

SSH, The Secure Shell: The Definitive Guide



SSH 权威指南

O'REILLY®
中国电力出版社

Daniel J. Barrett & Richard E. Silverman 著
冯锐 由渊霞 译

SSH 权威指南



快点用 SSH 保护你的计算机网络吧！SSH（Secure Shell）具有透明、保密性强、公钥认证可信的特点，并具有高度可配置的客户端/服务器体系结构，是基于 TCP/IP 协议、目前十分流行且健壮的网络安全与隐私保护解决方案。SSH 支持安全远程登录，可用于计算机之间安全地传输文件。它特有的“隧道”功能能够保护其他一些原本并不安全的网络应用程序。而所有这一切中最重要的一点是，SSH 既有免费版本，也有功能极其强大的商用版本。

本书从系统管理员和普通用户两种视角深入讨论了 SSH 的技术细节，揭开了 SSH 手册页的神秘面纱，具体内容包括：

- SSH1、SSH2、OpenSSH 以及 F-Secure SSH 的 Unix、Windows 和 Macintosh 版本，包括基本概念、技术内幕以及复杂应用
- SSH 服务器与客户端的配置，包括服务器范围配置和每账号配置，还推荐了一套安全性最强的设置方法
- 基于代理的高级密钥管理、密钥转发、强制命令
- 深入探讨了 TCP 及 X11 应用程序的转发（即隧道传输），进而讨论了存在防火墙与网络地址转换（NAT）机制时的情况
- SSH 与 Kerberos、PGP、PAM 等其他安全产品的集成
- 目前流行的 SSH 实现中的一些没有文档资料记载的行为
- SSH 系统的安装与维护
- 疑难解答：既包括常见的问题，也包括不是很常见的问题

不论你的通信是在一个很小的 LAN 上进行，还是要跨越整个 Internet，SSH 都可以将数据安全有效地从“这儿”传输到“那儿”。所以请抛弃掉那些不安全的 *rhosts*、*hosts.equiv* 文件，升级到 SSH 上来吧，你的网络必将成为一块真正安全的生存和工作空间。

ISBN 7-5083-1085-3

9 787508 310855

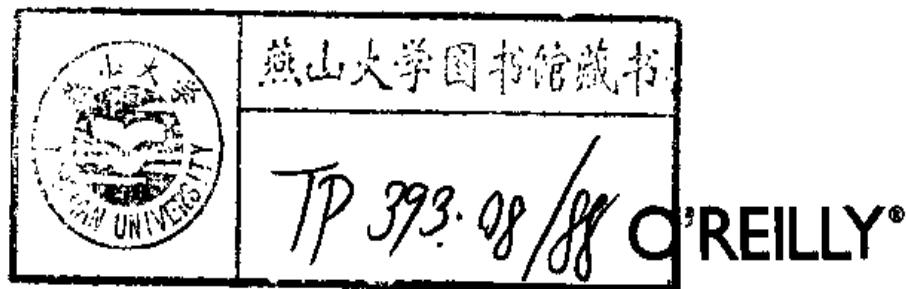
O'Reilly & Associates, Inc. 授权中国电力出版社出版

ISBN 7-5083-1085-3

定价：69.00 元

SSH 权威指南

Daniel J. Barrett & Richard E. Silverman 著
冯锐 由渊霞 译



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

O'Reilly & Associates, Inc. 授权中国电力出版社出版

中国电力出版社



0747852

图书在版编目 (CIP) 数据

SSH权威指南 / (美) 巴勒特 (Barrett, D.), 斯夫曼 (Silverman, R.) 著; 冯锐,
由渊霞译. - 北京: 中国电力出版社, 2002.11

书名原文: SSH, The Secure Shell: The Definitive Guide

ISBN 7-5083-1085-3

I.S... II. ①巴 ... ②斯 ... ③冯 ... ④由 ... III. 计算机网络 - 安全技术 IV. TP393.08

中国版本图书馆 CIP 数据核字: (2002) 第 084736 号

北京市版权局著作权合同登记

图字: 01-2002-1191 号

©2001 by O'Reilly & Associates, Inc.

Simplified Chinese Edition, jointly published by O'Reilly & Associates, Inc. and China Electric Power Press, 2003. Authorized translation of the English edition, 2001 O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly & Associates, Inc. 出版 2001。

简体中文版由中国电力出版社出版 2003。英文原版的翻译得到 O'Reilly & Associates, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly & Associates, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书 名 / SSH权威指南

书 号 / ISBN 7-5083-1085-3

责任编辑 / 陈维宁

封面设计 / Ellie Volckhausen, 张健

出版发行 / 中国电力出版社 (www.infopower.com.cn)

地 址 / 北京三里河路 6 号 (邮政编码 100044)

经 销 / 全国新华书店

印 刷 / 北京市地矿印刷厂

开 本 / 787 毫米 × 1092 毫米 16 开本 38 印张 559 千字

版 次 / 2003 年 4 月第 1 版 2003 年 4 月第一次印刷

印 数 / 0001-5000 册

定 价 / 69.00 元 (册)

O'Reilly & Associates 公司介绍

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

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

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly & Associates 一直处于 Internet 发展的最前沿。

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

赵立伟
2001年1月

作者简介

Daniel J. Barrett, 博士, 从 1985 年就开始深入研究 Internet 相关技术。他目前供职于一家著名的金融服务公司, 是软件工程师和副总裁。Dan 还是重金属歌手、Unix 系统管理员、大学讲师、Web 设计师和幽默大师。

Dan 已经为 O'Reilly 撰写了多部著作, 其中包括《NetResearch: Finding Information Online》和《Bandits on the Information Superhighway》, 他每个月还为《Compute!》与《Keyboard Magazine》杂志撰写专栏。现在他和他的家人一起住在波士顿。

你可以用 *dbarrett@oreilly.com* 这个邮箱与 Dan 联系。

Richard E. Silverman 第一次接触计算机是在 1986 年上大学三年级时, 他登录到一台 DEC-20 上, 用“MM”命令给别人发送邮件, 然后就迅速地迷失在那个世界中了。他最终还是清醒过来, 发现并找到了自己的职业, 一份既能给别人带来便利而又有一定迷惑性的职业; 而在这之前他可还没有真正寻找过呢。在获得计算机科学学士学位与理论数学硕士学位之后, Richard 先后工作过的领域有网络、软件开发的规范方法、公钥基础体系、路由安全以及 Unix 系统管理。工作之余, Richard 还热爱阅读、学习语言与数学、唱歌、跳舞及体育锻炼。

你可以用 *res@oreilly.com* 这个邮箱与 Richard 联系。

封面介绍

本书封面上的动物是一只蜗牛(软体类腹足纲)。

作为软体动物的一员, 蜗牛具有柔软、潮湿的躯体, 外面是起保护作用的硬壳, 当它遇到危险或身处干旱、刺眼的环境中时, 该硬壳都能保护它。蜗牛喜欢湿天, 而且它虽然并非夜间活动, 但也会避开明亮的阳光。蜗牛长长的身体前端有两套触角, 它的眼睛在其中一套上, 而另一套则用于嗅探和导航。

蜗牛是雌雄同体动物，每只都具有雌性和雄性的性器官，不过为了授精仍需两只蜗牛配合。一只蜗牛每年产卵约6次，每次近100个。小蜗牛需孵化两个月，于两年后成熟。蜗牛的寿命大约为5~10年。

大家知道蜗牛运动得很慢，它完全是靠下侧肌肉的伸缩来进行小幅移动的。它会留下一条粘液的轨迹，以保护它在觅食时不会被尖硬物体伤害。蜗牛的食物是庄稼、树皮和水果，因此它是害虫，在世界上大多数地方都因毁坏庄稼而声名狼藉。

目录

前言	1
第一章 SSH 简介	9
1.1 SSH 是什么	10
1.2 SSH 不是什么	11
1.3 SSH 协议	12
1.4 SSH 特性概述	16
1.5 SSH 的历史	19
1.6 相关技术	21
1.7 小结	27
第二章 SSH 客户端的基本用法	28
2.1 实例	28
2.2 使用 ssh 建立远程终端会话	29
2.3 增加实例的复杂性	31
2.4 使用密钥进行认证	36
2.5 SSH 代理	42
2.6 不用密码或口令进行连接	48

2.7 各种客户端	48
2.8 小结	51

第三章 SSH 内幕 52

3.1 SSH 特性概述	53
3.2 密码学基础	56
3.3 SSH 系统框架	60
3.4 SSH-1 内幕	63
3.5 SSH-2 内幕	85
3.6 伪装用户访问权限	100
3.7 随机数	101
3.8 SSH 和文件传输 (scp 和 sftp)	103
3.9 SSH 使用的算法	107
3.10 SSH 可以防止的攻击	116
3.11 SSH 不能防止的攻击	120
3.12 小结	123

第四章 SSH 的安装和编译时配置 125

4.1 SSH1 和 SSH2	125
4.2 F-Secure SSH 服务器	149
4.3 OpenSSH	150
4.4 软件清单	154
4.5 使用 SSH 代替 r- 命令	156
4.6 小结	159

第五章 服务器范围的配置 161

5.1 服务器名	162
5.2 运行服务器	163
5.3 服务器配置：概述	166

5.4 开始：原始设置	171
5.5 允许用户登录：认证和访问控制	191
5.6 用户登录和账号	216
5.7 子系统	219
5.8 历史记录、日志记录和调试	221
5.9 SSH-1 和 SSH-2 服务器的兼容性	233
5.10 小结	234
第六章 密钥管理与代理	235
6.1 什么是身份标识	236
6.2 创建一个身份标识	240
6.3 SSH 代理	248
6.4 使用多个身份标识	270
6.5 小结	274
第七章 客户端的高级用法	275
7.1 如何配置客户端	275
7.2 优先级	287
7.3 详细模式简介	288
7.4 深入客户端配置	289
7.5 使用 scp 安全拷贝文件	327
7.6 小结	336
第八章 每账号服务器配置	337
8.1 该技术的局限	338
8.2 基于公钥的配置	340
8.3 可信主机访问控制	360
8.4 用户 rc 文件	362
8.5 小结	362

第九章 端口转发与 X 转发	363
9.1 转发是什么?	364
9.2 端口转发	365
9.3 X 转发	389
9.4 转发安全性: TCP-wrappers 及 libwrap	402
9.5 小结	408
第十章 推荐配置	409
10.1 基础知识	410
10.2 编译时配置	410
10.3 服务器范围配置	411
10.4 每账号配置	416
10.5 密钥管理	416
10.6 客户端配置	417
10.7 远程主目录 (NFS, AFS)	417
10.8 小结	420
第十一章 案例分析	422
11.1 无人值守的 SSH: 批处理或 cron 任务	422
11.2 FTP 转发	429
11.3 Pine、IMAP 和 SSH	451
11.4 Kerberos 与 SSH	459
11.5 跨网关连接	481
第十二章 疑难解答及 FAQ	490
12.1 第一道防线: 调试消息	490
12.2 疑难问题及解答	493
12.3 其他 SSH 资源	513
12.4 错误报告	515

第十三章 其他产品概述	516
13.1 通用功能	517
13.2 将要介绍的产品	517
13.3 产品列表	517
13.4 其他与 SSH 有关的产品	526
第十四章 Sergey Okhapkin 移植的 Windows 版的 SSH1	528
14.1 获取并安装客户端	528
14.2 客户端的用法	533
14.3 获取并安装服务器	534
14.4 疑难解答	537
14.5 小结	538
第十五章 SecureCRT (Windows)	539
15.1 获取及安装	539
15.2 客户端的基本用法	540
15.3 密钥管理	540
15.4 客户端的高级用法	542
15.5 转发	544
15.6 疑难解答	546
15.7 小结	547
第十六章 F-Secure SSH Client (Windows, Macintosh)	548
16.1 获取与安装	548
16.2 客户端的基本用法	549
16.3 密钥管理	550
16.4 客户端的高级用法	551

