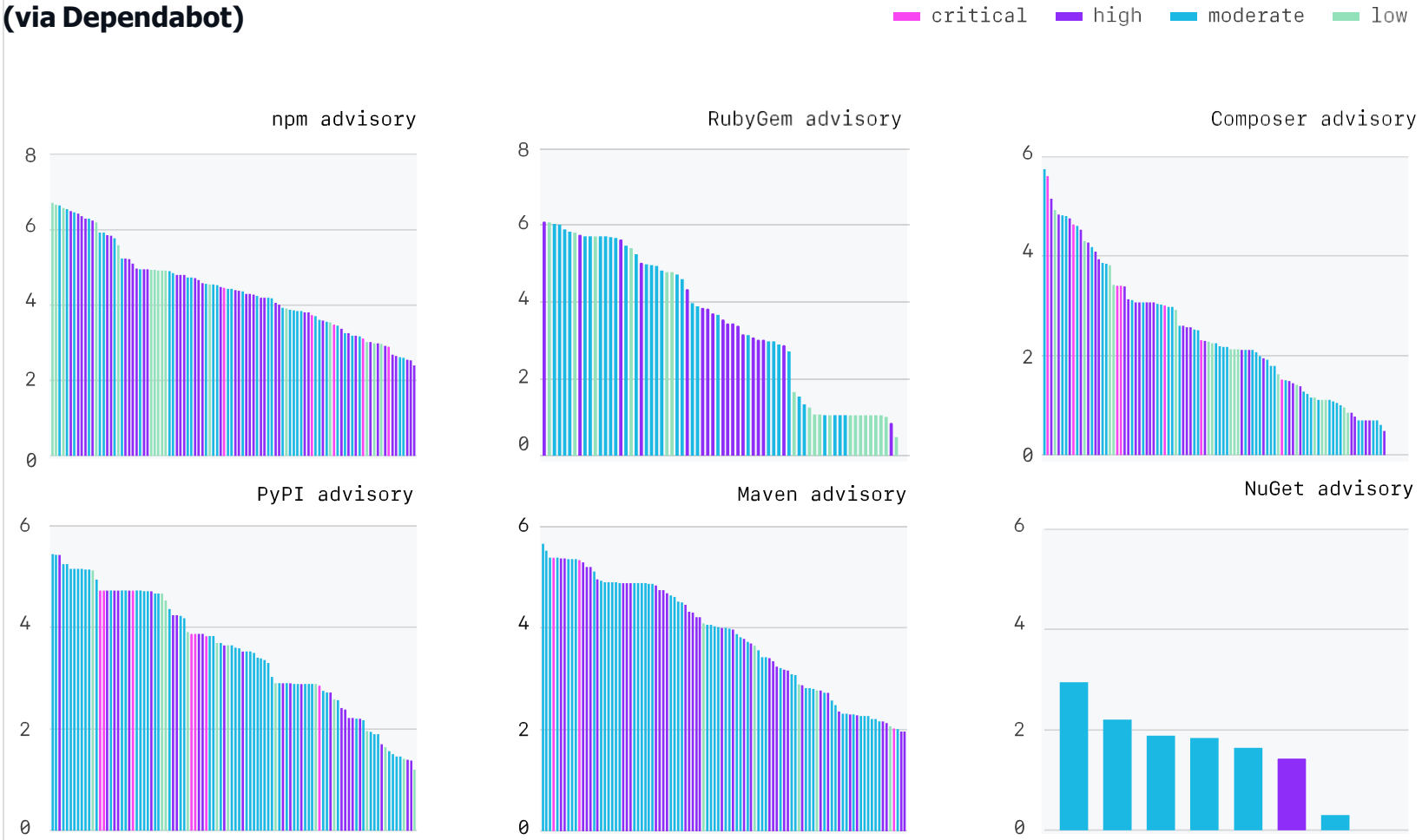
文字编辑: Leah Clark, Cheryl Coupé, Stephanie

Willis

设计师: Siobhán Doyle, Aja Shamblee

// appendix

Alerts sent for each advisory, log scale (via Dependabot)



Alerts sent for each language, log scale

Read more on p17 ➜