1

捉　　虫

捉虫是从软件或硬件中查找 bug 的过程，然而在本书中我们用这个术语特指发现安全攸关的软件 bug 的过程。安全攸关的 bug，也被称作软件的安全漏洞（security vulnerability），攻击者可利用它远程攻击系统、提升本地权限、跨越权限边界，或者严重破坏系统。

大约 10 年前，搜寻软件安全漏洞大多还只是一种业余爱好或引起媒体注意的方法。等到人们意识到这里存在大量的利益时，捉虫才开始慢慢成为一种正式职业。[1]

媒体大量报道软件的安全漏洞和利用这些漏洞的程序(也称作利用程序), 此外 ,有许多书籍和网络资源介绍如何利用安全漏洞 ,还有关于如何披露已发现 bug 的无休止的争论。尽管如此，涉及捉虫过程本身的出版物却少得可怜。虽然软件漏洞和破解这样的术语已被广泛使用，很多人，包括很多信息安全专业人士，并不清楚捉虫人是怎样发现软件安全漏洞的。

问 10 个捉虫人他们怎样在软件里寻找安全相关的 bug ,可能会得到 10 个不同的答案，这就是至今仍没有“ 捉虫秘籍” 之类的书的原因，并且将来可能也不会有。我的目的不是写一本一般的操作指南，而是要描述自己在真实软件中找到这类 bug 的方法和技巧。愿这本书能帮助你形成自己的风格，从而让你也可以找到一些有趣的软件安全 bug。

1.1 兴趣还是利益

人们搜寻软件 bug 的目的和动机各式各样。一些独立的捉虫人希望能改善软件的安全性，而另一些人追逐名利、媒体关注、报酬以及工作机会等个人利益。公司可能需要找出这样的 bug，从而在市场营销中大肆宣扬。当然，也总是有心怀不轨的家伙在寻找新方法侵入你的系统或网络。另一方面，也有人这样做仅仅

因为觉得有趣，没准儿想藉此拯救世界。☺

1.2 通用技巧

虽然没有正式文档描述标准的捉虫过程，但通用技巧还是存在的。这些技巧可以分成两类：静态的和动态的。在静态分析中（通常也称作静态代码分析），我们会检查软件的源代码或者二进制文件的反汇编码，但不会执行软件。而动态分析是在执行软件时对它进行调试或模糊测试。两种技巧各有利弊，大部分捉虫人会将两者结合起来用。

1.2.1 个人技术偏好

多数时候，我推荐静态分析方法。我经常逐行阅读源代码或软件的反汇编码，去理解它。当然，从头到尾阅读源代码往往不大现实，查找 bug 时，我通常首先尝试找出哪里是受用户影响的输入数据得以进入软件的对外接口。这些数据可能来自网络、文件或者执行环境等。

接下来，我会研究输入数据在软件中“游走”的不同路径，留意任何潜在的可被利用来破坏数据的代码。有时候我会在读源代码（见第 2 章）或者反汇编码

(见第 6 章)时标记这些不同路径的入口。有时候我会结合静态代码分析和调试结果(见第 5 章),定位处理输入数据的代码。开发漏洞利用程序时我也倾向于结合静态分析和动态分析两种方法。

发现一个 bug 之后,我会去证明它确实是可利用的。因此我会尝试为它构建一个漏洞利用程序,这样一个程序做好之后,就可以投入大把的时间到调试中了。

1.2.2 代码中潜在的漏洞

静态分析只是捉虫的一种方法,发现代码中潜在漏洞的另一种方法是留心观察靠近“不安全”C/C++库函数的代码,譬如 `strcpy()`和 `strcat()`,寻找可能的缓冲区溢出。或者,可以在反汇编后搜索 `movsx` 汇编指令,寻找符号扩展(sign-extension)漏洞。如果找到了一处漏洞,可以向前追查这段代码,确定它是否暴露了可通过应用入口来访问的漏洞。我很少用这种方法,但许多捉虫人很信赖它。

1.2.3 模糊测试

模糊测试是一种完全不同的捉虫方法,它是一种动态分析技术,由一组提供

非正常输入的测试组成。我不是模糊测试或模糊测试框架的专家，但我知道有的捉虫人自行开发模糊测试框架，用他们的工具能发现大部分的 bug，我自己也常用这种方法来确定用户在哪里输入，有时也用来查找 bug（见第 8 章）。

你可能想知道模糊测试是如何确定用户输入从哪里进入软件的。想象一下你需要检查一个复杂应用的二进制程序的 bug，找出它的各个入口点并不容易，但是复杂应用程序经常因错误的输入数据而崩溃，这在需要分析处理数据文件的应用软件中非常普遍，譬如办公软件、媒体播放器、浏览器等。大多数这样的崩溃与安全性无关（譬如浏览器中发生除零错误），但是它们通常会提供一个入口，于是我就可以开始寻找受用户影响的输入数据。