16.5 转发	554
16.6 疑难解答	556
16.7 小结	558
第十七章 NiftyTelnet SSH (Macintosh)	559
17.1 获取与安装	559
17.2 基本客户端应用	560
17.3 疑难解答	562
17.4 小结	563
附录一 SSH2 手册页, sshregex 部分	565
附录二 SSH 快速参考	569
词汇表	587

前言

隐私权是一项基本的人权，但是在现在的计算机网络中，人们的隐私权并不一定能得到保障。Internet和本地网络上传输的大多数数据都是以明文形式传输的，任何人使用一些技术都可以截获并观察到这些数据。你所发送的电子邮件、在计算机之间传输的文件、甚至是你的口令都可能被别人看到。设想一下，如果一个不可信的第三方（你的竞争对手、CIA或你的亲戚）截获了你传输的最敏感的信息，损失会是多么惨重。

网络安全是个大问题，因此很多公司会采取很多措施来保护自己的信息资产，例如使用防火墙、建立虚拟专用网络（VPN）、对文件和传输进行加密。但是除了采用这些措施之外，很多大公司实际上遗漏了一种小得不起眼但却很健壮有效的解决方案。这种方案十分可靠，相当易用，又廉价，可以在现在大部分操作系统上运行。

这就是SSH，安全Shell（Secure Shell）。

使用SSH保护自己的网络

SSH是一种廉价的软件解决方案，可以防止网络上的数据被窥视。虽然SSH并不能解决所有的隐私权和安全性问题，但是它可以有效地解决部分问题。其主要特性有：

- 具有一个安全的客户端/服务器协议，可以在网络上加密并传输数据。

- 可以使用口令、主机或公钥对用户进行认证（识别），还可以随意集成其他通用的认证系统，其中包括 Kerberos、SecurID、PGP、TIS Gauntlet 和 PAM。
- 可以为一些不安全的网络应用程序（例如，Telnet、FTP 及很多其他基于 TCP/IP 的程序和协议）增加安全性。
- 对于终端用户来说几乎是完全透明的。
- 在大多数操作系统上都可以运行。

本书的读者对象

我们编写本书主要是供系统管理员和技术爱好者使用。本书部分章节可以适用于更广泛的读者对象，其他章节则是纯技术的，适合计算机和网络专业人士阅读。

终端用户

你在不同的计算机上拥有两个甚至更多个账号吗？SSH让你可以在高度安全的基础上从一个账号连到另一个账号上。你可以在这些账号之间拷贝文件、可以从一个账号远程登录到另一个账号中，还可以执行远程命令，所有这些操作都可以绝对放心：没有人会截获你的用户名、口令和所传输的数据。

你要把自己的计算机连接到Internet服务提供商（ISP）吗？特别是，你是连接到ISP的Unix shell账户上吗？如果答案是肯定的，SSH就可以让你的这种连接更加安全。现在越来越多的ISP开始为自己的用户提供SSH服务器。如果你的ISP并没有提供，我们也会告诉你如何自己运行SSH服务器。

你在开发软件吗？你正在创建必须在网络上进行安全通信的分布式应用程序吗？那么你就不要闭门造车了：使用SSH来加密自己的连接吧。这是一项很棒的技术，可以大大减少你的开发时间。

即使现在你只有一个账号，只要这个账号连接到网络上，SSH就依然有用。例如，如果你想让其他人（比如你的家庭成员或者同事）使用自己的这个账号，但是又不想让他们无限制地使用，那么就可以使用SSH，从而为你的账号提供一条控制良好的受限访问通道。

读者基础

我们假设你熟悉计算机，也熟悉现代商业办公场所或连有 Internet 的家庭的网络系统。更理想一点，我们假设你很熟悉 Telnet 和 FTP 应用程序。如果是 Unix 用户，那么就应该熟悉 *rsh*、*rlogin* 和 *rcp*，也应该具有编写 shell 脚本的基础。

系统管理员

如果你是 Unix 系统管理员，那么你就可能知道 Berkeley 的 r-命令 (*rsh*、*rcp*、*rlogin*、*rexec* 等等) 本质上就是不安全的。SSH 提供了一种安全的替代品，不再使用 *rhosts* 和 *hosts.equiv* 文件，而是使用密钥对用户进行认证。SSH 还可以增强系统中其他基于 TCP/IP 的应用程序的安全性，有了加密的 SSH 连接，应用程序就好象进入了一条透明的隧道。随着本书的介绍，你会逐渐喜欢上 SSH 的。

读者基础

除了上一节中终端用户所需的读者基础之外，还应该熟悉 Unix 账号和组、诸如 TCP/IP 和数据包之类的网络概念以及基本的加密知识。

本书导读

本书大体上可以分为三个部分。前三章是有关 SSH 的基本介绍，首先向所有读者介绍高层的理论（第一、二章），然后为技术读者具体介绍技术细节（第三章）。

接下来的九章介绍 Unix 上的 SSH。前两章（第四、五章）为系统管理员介绍 SSH 的安装和服务器级的配置。接下来的四章（第六章到第九章）为终端用户介绍高级特性，包括密钥管理、客户端配置、每账号服务器配置以及转发。最后，我们给出了推荐配置（第十章）、案例详细研究（第十一章）和问题解决技巧（第十二章），这样 Unix 系列就完整了。

其余的章节介绍了 Windows 和 Macintosh 上的 SSH 产品，还简要介绍了 SSH 在其他平台上的实现（第十三章）。

本书中的每节都使用数字进行编号，所有正文中都提供了交叉索引。如果在 7.1.3.2 节中有更详细的讨论，我们使用 [7.1.3.2] 符号来表示。

本书的组织

本书是通过概念而不是语法进行组织的。我们从概述入手，循序渐进地引导读者逐步深入了解SSH的功能。因此我们可能在第一章中介绍一个主题，在第二章介绍其基本用法，在第七章中再介绍其高级用法。如果你喜欢一步到位，可以参考附录二中集中给出的所有命令及其选项。

我们着重介绍了三个级别上的服务器配置，即编译时配置、服务器级配置和每账号配置。编译时配置（第四章）是指在搭建SSH客户端和服务器时就选择适当的选项进行配置。服务器级配置（第五章）通常是系统管理员在SSH服务器运行期间所执行的配置；而每账号配置（第八章）可以由终端用户随时执行。对于系统管理员来说，正确理解这三个级别的配置之间的联系和区别是至关重要的。否则，SSH就显得混乱不堪了。

虽然本书的大部分内容都着重在介绍SSH的Unix实现，但是要理解这些内容并不强求读者是Unix用户。Windows和Macintosh的爱好者可以参考后面专门介绍相应平台的章节，但是大部分细节内容都在Unix的章节中介绍，因此我们建议读者仔细阅读这些章节，至少参考一下。

哪一章适合你？

我们为不同兴趣和具有不同技巧的用户指出几条阅读的途径：

系统管理员

第三章到第五章以及第十章对于理解SSH以及如何搭建并配置服务器来说是最重要的。但是作为安全产品的系统管理员，则应该通读全书。

Unix 用户（非系统管理员）

第一、二章对SSH进行概述，第六章到第九章深入讨论了SSH客户端。

Windows 终端用户

新手应该阅读第一、二章以及第十三章到第十六章，其他用户可以根据自己的兴趣阅读本书。

Macintosh 终端用户

新手应该阅读第一、二、十三、十六章和第十七章，其他用户可以根据自己的兴趣阅读本书。

其他平台的用户

新手应该阅读第一、二、十三章，其他用户可以根据自己的兴趣阅读本书。

即使具备使用SSH的经验，在从第三章到第十二章中你还是可以发现很多有价值的内容。我们在本书中介绍了一些Unix手册页中没有介绍清楚或者没有介绍的重要细节，包括基本概念、编译时标志（compile-time flag）、服务器配置以及转发。

支持的平台

本书介绍了在 Unix、Windows 和 Macintosh 上的 SSH 实现，类似的产品也可以用于 Amiga、BeOS、Java、OS/2、Palm Pilot、VMS 和 Windows CE。尽管我们在本书中没有介绍这些内容，但是其原理都是相同的。

本书目前适用于以下的 Unix SSH 版本。

SSH1	1.2.30
F-Secure SSH1	1.3.7
OpenSSH	2.2.0
SSH Secure Shell (又称 SSH2)	2.3.0
F-Secure SSH2	2.0.13

Unix 上的 F-Secure 与 SSH1 和 SSH2 几乎没有区别，所以除了其独特的特性之外，我们不会再单独介绍 F-Secure。附录二对它们之间的区别进行了总结。

非 Unix 产品的版本信息在其各自章节中介绍。

声明

我们把某些程序特性称为“非正式的（undocumented）”。也就是说这些特性没有在任何正式文档中介绍，但是在当前发行版本中可以工作，或者是从程序源代码中能

明显看出的。软件作者可能现在没有正式支持这些特性，在今后的发行版本中这些非正式特性也可能会消失。

排版约定

本书使用下列排版约定：

等宽字体 (*Constant width*)

配置文件中的内容（例如，关键字和配置文件选项）、源代码和交互式终端会话内容使用等宽字体。

等宽斜体 (*Constant width*)

命令行和配置文件中的可替换参数使用等宽斜体。

斜体 (*italic*)

文件名、URL、主机名、命令名、命令行选项和术语第一次定义时使用斜体。

A_K

在图中，标有 A 的对象是使用标有 K 的密钥进行安全处理过的。“安全处理”的含义根据上下文而不同，可能是表示加密、签名，或者一些更复杂的技术。如果对象 A 是使用多个密钥（比方 K 和 L）进行安全处理的，那么就会以下标方式给出，中间使用逗号分隔开： $A_{K,L}$ 。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly & Associates, Inc.

101 Morris Street

Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

关于本书的网页列出了勘误表、样例和额外信息，网址为：

<http://www.oreilly.com/catalog/sshtdg/>

询问技术问题或对本书的评论，请发电子邮件到：

info@mail.oreilly.com.cn

最后，您可以在 WWW 上找到我们：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

首先非常感谢 O'Reilly & Associates 公司给我们编写本书的机会，特别感谢我们的编辑 Mike Loukides，他允许我们延长日程表，从而能够深入介绍高级主题。我们要感谢 Frank Willison 对我们的信任，感谢 Christien Shangraw 卓越的管理能力以及首次排版时的气度，感谢 Mike Sierra 提供的工具和建议，感谢 Rob Romano 把我们的草图绘制成为优美的插图。

我们要感谢优秀的技术审校团队，他们通读本书并提出了富有洞察力的注释和评论，他们是：Anne Carasik、Markus Friedl、Joseph Galbraith、Sergey Okhapkin、Jari Ollikka、Niels Provos、Theo de Raadt、Jim Sheaffer、Drew Simonis、Mike Smith 和 Dug Song。

非常感谢 SSH 产品的提供商和开发人员，他们为我们提供了免费的软件拷贝，并回答了我们很多问题，他们是：Tatu Ylönen、Anne Carasik 和 Arlinda Sipilä (SSH Communication Security, Ltd.); Sami Sumkin、Heikki Nousiainen、Petri Nyman、Hannu Eloranta 和 Alexander Sayer (F-Secure Corporation); Dan Rask (Van Dyke

Technologies, Inc.); Gordon Chaffee (Windows SSH port); Ian Goldberg (Top Gun SSH); Douglas Mak (FiSSH); Jonas Walldén (NiftyTelnet SSH); 还有 Stephen Pendleton (sshCE)。SSH Communication Security, Ltd. 还授权给我们在本书中使用 *sshregex* 手册页 (附录一) 和 *sshdebug.h* 错误代码 (表 5-6)。

我们还要感谢 Rob Eigenbaum、James Mathiesen 和 J.D. Paul 在本书成文过程中所提供的技巧和灵感; 感谢 Chuck Bogorad、Ben Gould、David Primmer 和 Brandon Zehm 所提供的有关 NT 上的 SSH 的主页。Richard Silverman 非常感谢前面提到的公司的同事, 特别是 Michelle Madelien, 他在本书的写作期间对自己的古怪行为表现出极大的理解, 以及对自己飘忽不定的时间安排的配合。Richard Silverman 还要感谢 Deborah Kaplan 在 LART 应用程序中的英明决断和灵感。最后, 我们非常感谢 Usenet 上 *comp.security.ssh* 的众多参与者, 他们提出的很多问题提高了本书的质量, 尤其是第十二章。

第一章

SSH 简介

本章内容：

- SSH是什么
- SSH不是什么
- SSH协议
- SSH特性概述
- SSH的历史
- 相关技术
- 小结

现在很多人都有多个计算机账号。如果是一个明智的用户，就可能在 Internet 服务提供商 (ISP) 那里有一个个人账号，在公司局域网内有一个工作账号，在家里有一台或多台 PC。你可能还会使用家人或朋友的其他账号。

如果你有多个账号，就自然会想在这些账号之间实现连接。例如，你可能想要通过网络在两台计算机之间拷贝文件，从一个账号远程登录到另外一个账号，或者向远程计算机发送命令并在远程计算机上执行。很多程序都是出于这种目的而设计的，例如用于文件传输的 *ftp* 和 *rcp*，用于远程登录的 *telnet* 和 *rlogin*，还有用于远程执行命令的 *rsh*。

不幸的是，很多这种有关网络的程序都有一个基本问题：缺少安全性。如果通过 Internet 传输一个敏感文件，入侵者就有可能将该文件截获并能看到其中的数据。更糟糕的是，如果使用诸如 *telnet* 之类的程序远程登录到另外一台计算机上，那么用户名和密码在网络上传输时都可能被截获。真可怕！

怎样才能预防这些严重的问题呢？我们可以使用加密程序把自己的数据加密成一段其他人都无法读懂的密码。可以安装防火墙，防火墙是可以建立一道屏障、把入侵者阻隔在计算机网络之外的设备。或者可以使用其他更广泛的解决方案，既可以单独使用一个方案，也可以结合使用几个方案，但这样做的复杂度和费用都不同。

1.1 SSH 是什么

SSH (Secure Shell, 安全 Shell) 是一种通用的、功能强大的、基于软件的网络安全解决方案(注1)。计算机每次向网络发送数据时，SSH都会自动对其进行加密。当数据到达目的地时，SSH自动对加密数据进行解密(译码)。结果是整个加密过程都是透明的：用户可以正常工作，根本觉察不到他们的通信在网络上是经过安全加密的。另外，SSH使用了现代的安全加密算法，足以胜任大型公司的任务繁重的应用程序的要求。

SSH具有客户端/服务器的体系结构，如图 1-1 所示。SSH 服务器程序通常都是由系统管理员安装并运行的，它可以接受或拒绝到达自己主机的连接。然后用户运行 SSH 客户端程序(这通常是在其他计算机上执行的)来对 SSH 服务器发出请求，例如“请让我登录”、“请发送给我一个文件”或“请执行这个命令”。客户端和服务器之间的所有通信都经过安全加密，确保不会被修改。

到目前为止，虽然我们介绍得十分简单，但是读者对 SSH 究竟做些什么应该有了一些基本的了解。稍后我们会深入介绍。现在，只要记住 SSH 客户端和 SSH 服务器是通过加密的网络连接进行通信的就可以了。

基于 SSH 的产品可能包含客户端或服务器，也可能全都包含。Unix 产品通常都包含有客户端和服务器；其他平台上的产品通常都只包含客户端，但是现在基于 Windows 的服务器很快就会出现。

如果是 Unix 用户，可以认为 SSH 是 Unix 中 r- 命令——*rsh* (远程 Shell)，*rlogin* (远程登录)，*rcp* (远程拷贝) ——的一种安全形式。事实上，Unix 平台上最初的 SSH 就包含了一些名字类似的安全命令 *ssh*、*scp* 和 *slogin*，用它们来替代这些 r- 命令。是的，现在我们可以丢弃这些不安全的 *.rhosts* 和 *hosts.equiv* 文件了！(如果你喜欢，SSH 也可以和这些文件共同工作。) 如果你仍在使用 r- 命令，就直接转换到 SSH 上来吧：需要学习的内容很少，但是安全性却好得多。

注1：“SSH”的发音为 S-S-H。读者可能会觉得“安全 Shell”这个名字很奇怪，因为它实际上根本就不是一种 shell。这个名字来源于 *rsh* 工具，该工具是一个非常通用的 Unix 程序，也可以提供远程登录的功能，但是很不安全。

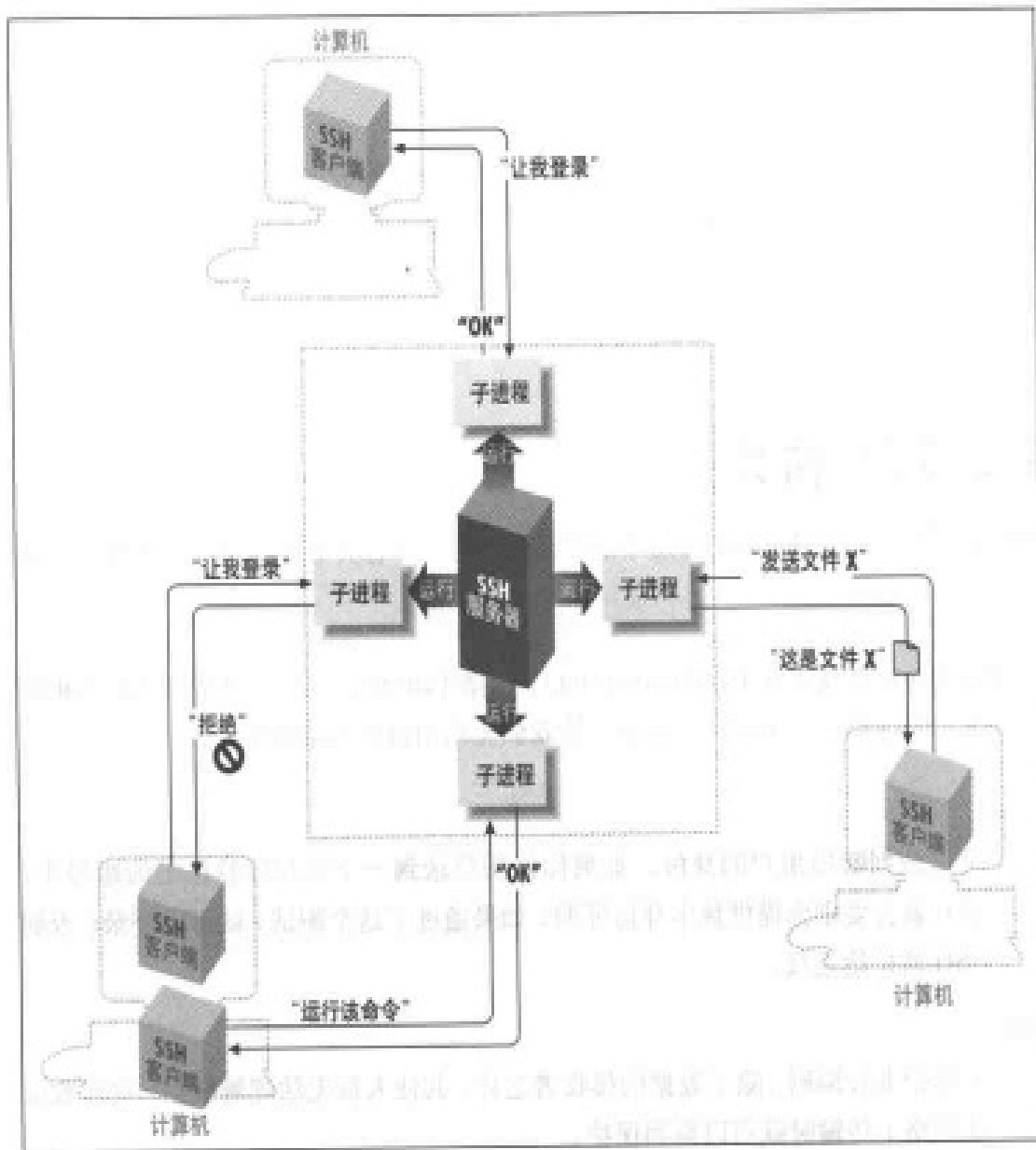


图 1-1：SSH 的体系结构

1.2 SSH 不是什么

虽然 SSH 代表的是安全 Shell，但它并不像 Unix 的 Bourne shell 和 C shell 一样是一种真正意义上的 shell。SSH 并不是一个命令解释器，也没有提供通配符扩展、命令历史记录之类的功能。实际上，SSH 创建了一个通道来在远程计算机上运行 shell，

其风格和 Unix 的 *rsh* 命令类似，不同的是在本地计算机和远程计算机之间使用了端到端的加密。

SSH 也不是一种完整的安全解决方案——但是到目前为止，也还没有什么完整的安全解决方案。SSH 并不能防止计算机的主动入侵或服务拒绝攻击，也不能防止出现病毒、Trojan 木马和咖啡豆（coffee spill）之类的危害。但是，SSH 提供了一种健壮的、用户界面友好的加密和认证措施。

1.3 SSH 协议

SSH 是一种协议（protocol），而不是产品。它是一种有关如何在网络上构建安全通信的规范（注 2）。

SSH 协议内容涉及认证（authentication）、加密（encryption）和网络上传输数据的完整性（integrity），如图 1-2 所示。让我们先给出这些术语的定义：

认证

可信地判断出用户的身份。如果你试图登录到一个远程计算机上的账号中，SSH 就会要求你提供数字身份证明。如果通过了这个测试，就可以登录；否则 SSH 就拒绝连接。

加密

对数据进行编码，除了数据的接收者之外，其他人都无法理解数据。这样数据在网络上传输时就可以得到保护。

完整性

确保网络上传输的数据到达目的地时没有被改变。如果某个第三方截获了传输的数据并对其进行了修改，SSH 就能检测到这种变化。

简而言之，SSH 在计算机之间建立网络连接，并能充分保障连接的双方是真实可信的。SSH 还能确保使用该连接传输的所有数据都不会被窃听者读取或修改。

注 2： 虽然我们说“SSH 协议”，但实际上该协议通常使用的有两个不兼容的版本：SSH-1（也就是 SSH-1.5）和 SSH-2。稍后我们就会区分这两种协议。