1.2.4 延伸阅读

现有发现 bug 的方法和技巧是有限的，关于寻找软件中安全漏洞的更多信息，我推荐 Mark Dowd、John McDonald 和 Justin Schuh 合著的 *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*（Addison-Wesley, 2007）一书。如果想了解更多关于模糊测试的信息，可以参考 Michael Sutton、Adam Greene 和 Pedram Amini 合著的《模糊测试：强制性安全漏洞发掘》（*Fuzzing: Brute Force*

Vulnerability Discovery）。

1.3 内存错误

本书提到的漏洞有一个共同点，它们都会导致可利用的内存错误。这样的内存错误会发生在进程、线程或内核出现以下情况时：

- 使用了不属于它的内存（例如空指针解引用，A.2 节详细介绍）
- 使用了分配之外的内存（例如缓冲区溢出，A.1 节详细介绍）
- 使用了未初始化的内存（例如未初始化的变量）[2]
- 使用了错误的堆内存管理（例如二次释放）[3]

内存错误通常发生在错误地使用了 C/C++的某些强大特性时，比如使用显式内存管理或指针算术。

有一类内存错误叫做内存数据损坏（memory corruption）。这种情况往往发生在一个进程、线程或内核修改不属于自己的内存位置，或者对内存内容的修改破坏了其状态时。

如果不熟悉这些内存错误，建议你看一看 A.1 节、A.2 节和 A.3 节，这几节介绍了编程错误的基本概念以及本书讨论的漏洞。

除了可利用的内存错误，还存在几十种其他类型的漏洞，包括逻辑错误和 Web 特有的漏洞，譬如跨站点脚本攻击（cross-site scripting）、跨站点请求伪造（cross-site request forgery）、SQL 注入等。这里只列出一小部分，其他类型的漏洞不属于本书讨论的范围，本书中我们主要讨论可利用的内存错误。

1.4 专用工具

寻找软件 bug 或者构造漏洞利用程序来测试这些 bug 时，我需要借助工具来看清应用程序的内部机制，通常我使用调试器和反汇编工具。

1.4.1 调试器

调试器通常提供附加到用户空间进程或者内核、读写寄存器和内存、用断点和单步控制程序执行流等功能。每个操作系统通常都有自带的调试器，另外还有一些第三方调试器。表 1-1 列出了本书涉及的几种操作系统平台和调试器。

表1-1 本书中用到的调试器

操作系统	调 试 器	内核调试
Microsoft Windows	WinDbg（微软官方提供的调试器）	是
	OllyDbg及其变体Immunity Debugger	否
Linux	The GNU Debugger（gdb）	是
Solaris	The Modular Debugger（mdb）	是
Mac OS X	The GNU Debugger（gdb）	是
Apple iOS	The GNU Debugger（gdb）	是

这些调试器将用于定位、分析和攻击我所发现的漏洞。B.1 节、B.2 节和 B.4 节介绍了一些调试器常用的命令清单。

1.4.2 反汇编工具

要检查一个没有源代码的程序，可以通过读汇编代码来分析。调试器都有反汇编进程或内核代码的功能，但往往不是很直观，不那么好用。Interactive Disassembler Professional（IDA Pro）[4]填补了这一空白，它支持 50 多种处理器系列，提供了良好的交互性、可扩展性，还有代码图解表示功能（code graphing）。要分析程序的二进制码，IDA Pro 是必备工具，详细的使用介绍可参考 Chris Eagle

的《IDA Pro 权威指南（第 2 版）》

1.5 EIP = 41414141

为了演示这些 bug 所暗含的安全问题，我将介绍如何通过控制 CPU 的指令指针（IP）来取得对漏洞程序执行流的控制。指令指针寄存器或者程序计数（PC）寄存器里存放的是当前代码段中下一条将要执行指令的偏移量。[5]控制了这个寄存器，就完全控制了这个漏洞进程的执行流。我将把这个寄存器的值修改成 0x41414141（ASCII 字符串 AAAA 的十六进制表示）、0x41424344（ASCII 字符串 ABCD 的十六进制表示）或者其他类似的值，来演示对指令指针的控制。所以在后面的章节中，如果你看到 `EIP = 41414141`，就意味着我已经取得对漏洞进程的控制权。

指令指针/程序计数器：

- EIP——32 位指令指针（IA-32）
- RIP——64 位指令指针（Intel 64）
- R15（即 PC）——Apple

一旦控制了指令指针，就有很多方法可以把它变成一个完全可行、可用作武器的漏洞利用程序。关于漏洞利用程序开发的更多信息，可以参考 Jon Erickson 的《黑客之道：漏洞发掘的艺术》（*Hacking: The Art of Exploitation, 2nd edition*），或者可以输入 exploit writing，在 Google 的海量在线信息中搜索。