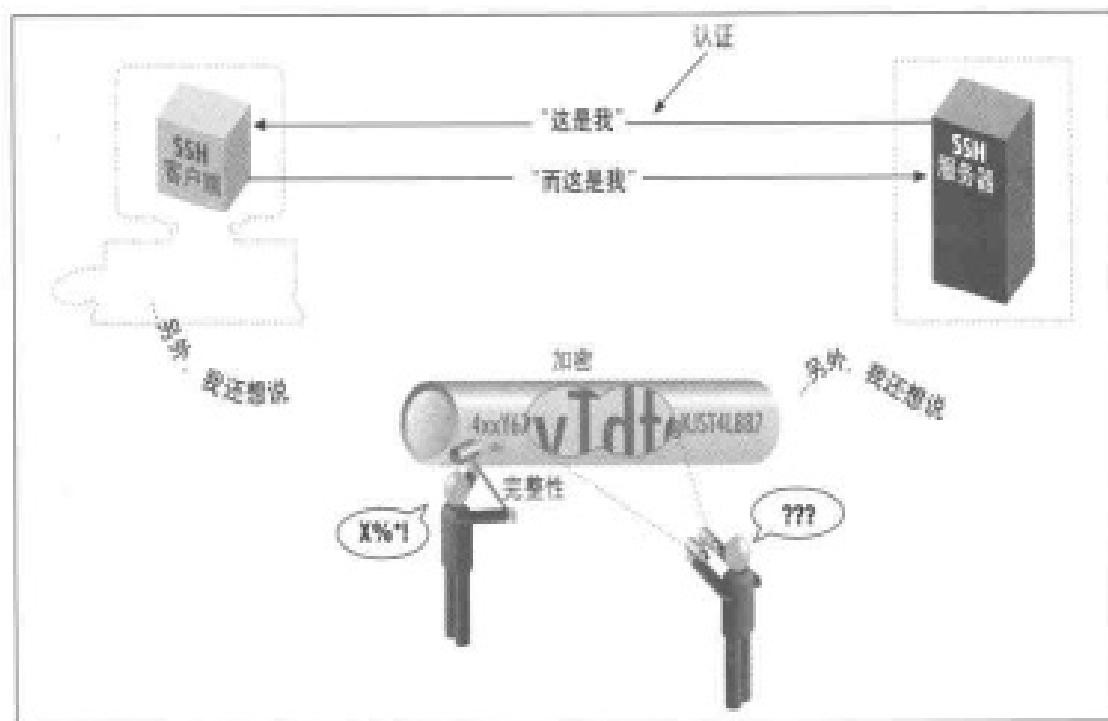


图 1-2：认证、加密和完整性

1.3.1 协议、产品、客户端和一些易混淆名称

基于SSH的产品（即实现SSH协议的产品）广泛存在于Unix、Windows、Macintosh和其他操作系统上，既有免费发布的产品，也有商业产品。[13.3]

SSH的第一个产品是由Tatu Ylönen在Unix平台上开发的，该产品的名字就叫“SSH”。这个名字可能会产生一些混淆，因为SSH本身也是协议名。虽然有人把Tatu Ylönen的软件称为“Unix SSH”，但是现在有很多其他基于Unix实现的产品，因此这个名字也不尽如人意。在本书中，我们使用更精确的术语来代表协议、产品和程序，我们在“术语：SSH协议和产品”中进行了总结。简而言之：

- 协议使用连词符表示：SSH-1、SSH-2。
- 产品使用大写字母表示，其中不含连词符：SSH1、SSH2。
- 客户端程序使用小写字母表示：ssh、ssh1、ssh2等。

术语：SSH 协议和产品

SSH

通用术语，泛指 SSH 协议或 SSH 软件产品。

SSH-1

SSH 协议版本 1。该协议历经几次修订，其中最广为人知的是 1.3 和 1.5 版本，为了加以区别，我们分别将其记为 *SSH-1.3* 和 *SSH-1.5*。

SSH-2

SSH 协议版本 2，由 IETF SECSH 工作组的几个草案标准文档定义。[\[3.5.1\]](#)

SSH1

Tatu Ylönen 实现 SSH-1 协议的软件；也是最早的 SSH 软件。现在由 SSH Communications Security Inc. 发行并维护（现在维护工作已经非常少了）。

SSH2

SSH Communications Security Inc. 的“SSH 安全 Shell”产品 (<http://www.ssh.com>)。这是一个实现了 SSH-2 协议的商业产品，但在某些环境中其许可证是免费的。

ssh (全部小写)

运行安全终端会话和远程命令的客户端程序，在 SSH1、SSH2、OpenSSH、F-Secure SSH 以及其他一些产品都包含了这样一个客户端程序。在 SSH1 和 SSH2 中，分别称为 *ssh1* 和 *ssh2*。

OpenSSH

OpenSSH 是 OpenBSD 项目的产品（请参看 <http://www.openssh.com>），它同时实现了 SSH-1 协议和 SSH-2 协议。

OpenSSH/1

OpenSSH 在使用 SSH-1 协议时就称为 OpenSSH/1。

OpenSSH/2

OpenSSH 在使用 SSH-2 协议时就称为 OpenSSH/2。



术语：网络

本地计算机（本地主机、本地机器）

用户所登录到的计算机，通常运行 SSH 客户端。

远程计算机（远程主机、远程机器）

用户从本地计算机连接的第二台计算机。远程计算机通常都运行 SSH 服务器，可以使用 SSH 客户端连接上来。特殊情况下本地计算机和远程计算机可以是同一台机器。

本地用户

登录到本地计算机上的用户。

远程用户

登录到远程计算机上的用户。

服务器

SSH 服务器程序。

服务器主机

运行 SSH 服务器程序的计算机。在上下文能清晰地区分开 SSH 服务器程序和服务器主机时，我们有时会简称之为“服务器”。

客户端

SSH 客户端程序。

客户端主机

运行 SSH 客户端程序的计算机。和服务器的术语类似，在上下文能清晰地区分开 SSH 客户端程序和客户端主机时，我们有时也会简称之为“客户端”。

~ 或 \$HOME

用户在 Unix 机器上的主目录 (home directory)，通常是在文件路径中使用，例如 `~/filename`。大部分 shell 都把 `~` 当作用户主目录，但 Bourne shell 在这方面是一个例外。所有的 shell 都可以识别 `$HOME`。

1.4 SSH 特性概述

那么SSH可以完成什么功能呢？让我们通过几个例子来说明SSH的基本特性，例如远程登录、安全文件拷贝和安全调用远程命令。在本例中我们使用SSH1，但实际上OpenSSH、SSH2 和 F-Secure SSH 都一样可以使用。

1.4.1 安全远程登录

假设你在Internet上的几台计算机上有多个账号。通常情况下，你都是从家用PC连到ISP上，然后使用*telnet*程序登录到其他计算机上的账号。不幸的是，*telnet*把你的用户名和密码以明文形式在Internet上传输，而这里可能有不怀好意的第三方会将其截获（注3）。网络监听者还可以看到全部的*telnet*会话。

SSH可以完全避免这些问题。用户现在不用再运行不安全的*telnet*程序，可以运行SSH客户端程序*ssh*。要登录到远程计算机*host.example.com*的smith用户账号中，可以使用这个命令：

```
$ ssh -l smith host.example.com
```

客户端可以对用户进行认证，让用户使用一个加密连接连往远程计算机的SSH服务器上（也就是说用户名和密码在离开本地主机之前都是经过加密的）。然后SSH服务器就允许用户登录进来，整个登录会话在客户端和服务器之间传输时都是经过加密的。因为加密是透明的，所以用户根本就觉察不出*telnet*和这种类似于*telnet*的SSH客户端之间有什么区别。

1.4.2 安全文件传输

假设你在两台Internet主机上都有账号，分别为*me@firstaccount.com*和*metoo@secondaccount.com*，现在你想把一个文件从第一个账号传送到第二个账号上。但是这个文件包含商业机密，一定不能让其他人看到。传统的文件传输程序（例如，*ftp*、*rcp*或*email*）都不能提供一种安全的解决方案。当文件在网络上传输时，第三方总可以将其截获并读取其中的数据包。要防止出现这种问题，可以采取很多

注3： 在标准的Telnet中情况的确如此，但有些Telnet实现增加了一些安全特性。

措施，在`firstaccount.com`中使用PGP（Pretty Good Privacy）之类的程序对该文件进行加密，然后使用传统的方法把文件传输到`secondaccount.com`，并在此处解密文件。然而这个过程比较繁杂，而且对用户不是透明的。

如果使用了SSH，那么就只需使用一个安全拷贝命令就可以在计算机之间安全传输文件。假设该文件的文件名为`myfile`，那么在`firstaccount.com`上执行的命令就是：

```
$ scp myfile metoo@secondaccount.com;
```

在我们使用`scp`传输文件时，该文件在离开`firstaccount.com`时就被加密了，到达`secondaccount.com`时再自动解密。

1.4.3 安全执行远程命令

假设你是一个系统管理员，需要在多台计算机上运行相同的命令。例如想使用Unix命令`/usr/ucb/w`查看一下局域网中四台计算机（`grape`、`lemon`、`kiwi`和`melon`）上每个用户启动的进程。按照传统的方法，如果我们已经在远程计算机上正确配置好了`rsh`的后台守护进程`rshd`，那么就可以使用`rsh`：

```
#!/bin/sh  
for machine in grape lemon kiwi melon  
do  
    rsh $machine /usr/ucb/w  
done
```

表明本段程序是 shell 脚本
对要处理的四台计算机依次执行操作
调用“/usr/ucb/w”程序
打印所有正在运行的进程

虽然这种方法可以达到我们的目的，但却不安全。`/usr/ucb/w`的结果在网络上是明文传输的；如果用户认为这些信息是很敏感的，那么就不能冒这种风险，何况`rsh`的认证机制是相当不安全的，很容易就能被攻破。我们可以使用`ssh`命令来代替`rsh`，程序如下：

```
#!/bin/sh  
for machine in grape lemon kiwi melon  
do  
    ssh $machine /usr/ucb/w  
done
```

注意是“ssh”而不是“rsh”

后者的语法和前者几乎完全相同，我们看到的输出结果也相同，但实际上`ssh`及其结果在网络上传输时都是经过加密的，在连接到远程主机时要使用强认证技术。

1.4.4 密钥和代理

假设你在网络上多台计算机中都有账号。出于安全因素的考虑，你希望在这些账号中使用不同的密码；但是要记住如此多的密码实在是件困难的事情。其实密码本身就存在安全问题。你输入一个密码越频繁，就越可能在错误地点输入该密码。（你以前曾经把密码当成用户名输入过并让别人看到过么？哎呀！在很多系统中，这种错误会记录在系统日志文件中，而你的密码是以明文形式记录的。）只需一次身份认证，然后就不用再反复输入密码就可以安全访问所有的账号，这难道不是个好主意么？

SSH有很多种认证机制，其中最安全的机制是基于密钥（key）的，而不是基于密码的。我们会在第六章中详细讨论密钥，现在可以把密钥定义为可以唯一标识一个 SSH 用户的位序列。考虑到安全性，密钥应该是加密过的；只有在用户输入口令（passphrase）将其解密之后才能使用。

使用密钥，同时使用一个称为认证代理（authentication agent）的程序，SSH 就能让用户不用记住太多的密码，也不用重复输入密码就可以安全通过认证登录到计算机账号中。其工作流程如下：

1. 预先把公钥文件放到自己的远程计算机账号中（这个步骤只需要执行一次），从而允许 SSH 客户端（*sch*、*scp*）访问远程账号。
2. 在本地计算机中，调用 *ssh-agent* 程序，该程序是在后台运行的。
3. 在登录会话时选择需要的密钥。
4. 使用 *ssh-add* 程序把该密钥装载到代理中。这需要知道每个密钥的口令。

现在，你在本地计算机中运行了一个 *ssh-agent* 程序，并把密钥保存在内存中。到现在为止，你已经完成了所有的工作。以后再访问含有公钥文件的远程账号时就不需要再如此频繁地输入密码了。跟反复输入密码的日子告别吧！这些设置可以一直持续到退出本地机器或结束 *ssh-agent* 为止。

1.4.5 访问控制

假设用户想允许另外一个人使用自己的计算机账号，但是他只能用于特定的目的

(执行受限操作)。例如，在你出差时你会想让秘书使用你的账号替你阅读email，但不能执行其他操作。使用SSH，你不用把密码告诉秘书，也不用修改密码就可以授权秘书访问自己的账号，并限定他只能运行邮件程序。设置这种受限访问的操作并不需要系统管理员权限。(相关内容在第八章中重点介绍。)

1.4.6 端口转发

SSH可以增加其他基于TCP/IP的应用程序（例如，*telnet*、*ftp*和X Window系统）的安全性。有一种称为端口转发（port forwarding）或隧道（tunneling）的技术可以对TCP/IP连接进行重新路由，使其通过SSH连接传输，并且透明地进行端到端的加密。端口转发也可以把这些程序通过网络防火墙传输，否则其用处就有限了。

假设用户已经登录到一台离办公地点很远的机器中，现在想访问公司内部的新闻服务器*news.yoyodyne.com*。虽然Yoyodyne网络是连接在Internet上的，但是有一个网络防火墙把到达大部分端口的连接都过滤掉了，其中就包括119端口，也就是新闻端口。然而该防火墙允许到达的SSH连接通过，这是因为SSH协议十分安全，就连Yoyodyne网络的吹毛求疵的系统管理员也信任它。SSH可以在本地专用TCP端口（也就是3002端口）上建立一条隧道连往远程主机的新闻端口。下面这条命令在此时我们看来可能有些莫名其妙，但它的确就是这样使用的：

```
$ ssh -L 3002:localhost:119 news.yoyodyne.com
```

该命令的意思是说，“*ssh*，请建立一条安全连接，它从我的本地机器TCP端口3002连到*news.yoyodyne.com*的新闻端口（TCP端口119）”。因此，为了安全地阅读新闻，用户需要正确配置自己的新闻阅读程序，使其连接到本地机器的3002端口。*ssh*创建的安全隧道可以自动和*news.yoyodyne.com*的新闻服务器通信，通过该隧道传递的所有新闻信息都是经过加密的。[9.1]

1.5 SSH的历史

SSH1和SSH-1协议是由Tatu Ylönen在1995年开发的，Tatu Ylönen是芬兰Helsinki科技大学的一名研究人员。在1995年年初，他的学校的网络曾经受到密码窃听的攻击，此后他就开发了SSH1，当时只是想自己使用。但到了beta版本，SSH1就引起了人们的广泛关注，Tatu Ylönen也意识到自己的安全产品可以更广泛地应用。

1995年7月，SSH1以自由软件的形式伴随源代码一起发布，这允许人们无需任何费用就拷贝并使用SSH1。到1995年底为止，估计已经有50个国家的20000个用户使用了SSH1，Ylönen每天要处理150封要求技术支持的邮件。为了适应这种情况，Ylönen于1995年12月建立了SSH Communications Security, Ltd.（简称SCS，其主页为<http://www.ssh.com/>）对SSH进行维护，并使其商业化。现在他是该公司的总裁和首席技术官。

同样是在1995年，Ylönen将SSH-1协议编写成IETF的一份Internet草案，该草案从本质上说明了SSH1软件的操作。这是一个十分特别的协议，其中包括很多在SSH1逐渐开发使用过程中所发现的问题和不足。如果一定要保持向后兼容，这些问题就不可能解决，因此在1996年，SCS引入了该协议的一个新的主要版本：SSH 2.0或SSH-2；它采用了一种新的算法，不能和SSH-1兼容。与此相呼应，IETF也成立了一个称为SECSH（Secure Shell）的工作组，以对该协议进行标准化，并从公众兴趣出发来引导其开发。SECSH工作组于1997年2月提交了第一份SSH-2.0协议的Internet草案。

1998年，SCS发布了基于先进的SSH-2协议的软件产品“SSH安全Shell”（SSH2）。然而，SSH2并没有取代SSH1在该领域的广泛应用，这是由于两个原因。首先，SSH2没有保留SSH1那些好用的、经过实践验证的功能和配置选项。其次，SSH2的许可证限制更为严格了。最初的SSH1一直都可以从Ylönen和Helsinki科技大学那里免费获取。SCS公司开发的SSH1的新版本对于大部分用户来说仍然是免费的，只要该软件不直接销售获利也不用于向用户提供服务，即使在商业环境中它也是免费的。但是，SSH2则不同，它是一个商业产品，只允许具有资格的教育机构和非赢利性组织免费使用。结果是当SSH2第一次出现时，大部分SSH1用户并没有了解到SSH2的多少优点，他们依旧继续使用SSH1。到本书编写时为止，SSH-2协议已经发布了3年了，虽然SSH-2比SSH-1更好更安全，但是SSH-1协议依然是Internet上使用最广泛的版本。

但是这种情况随着两种开发趋势的发展肯定会改变，这就是SSH2版权许可证的放宽和免费的SSH-2实现产品的出现。在本书在2000年末要出版时，SCS放宽了SSH2许可证，允许为具有资格的非商业组织工作的个人免费使用。它还允许在Linux、NetBSD和OpenBSD操作系统上免费使用，使用这些操作系统不受限制，即使在商业环境中也可以。同时，OpenSSH（<http://www.openssh.com>）开辟了另外一条SSH实现的道路，OpenSSH是由OpenBSD项目（<http://www.openbsd.org>）资助开发并

遵循 OpenBSD 许可证免费使用的。OpenSSH 基于最初的 SSH 的最新发行版本 1.2.12，现在已经得到了迅速的发展。虽然有很多人都为 OpenSSH 的开发做过贡献，但是 OpenSSH 主要的工作都是由 Markus Friedl 完成的。OpenSSH 使用相同的程序同时支持 SSH-1 和 SSH-2，但是 SSH1 和 SSH2 具有不同的可执行程序，而且 SSH2 中和 SSH-1 兼容的功能需要两种产品都得安装才行。虽然 OpenSSH 是在 OpenBSD 上开发的，但是它已经被成功移植到 Linux、Solaris、AIX 和其他操作系统中，这些移植产品都与其主要发行版本紧密地保持同步。虽然 OpenSSH 相当年轻，而且没有 SSH1 和 SSH2 中的一些功能，但是它正在迅速发展，将来可能是最流行的 SSH 产品。

现在，SSH1 的开发已经停止了，惟一继续的是重要错误的修正；而 SSH2 和 OpenSSH 的开发依然继续着。还有其他很多 SSH 的实现产品，特别是 F-Secure Corporation 所维护和销售的 SSH1 和 SSH2 的一些商业版本，以及用于 PC、Macintosh、Palm Pilot 和其他操作系统的众多移植产品和原始产品。^[13.3]估计现在全世界有超过两百万 SSH 用户（其中包括几十万 SCS 产品的注册用户）。

注意：有时我们会使用“SSH1/SSH2 及其派生产品”这个术语，该术语指 SCS 的 SSH1 和 SSH2，F-Secure 的 SSH 服务器（版本 1 和 2）、OpenSSH 以及基于 Unix 或其他操作系统上的 SSH1 和 SSH2 代码的移植产品。该术语不包括其他 SSH 产品（SecureCRT、NiftyTelnet SSH、F-Secure 的 Windows 和 Macintosh 客户端等等）。

1.6 相关技术

虽然 SSH 非常流行，也十分方便，但是我们依然不能宣称 SSH 是所有网络的最终安全解决方案。认证、加密和网络安全问题在 SSH 之前很久就产生了，这些问题已经被合并进了许多系统中。现在让我们介绍一些典型的系统。

1.6.1 rsh 命令族（r- 命令）

Unix 程序 *rsh*、*rlogin* 和 *rcp*（合称为 r- 命令）是 SSH1 客户端 *ssh*、*slogin* 和 *scp* 的前身。其用户界面、功能与其对应的 SSH1 程序几乎是完全相同的，惟一不同的是 SSH1 的客户端是安全的。r- 命令正好相反，它并不对连接进行加密，其认证模型十分脆弱，很容易被攻克。

一个 r- 命令服务器存在两种安全机制：网络名字服务和“特权（privileged）”TCP 端口。在接收客户端连接之前，服务器首先获得源主机的网络地址并将其转换成主机名。这个主机名必须在服务器的配置文件（通常是 */etc/hosts.equiv*）中存在，这样服务器才能允许该主机访问。服务器还要检查源 TCP 端口号是否在 1 ~ 1023 的范围内，因为这些端口只能由 Unix 超级用户（或 uid 为 root 的用户）使用。如果连接可以通过这两种检测，服务器就确信自己正在和一个可信主机上的可信程序进行通信，可以允许客户端任意登录。

但是这两种安全检测都很容易被攻克。从网络地址到主机名的转换是由诸如 Sun 的网络信息服务（NIS）或 Internet 域名系统（DNS）完成的。大部分 NIS 和 DNS 服务的实现都有安全漏洞，这样就可能导致第三方欺骗服务器相信了不可信的主机。然后，远程用户就可以简单地使用相同的用户名登录到该服务器中的其他账号中。

同理，盲目地相信特权 TCP 端口也会引起严重的安全问题。在可信主机上获得 root 特权的入侵者可以运行一个专用 *rsh* 客户端，他可以以任何用户的身份登录到服务器主机中。总之，那些其用户具有系统管理权限的计算机都可能会产生这种问题，有些操作系统不支持多个用户或多种权限（例如，Windows 9x 和 Macintosh）。在这些环境中就不能信任这些端口了。

如果可信主机上的用户数据库和服务器总能保持同步，特权程序（*setuid* 为 root）的安装总能受到严格监控，root 特权只授给可信用户，并且物理网络受到保护，那么 r- 命令就是相当安全的。这些假设在网络的初期是可行的，那时机器数量很少、价格昂贵，只有一小部分人可以使用机器，系统管理员都是可以信赖的；但是这些假设到现在都不可能了。

由于 SSH 具有良好的安全特性，并且 SSH 可以向后兼容 *rsh* (*scp* 也可以向后兼容 *rcp*)，因此我们就没有理由再使用 r- 命令了。安装 SSH 是件快乐的事情。

1.6.2 PGP

PGP (Pretty Good Privacy) 是一个在多种计算机平台上都可以使用的通用加密程序，它是由 Phil Zimmerman 开发的。PGP 可以对用户进行认证并对数据文件和 email 消息进行加密。

SSH集成了一些和PGP类似的加密算法，但用法不同。PGP是基于文件的，通常都是某个时刻在一台计算机上加密一个文件或email消息。而SSH则不同，它在互连的计算机中对发出的会话进行加密。PGP和SSH之间的区别类似于批处理和交互式处理之间的区别。

注意：PGP 和 SSH 之间还有一点联系：SSH2 也可以使用 PGP 的密钥用于认证。[5.5.1.6]

有关 PGP 更多的信息可以参考 <http://www.pgpi.com/>。

1.6.3 Kerberos

Kerberos 是一种安全认证系统，用于网络可能被监视、而且计算机不是中心控制的环境。Kerberos 是作为 Athena 项目的一个部分开发出来的，Athena 项目是麻省理工学院（MIT）的一个研究项目。Kerberos 认证采用证书（ticket）的方式对用户进行认证，证书是一个小字节序列，具有有限的生命期，而用户密码在中心计算机中都是安全的。

Kerberos 和 SSH 解决的问题相似，但是范围有很大不同。SSH 是一种轻量级的解决方案，易于部署，其目标是在现行系统上不需多大修改就可以使用。由一台计算机对另外一台计算机进行一次安全访问，只需要在第一台计算机上安装一个客户端，在第二台计算机上安装一个服务器并把服务器启动就可以。Kerberos 则不同，在使用之前必须构建一些重要基础，例如系统管理员用户账号、重负载的中心控制主机以及网络范围的时钟同步所使用的软件。在增加了这些复杂内容之后，Kerberos 就可以确保用户密码尽可能少地在网络上传输，而是只存储在中心主机上。SSH 在用户每次登录时都要在网络上发送密码（当然密码是通过加密连接传输的），并将密钥存储在使用 SSH 的每台主机上。Kerberos 还可以用于 SSH 之外的目的，包括集中控制用户账号数据库、访问控制列表以及层次化认证模型。

SSH 和 Kerberos 之间的另外一个不同之处在于增加客户端程序安全性的方法。SSH 可以和那些在后台使用 *rsh* 的程序（例如 Pine，Pine 是通用的邮件阅读程序）方便地集成。[11.3] 对这些程序进行配置，使其使用 *ssh* 来代替 *rsh*，那么这些程序的远程连接就十分安全了。对于那些直接打开网络连接的程序来说，SSH 的端口转发

(port-forwarding) 功能则提供了另外一种方便的集成方式。与 SSH 不同，Kerberos 包含一套可编程的库来对其他应用程序增加认证和加密功能。开发者可以对自己的源代码进行修改，让其调用 Kerberos 库来实现与 Kerberos 的集成（注 4）。MIT Kerberos 发行版本中包含了一些已经与 Kerberos 集成过的通用服务，包括 *telnet*、*ftp* 和 *rsh*。

如果你觉得 Kerberos 和 SSH 的功能听起来都不错，那么十分幸运：这些功能已经集成在一起了。[11.4] 有关 Kerberos 的更多信息请参看：

<http://web.mit.edu/kerberos/www/>

<http://nii.isi.edu/info/kerberos/>

1.6.4 IPSEC

Internet 协议安全性 (Internet Protocol Security, IPSEC) 是解决网络安全性的一份逐步修订的 Internet 标准。IPSEC 由 IETF 工作组开发，包含在 IP 层实现的认证和加密。IP 层在网络层次中比 SSH 更低。它对于终端用户来说是完全透明的，用户不需使用 SSH 之类的特定程序来获得安全性；IPSEC 的存在可以确保网络通信受到底层系统的自动保护。IPSEC 可以从一台主机经由一个不可信网络安全地连接到远程网络中，也可以连接到整个网络，这就是“虚拟专用网络 (VPN)”的思想。

作为一种解决方案来讲，SSH 通常比 IPSEC 更快、更容易部署，因为 SSH 只是一个简单的应用程序，而如果两台主机还不符合条件，那么 IPSEC 在两台主机的操作系统上都需要增加内容，根据实际情况，可能还要在路由器之类的网络设备上增添内容。SSH 还提供了用户认证，而 IPSEC 只能处理单个主机。另外，IPSEC 是更基本的一种保护方法，可以处理 SSH 不能处理的一些内容。例如，在第十一章中，我们会详细讨论使用 SSH 保护 FTP 协议的一些难点。如果需要提高 FTP 之类现有的不安全协议的安全性，那么 SSH 就不能满足要求，此时 IPSEC 就是正确的选择了。

IPSEC 可以使用一种称为认证头 (Authentication Header, AH) 的方法单独提供

注 4：SSH2 也朝着这种模块化的方向发展，它把实现 SSH2 协议的程序组织成一些库，可以通过 API 进行访问。

认证，也可以使用一种称为封装安全负载（Encapsulated Security Payload，ESP）的协议提供认证和加密。有关 IPSEC 的详细内容请参看：

<http://www.ietf.org/ids.by.wg/ipsec.html>

1.6.5 安全远程密码

安全远程密码（Secure Remote Password，SRP）协议是 Stanford 大学开发的、是一种和 SSH 有很大不同的安全协议。SRP 是一种专用的认证协议；而 SSH 包括认证、加密、完整性、会话管理等内容，它是这些内容的一个综合体。SRP 本身并不是一个完整的安全解决方案，它只是安全系统的一个部分所使用的一种技术。

SRP 设计的目标是提高使用密码进行认证的模型的安全性，并保留了很多理想的实用优点。如果用户在旅行途中，特别是没有携带自己的计算机而是使用其他人的计算机，那么使用 SSH 公钥进行认证就会很困难。用户不得不随身带一张软盘存储自己的私钥，期望能在需要使用的机器上使用这个私钥。哎呀，不是已经有一个 X 终端了嘛，那就没问题了。

随身携带私钥并不是什么聪明的法子，因为有人可能会将其偷走，他们可以使用字典攻击（dictionary attack）进行解密来得到口令并恢复私钥。此时又回到密码的最初的问题上来了：密码为了可用必须短小易记，但是为了安全却必须足够长而且是随机的。

SRP 提供了严格的双方相互认证，客户端只需要记住一个很短的密码，这个密码不必具有很强的随机性。在传统的密码模式中，服务器要对那些必须进行保护的敏感数据库（例如密码及其加密过的版本，如，Unix 中的 /etc/passwd 和 /etc/shadow 文件）进行维护。这些数据必须足够隐秘，因为万一泄密之后就可能让攻击者冒充用户或通过字典攻击获得用户密码。SRP 的设计就防止了这种数据库可能泄漏的危险，而且由于它可以防止字典攻击，因此就可以降低密码的随机性（因此密码就更容易记忆而且更有用）。服务器仍然有很多十分敏感、需要保护的数据，但是这些数据泄漏的后果就没有这么严重了。

SRP 还有意设计为避免在操作中使用加密算法。这样就不会违反密码出口法，密码出口法禁止向外国出口特定的加密技术。

SRP 是一种有用的技术，我们希望它能引起更广泛的注意；它是 SSH 中其他认证方法的一个很好的替代品。目前 SRP 的实现包括用于 Telnet 和 FTP 协议的安全客户端和服务器，可以用于 Unix 和 Windows 平台。有关 SRP 的更多信息请参看：

<http://srp.stanford.edu/>

1.6.6 安全套接字层协议

安全套接字层（Secure Socket Layer, SSL）协议是一种认证和加密技术，它可以使用 Berkeley 套接字风格的 API 为 TCP 客户端提供安全服务。SSL 最初是由 Netscape Communication Corporation 开发的，用来在 Web 客户端和服务器之间增加 HTTP 协议的安全性。这是 SSL 的主要用途，但是 SSL 的用途并不仅仅局限于 HTTP。它在 IETF 标准中的编号是 RFC-2246，名为“TLS”（Transfer Layer Security），表示传输层的安全性。

采用 SSL 的程序通常使用一个数字证书（digital certificate）来表明自己的身份，数字证书是一串密码数据。证书说明有一个可信的第三方已经对身份和密钥之间的绑定关系进行了验证。Web 浏览器在使用 SSL 进行连接时会自动检查 Web 服务器所提供的证书，从而确保服务器就是用户想要连往的那个服务器。此后，浏览器和 Web 服务器之间的传输就都是经过加密的了。

SSL 主要用于 Web 应用程序，但也可以对其他协议增加一层“隧道”（tunnel）。只有“可信的第三方”存在，SSL 才是安全的。这种第三方组织称为证书管理机构（certificate authority, CA）。如果一个公司想从 CA 获得证书，那么该公司必须向 CA 证明自己的身份，例如使用法律文件。只要证据充分，CA 就可以颁发证书。

有关这方面更多的信息请参看 OpenSSL 项目：

<http://www.openssl.org/>

1.6.7 使用 SSL 增强 Telnet 和 FTP 的安全性

很多基于 TCP 的通信程序都使用 SSL 进行了功能扩充，从而提供 SSH 的一些功能，其中包括 telnet（例如，SSLtelnet、SRA telnet、SSLTel、STel）和 ftp（SSLftp）。

虽然这些工具都十分有用，但是它们的功能都很单一，通常都是一些原来不是为安全通信而编写的程序的修订版本。而 SSH 实现则不同，它可以和各种用途的工具集更好地集成在一起，完全是为提高安全性而编写的。

1.6.8 stunnel

stunnel 是由波兰的 Micha Trojnara 编写的一个 SSL 工具，它为 Unix 环境中现有的基于 TCP 的服务（例如，POP 服务器、IMAP 服务器）增加了 SSL 保护，而不用修改这些服务器程序的源代码。它可以作为一个任意端口上的服务守护进程的封装程序（wrapper）从 *inetd* 中调用，也可以单独运行，从而为特定服务接收网络连接。*stunnel* 对那些经由 SSL 到达的连接进行认证；如果这些连接允许被接收，*stunnel* 就运行服务器程序，并在客户端和服务器程序之间建立一个使用 SSL 保护的会话。

stunnel 十分有用，因为有很多通用的程序都有一些选项可以使用 SSL 来运行某些客户端/服务器协议。例如，Netscape 和 IE 浏览器都可以通过 SSL 连接到 POP、IMAP 和 SMTP 服务器上。有关 *stunnel* 的更多信息，请参看：

<http://mike.daewoo.com.pl/computer/stunnel/>

1.6.9 防火墙

防火墙是用来防止特定的数据进入或离开一个网络的硬件设备或软件程序。例如，位于 Web 站点和 Internet 之间的防火墙可以只允许 HTTP 和 HTTPS 通信到达该站点。再如，防火墙可以屏蔽那些不是来自特定网络地址的 TCP/IP 数据包。

防火墙并不能替代 SSH 或其他认证和加密方法，但是它们可以解决相似的问题。这几种技术可以结合在一起使用。

1.7 小结

SSH 是对计算机网络通信进行保护的一种功能强大而又便捷有效的方法。通过安全认证和加密技术，SSH 可以支持安全远程登录、安全远程命令执行、安全文件传输、访问控制、TCP/IP 端口转发，以及其他一些重要功能。

第二章

SSH 客户端的 基本用法

本章内容：

- 实例
- 使用 ssh 建立远程终端会话
- 增加实例的复杂性
- 使用密钥进行认证
- SSH 代理
- 不用密码或口令进行连接
- 各种 SSH 客户端
- 小结

SSH 本身的思想很简单，但是有些地方却很复杂。本章的介绍可以让你快速入门。我们首先介绍一下 SSH 的一些最有用的功能：

- 经由安全连接登录到远程计算机中。
- 通过安全连接在计算机之间传送文件。

我们还会介绍如何使用密钥进行认证，密钥比普通的密码更安全。本章最后还会介绍 SSH 客户端程序的一些高级用法，比如使用多个密钥、客户端配置文件以及 TCP 端口转发。

在本章的例子中我们大都使用 SSH1 和 SSH2，偶尔也会使用 OpenSSH。如果各种产品的语法有所差异，我们会分别进行介绍。

2.1 实例

假设你在出差的时候想查看你在 ISP *shell.isp.com* 提供的一台 Unix 主机上的电子邮件。在附近一个大学你有一个朋友可以让你登录到他在 *local.university.edu* 上的 Unix 账号上，并由此远程登录到你的账号中。要进行远程登录可以使用 *telnet* 或 *rlogin* 程序，但是在前面我们已经看到，使用这些程序在机器间建立的连接是不安全的。

全的。(无疑有些捣乱的学生会截获密码，并将其公布在制作盗版软件和MP3的Web服务器上。) 幸运的是，你朋友的 Unix 主机和你的 ISP 上都已经安装了 SSH 的产品。

在本章所采用的这个实例中，我们使用美元符号 (\$) 来表示本地主机 *local.university.edu* 的 shell 提示符，并把 *shell.isp.com* 的 shell 提示符表示成 *shell.isp.com>*。

2.2 使用 ssh 建立远程终端会话

假设你在 *shell.isp.com* 上的用户名是“pat”。要从 *local.university.edu* 上你朋友的账号连接到你的远程账号中，要输入：

```
$ ssh -l pat shell.isp.com  
pat's password: *****  
Last login: Mon May 24 19:32:51 1999 from quondam.nefertiti.org  
You have new mail.  
shell.isp.com>
```

上面这些操作将产生如图 2-1 所示的结果。*ssh* 命令运行一个客户端，通过 Internet 连接到 *shell.isp.com* 的 SSH 服务器上，要求你登录到远程账号 *pat* 上（注 1）。你可以使用 *user@host* 的语法代替 *-l* 选项实现相同的功能：

```
$ ssh pat@shell.isp.com
```

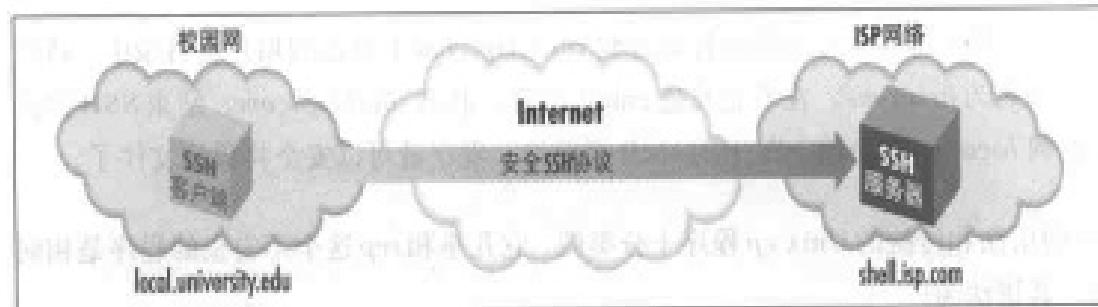


图 2-1：实例场景

在第一次进行通信时，SSH 就在客户端和服务端之间建立一条安全通道，此后二者之间的所有通信都是经过加密的。然后客户端就提示你输入密码，该密码就通过这

注 1：如果本地用户名和远程用户名相同，可以省略 *-l* 选项 (*-l pat*)，只输入 *ssh shell.isp.com* 即可。

一条安全通道提供给服务器。服务器通过对密码进行检查实现对身份的认证并允许登录。之后客户端和服务器之间交换的所有信息都由这条安全通道进行保护，包括前面使用邮件程序在 *shell.isp.com* 上读取的 email 的内容。

要注意这条安全通道只存在于 SSH 客户端和服务器之间。在使用 *ssh* 登录到 *shell.isp.com* 之后，如果又使用 *telnet* 或 *ftp* 登录到第三台主机 *insecure.isp.com* 上，那么 *shell.isp.com* 和 *insecure.isp.com* 之间的连接还是不安全的。但是，可以在 *shell.isp.com* 和 *insecure.isp.com* 之间也运行一个 *ssh* 客户端，再建立一条安全通道，这样就可以保证连接都是安全的。

到现在为止我们只介绍了 *ssh* 最简单的用法，在第七章中我们会更详细地介绍 *ssh* 的更多特性和选项。

2.2.1 使用 *scp* 传输文件

现在让我们继续介绍这个例子，假设在阅读电子邮件时看到一条消息，其中包含一个你想打印的附件。为了把这个文件发到该大学的本地打印机上，首先要把该文件传输到 *local.university.edu* 上。同理，也不能采用那些传统的不安全的文件传输程序，例如，*ftp* 和 *rcp*；而是使用另外一个 SSH 客户端程序 *scp* 把该文件通过一条安全通道从网络上拷贝过来。

首先，要使用邮件客户端把附件保存到 *shell.isp.com* 上自己的用户主目录中，将该文件命名为 *print-me*。在看完其他 email 之后，退出 *shell.isp.com*，结束 SSH 会话并返回 *local.university.edu* 上的 shell 提示符。现在就可以安全拷贝该文件了。

scp 的语法和传统的 Unix *cp* 程序十分类似，它几乎和 *rcp* 这个不安全的程序是相同的。其语法为：

```
scp name-of-source name-of-destination
```

在本例中，*scp* 把 *shell.isp.com* 上的文件 *print-me* 从网络上拷贝到 *local.university.edu* 上你朋友账号的本地文件中，新的文件名也是 *print-me*：

```
$ scp pat@shell.isp.com:print-me print-me
```

该文件是通过一个 SSH 安全化的连接进行传输的。我们不但可以使用文件名来指定

源文件和目的文件，而且可以使用用户名（本例中是“pat”）和主机名（*shell.isp.com*）来指定该文件在网络上的位置。根据用户需要的不同，我们可以省略源或目的的一部部分而使用缺省值。例如，如果省略了用户名和“at”符号（*pat@*），那就是默认远程用户名和本地用户名相同。

和 *ssh* 类似，*scp* 会提示输入远程密码并将其传送到 SSH 服务器上进行验证。如果验证成功，*scp* 就登录到 *shell.isp.com* 的 *pat* 账号中，把远程文件 *print-me* 拷贝成本地文件 *print-me*，然后退出 *shell.isp.com*。现在我们就可以把本地文件 *print-me* 发往打印机打印了。

目的文件名也可以和远程文件名不同。例如，如果懂法语，你也可以把本地文件命名为 *imprime-moi*：

```
$ scp pat@shell.isp.com:print-me imprime-moi
```

scp 的完整语法可以以各种方式来表示本地文件和远程文件，该程序本身也有很多命令行选项。[\[7.5\]](#)

2.3 增加实例的复杂性

前面的例子采用上机操作的方式向读者简要介绍了最常用的客户端程序——*ssh* 和 *scp*。现在读者对此已经有了一些基本的概念，下面让我们继续介绍这个实例，并给第一次的场景增加一些复杂性。这些复杂性包括“已知名主机（known host）”安全性和 SSH 转义字符。

注意：如果读者边看书边上机操作，那么就会看到自己的 SSH 客户端的情况可能和我们在本章中给出的不同。读者通读本书后，就会发现 SSH 实现可以进行广泛地定制，这可以由自己来完成，也可以由系统管理员完成，在安全连接的双方都是可以定制的。虽然本章是根据 SSH 的缺省安装来介绍 SSH 程序的一些通用特性，但是用户的设置可能与我们的不同。

如果输入的命令并不能像预期的那样执行，就试一下 *-v*（“verbose”）命令行选项，例如：

```
$ ssh -v shell.isp.com
```

该命令可以在客户端上显示很多 *ssh* 具体执行步骤的信息，通常能反映出这些差异的来源。

2.3.1 已知名主机

SSH客户端首次碰到一个新远程主机时，要执行一些额外的工作并显示和下面类似的消息：

```
$ ssh -l pat shell.isp.com  
Host key not found from the list of known hosts.  
Are you sure you want to continue connecting (yes/no)?
```

假设选择 yes（这是最常用的选择），那么客户端就会继续显示：

```
Host 'shell.isp.com' added to the list of known hosts.
```

这些消息只有在首次连接到一个特定的远程主机上时才显示，它是和SSH的已知名主机概念有关的一个安全特性。

假设有一个入侵者想要获得你的密码。他知道你正在使用SSH，就不能通过监听网络来监视你的连接，他可以修改你本地主机所使用的域名服务，从而把你想要连接到的远程主机*shell.isp.com*转换成一个由他控制的计算机的IP地址。然后他就可以在这台假冒的远程主机上安装一个SSH服务器，等待你登录上来。当你使用自己信任的SSH客户端登录时，这个假冒的SSH服务器就记录下你的密码供他以后使用（甚至会滥用）。然后这个假冒的服务器就显示一条诸如“*System down for maintenance-please try again after 4:00 p.m.*”之类的预先准备好的错误消息并断开连接。更有甚者，它可以把完全蒙在鼓里，使用你的密码登录到真实的*shell.isp.com*中，在你和服务器之间透明地往返传递信息，把你的所有会话都监听下来。这种入侵方法称为中间人攻击（man-in-the-middle attack）。[\[3.10.4\]](#)除非你想起检查服务器上自己会话的原始IP地址，否则你永远也发现不了这种诡计。

SSH已知名主机机制就可以防止这种攻击。当SSH客户端和服务器建立连接时，双方都要向对方证明自己的身份。是的，正如我们在服务器检查pat密码时就已经看到的一样，不但服务器要对客户端进行认证，客户端也要使用公钥对服务器进行认证。[\[3.4.1\]](#)简而言之，每个SSH服务器都有一个加密的惟一ID（称为主机密钥）来向客户端标识自己。在你第一次连接到一台远程主机上时，就将一个主机密钥的公用部分拷贝一份，并将其存储到自己的本地账号中（假设你在这之前对客户端有关主机密钥的提示信息回答的是“yes”）。之后你每次重新连接到这台远程主机上时，SSH客户端就要使用这个公钥来验证远程主机的身份。

当然，最好是能在首次连接到服务器之前就能获得这个服务器主机的公钥，否则在第一次连接服务器时你就可能受到中间人攻击。系统管理员可以在整个系统的范围内维护已知名主机列表，但是这对于要连接到整个世界上那么多主机上来说并不是什么好法子。因此在一种可靠的、广泛被采用的安全获得这种密钥的方法（例如，安全 DNS，或基于 X.509 的公钥结构）出现之前，这种首次使用时记录的机制是一种比较好的解决方案。

如果服务器认证失败，那么根据失败的原因和 SSH 配置的不同可能发生很多不同的事情。通常都会在屏幕上显示一条警告消息，提示你所连接的主机不在已知名主机数据库中：

```
Host key not found from the list of known hosts.  
Are you sure you want to continue connecting (yes/no)?
```

更糟糕的情况会显示如下信息：

```
@@@@@@@  
@     WARNING: HOST IDENTIFICATION HAS CHANGED!     @  
@@@@@@@  
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!  
Someone could be eavesdropping on you right now (man-in-the-middle attack)!  
It is also possible that the host key has just been changed.  
Please contact your system administrator.  
Add correct host key in <path>/known_hosts to get rid of this message.  
Agent forwarding is disabled to avoid attacks by corrupted servers.  
X11 forwarding is disabled to avoid attacks by corrupted servers.  
Are you sure you want to continue connecting (yes/no)?
```

如果回答 yes，ssh 就允许建立这个连接，但会禁用一些特性作为安全防护，而且不会使用这个新密钥更新你的个人已知名主机数据库；如果想以后不再显示这条消息，就得自己手工更新已知名主机数据库。

正如警告消息所述，如果你看到了这条警告消息，你现在也未必正被攻击：例如，远程主机由于某些原因而正常地修改了主机密钥。即使在通读本书之后，在某些情况下你也不会明白这些警告消息产生的原因。如果需要帮助，与其冒风险泄漏密码，还不如联系系统管理员。我们会在讨论个人的已知名主机数据库以及如何根据主机密钥修改 SSH 客户端行为时介绍这些问题。[\[7.4.3\]](#)

2.3.2 转义字符

让我们回到 *shell.isp.com* 的例子上来，刚才你已经在远程 email 消息中得到了附件，

并将其保存为远程主机上的文件 *print-me*。在我们最开始的例子中，你紧接着退出了 *shell.isp.com* 并运行 *scp* 来传输这个文件。但如果现在并不想退出怎么办呢？如果使用的是运行窗口系统的工作站，那么可以新开一个窗口来运行 *scp*。但如果使用的是一个简单文本终端，或你并不熟悉你朋友计算机上运行的窗口系统怎么办呢？这也有另外一种方法，你可以暂时中断 SSH 连接，先传输文件（并根据需要运行其他本地命令），然后再重新连接。

ssh 支持转义字符，转义字符可以通知 SSH 客户端：通常 *ssh* 把输入的每个字符都发送给服务器，但转义字符可以被客户端捕获，说明紧随其后的都是特殊命令。缺省情况下，转义字符是“~”，但也可以使用其他字符作为转义字符。为了不会无意地发送转义字符，转义字符必须是命令行的首字符，然后是一个换行（Control-J）或回车（Control-M）字符。否则，客户端会将其作为普通字符处理，而不是作为转义字符处理。

转义字符通知客户端要输入特殊命令之后，下一个输入的字符就可以确定转义字符的作用。例如，转义字符后面紧跟 Control-Z 会把 *ssh* 像其他 shell 任务一样挂起并返回本地 shell。这种字符串称为转义序列（escape sequence）。表 2-1 总结了 *ssh* 支持的转义序列，后面附有相应的含义。

表 2-1: *ssh* 转义序列

转义序列	例子: <ESC>=~	含义
<ESC> ^Z	~ ^Z	挂起连接 (^Z 表示 Control-Z)
<ESC> .	~ .	中断连接
<ESC> #	~ #	显示已有连接 ^a
<ESC> &	~ &	(在等待连接中止时) 把 <i>ssh</i> 转入后台执行 ^a
<ESC> r	~ r	请求立即重新生成密钥 (仅对 SSH2)
<ESC><ESC>	~ ~	发送转义字符 (输入两次)
<ESC> ?	~ ?	显示帮助消息
<ESC> -	~ -	禁用转义字符 (仅对 SSH2)
<ESC> V	~ V	显示版本信息 (仅对 SSH2)
<ESC> s	~ s	显示有关本会话的统计信息 (仅对 SSH2)

a. 对于 SSH2 来说，这个选项在 2.3.0 的文档中有说明，但是没有实现。

- “挂起连接”会把 *ssh* 转入后台，将其挂起，并将控制权交还给本地 shell 终端。要返回 *ssh*，就要使用 shell 中适当的任务控制命令，通常是 *fg*。在挂起期间，*ssh* 并不运行，如果挂起时间足够长，那么该连接就可能由于客户端不响应服务器而中断。另外，在 *ssh* 挂起期间，所有之前的连接都同样被阻塞。[\[9.2.9\]](#)
- “中断连接”会立即中断 SSH 会话。如果会话现在已经不受你控制了，这就非常有用：例如，远程主机上的 shell 命令已经挂起而不能删掉。任何 X 转发或 TCP 端口转发也同时被中断。[\[9.2.9\]](#)
- “显示已有连接”显示现在已经建立起来的所有 X 转发和 TCP 端口转发连接。这只会显示活动的转发实例；如果转发服务虽然仍然可以使用，但现在没有使用，那么就不会在这里显示。
- “把 *ssh* 转入后台执行”和“挂起连接”命令类似，都会重新返回开始执行 *ssh* 的 shell 终端，但是它不会挂起 *ssh* 进程。相反，*ssh* 会继续运行。通常这并没有什么用处，因为后台 *ssh* 进程会立即碰到一个错误（注 2）。当退出 *ssh* 时 *ssh* 会话还有活动连接时，这个转义字符序列就有用了。在这种情况下，客户端在等待退出之前的连接关闭时通常都会显示这样的消息：

```
Waiting for forwarded connections to terminate...
The following connections are open:
X11 connection from shell.isp.com port 1996
```

当客户端处于这种状态时，这个转义字符序列就能让你返回本地 shell 提示符。

- “请求立即重新生成密钥”会让 SSH2 客户端和服务器重新生成新密钥，并使用新密钥进行加密，确保完整性。
- “发送转义字符”告诉客户端把转义字符当作普通字符发给 SSH 服务器，而不将其作为转义字符进行处理。“禁用转义字符”可以禁用以后的转义字符。其他转义字符的意义都是显而易见的，就不再详细介绍。

要修改 *ssh* 转义字符，可以使用 *-e* 命令行选项。例如，在用户 pat 连往 *shell.isp.com* 时，要想将百分号（%）作为转义字符使用，可以使用下面的命令：

```
$ ssh -e "%" -l pat shell.isp.com
```

注 2：当 *ssh* 试图从这个现在断开连接的伪终端中读取输入信息时就会产生这个错误。

2.4 使用密钥进行认证

在我们的例子中，SSH服务器使用登录密码对用户 pat 进行认证。然而，密码有一些严重的缺陷：

- 为了确保密码安全，密码必须很长而且是随机的，但这种密码很难记忆。
- 如果远程主机已经被攻击，即使使用SSH安全通道进行保护，在网络上发送的密码在到达远程主机时也可能被截获。
- 大部分操作系统只支持一个账号使用一个密码。对于共享账号（例如，超级用户账号）来说，这就导致了一些困难：
 - 密码修改不便，因为新密码必须通知使用该账号的所有用户。
 - 跟踪账号变得困难，因为操作系统不能区分使用该账号的多个用户。

为解决这些问题，SSH 支持公钥认证：可以使用加密密钥，而不依赖于主机操作系统的密钥方案。[\[3.2.2\]](#)通常，密钥比密码更安全，并且解决了以前提到的问题。

2.4.1 密钥简介

密钥是一个数字证书，它是一个唯一的二进制字符串，用来声明“这就是我，真的是我，我发誓。”借助于加密的功劳，SSH客户端可以向服务器证明自己的密钥是真实无误的，你真的就是你。

SSH证书使用一对密钥：一个私钥和一个公钥。私钥（private key）只保存你独有的一些秘密信息。SSH客户端用其向服务器证明自己的身份。顾名思义，公钥（public key）是公开的，可以随便将其放入SSH服务器上自己的账号中。在认证时，SSH客户端和服务器要进行一次有关私钥和公钥的协商。如果（根据加密验证）私钥和公钥可以匹配，那么身份就得到了证明，认证就成功了。

以下的过程说明了客户端和服务器之间的协商过程。[\[3.4.1\]](#)（这些过程对用户来说都是透明的，因此不需要记住这些过程；我们只是认为你可能会对此感兴趣才详细介绍这些过程。）

1. 客户端说：“嗨！服务器，我想通过SSH连接到你系统中用户smith的账号上。”

2. 服务器说：“好吧，也许你可以。首先你得接受一个提问，向我证明你的身份！”
服务器就向客户端发送一些数据，称为验证提问。
3. 客户端说：“我接受你的提问。这是我的身份证明，是用我的私钥对你的提问进行数学计算之后生成的。”这种对服务器的响应称为认证者 (authenticator)。
4. 服务器说：“多谢你的认证。现在我要检查一下smith的账号来确定你能不能进入。”通常，服务器要对smith的公钥进行检查，来确认认证者是否与其“匹配”。（“匹配”是另外一次加密操作。）如果可以，服务器就说：“好了，进来吧！”否则，认证就失败了。

在可以使用公钥认证之前，首先要进行一些设置：

1. 需要一个公钥和一个私钥，合起来称为一个密钥对 (key pair)。还需要使用一个口令来保护自己的私钥。[2.4.2]
2. 需要在 SSH 服务器上安装自己的公钥。[2.4.3]

2.4.2 使用 ssh-keygen 生成密钥对

要使用加密认证，必须首先生成自己的密钥对，其中包含一个私钥（你的数字证书，存放在客户端机器上）和一个公钥（存放在服务器上）。要完成这个工作，就得使用 *ssh-keygen* 程序。*ssh-keygen* 在 SSH1、SSH2 和 OpenSSH 上的表现不同。在 SSH1 系统中，该程序名为 *ssh-keygen* 或 *ssh-keygen1*。调用该程序时，*ssh-keygen* 就创建一个 RSA 密钥对并让你输入口令来保护私钥（注 3）。

```
$ ssh-keygen1
Initializing random number generator...
Generating p: .....++ (distance 1368)
Generating q: ....++ (distance 58)
Computing the keys...
Testing the keys...
Key generation complete.
Enter file in which to save the key (/home/pat/.ssh/identity):
Enter passphrase: ****
Enter the same passphrase again: ****
Your identification has been saved in identity.
```

注 3： RSA 是 SSH 密钥及其他内容使用的一种加密算法。[3.9.1]DSA 是另外一种加密算法，稍后读者就会看到。

```
Your public key is:
1024 35 11272721957877936880509167858732970485872567486703821636830\
1950099934876023218886571857276011133767701853088352661186539160906\
921498698924021450762186406354890873029854678215446737245984456708\
9631066077107611074114663544313782992987840457273825436579285836220\
2493395730648451296601594344979290457421809236729 path@shell.isp.com
Your public key has been saved in identity.pub.
```

在SSH2系统中，该命令可能是`ssh-keygen`或`ssh-keygen2`，其表现有点不同，可以生成一个DSA密钥（缺省）或一个RSA密钥：

```
$ ssh-keygen2
Generating 1024-bit dsa key pair
1 ..000..000..00
2 0..000..000..00
3 0..000..000..00
4 0..000..000..00
Key generated.
1024-bit dsa, created by pat@shell.isp.com Mon Mar 20 13:01:15 2000
Passphrase : ****
Again : ****
Private key saved to /home/pat/.ssh2/id_dsa_1024_a
Public key saved to /home/pat/.ssh2/id_dsa_1024_a.pub
```

OpenSSH上的`ssh-keygen`也可以生成RSA密钥或DSA密钥，缺省的是RSA。其操作和`ssh-keygen1`类似。

`ssh-keygen`通常会执行所有需要的数学工具来生成密钥，但是在某些操作系统中，可能会需要用户指定。密钥的生成需要一些随机数。如果你用的操作系统没有随机数生成器，那么就会要求输入一些随机文本。`ssh-keygen`使用击键的时间来初始化内部随机数生成器。在运行Linux的主频为300MHz的Pentium系统中，生成一个1024位的RSA密钥大概需要3秒钟；如果硬件比这慢或者系统负载过重，那么所需要的时间可能会更长，甚至会长达1分钟以上。如果随机位不足，那么所需要的时间会更长，此时`ssh-keygen`就得等待更久。

然后，如果尚不存在SSH目录，`ssh-keygen`会创建本地SSH目录（对SSH1和OpenSSH来说是`~/.ssh`，对SSH2来说是`~/ssh2`），并将所生成的密钥的公有部分和私有部分分成两个文件存储在这儿。缺省情况下，这两个文件名为`identity`和`identity.pub`（SSH1、OpenSSH）或`id_dsa_1024_a`和`id_dsa_1024_a.pub`（SSH2）。SSH客户端把两个文件作为进行认证所使用的缺省证书。

警告：永远不要把自己的密钥和口令泄漏给任何人，这些数据和你的登录密码一样重要而且敏感。任何人有了这些数据之后都可以以你的身份登录系统。

*identity*文件在创建时只能使用你的账号读取，其内容使用创建过程中你所提供的口令进行了进一步的保护。我们使用“口令（passphrase，或称为密码短语）”而不都使用“密码（password）”是为了将其与登录密码区分开来，并强调在这中间可以使用空格和标点符号，而且也鼓励用户这样使用。我们建议口令至少要有10~15个字符，而且不要是一个合乎文法的句子。

*ssh-keygen*有很多选项用于管理密钥：修改口令、为密钥文件选择不同的文件名，等等。[\[6.2\]](#)

2.4.3 在 SSH 服务器上安装公钥

在使用密码进行认证时，主机操作系统要对用户名和密码之间的联系进行维护。如果使用密钥，用户必须手工建立一种类似的联系。在本地主机上创建密钥对之后，用户必须将自己的公钥安装到远程主机上自己的账号中。一个远程账号可以安装很多公钥用于各种方式的访问。

现在回到我们的例子上来，用户必须在 *shell.isp.com* 上安装一个公钥到“pat”账号中。这可以通过编辑 SSH 配置目录中的一个文件实现：对 SSH1 和 OpenSSH（[注 4](#)）来说该文件是 *~/.ssh/authorized_keys*，对 SSH2 来说该文件是 *~/.ssh2/authorization*。

对 SSH1 或 OpenSSH 来说，要创建或编辑文件 *~/.ssh/authorized_keys*，将自己的公钥（即用户在本地机器上生成的 *identity.pub* 文件的内容）加入其中。典型的 *authorized_keys* 文件包含一系列公钥数据，每个公钥一行。下面的例子中只有两个公钥，每个都在文件中占据一行，但是由于公钥太长，在本页中一行显示不下，所以打印设置程序在这一长串数字中加了换行符。如果在文件中也是这样，其格式就不正确，不能正常工作：

注 4： OpenSSH 中 SSH-2 连接使用 *authorized_keys2* 文件。为了简化本章内容，我们将在后文中讨论 OpenSSH。[\[8.2.3\]](#)

```

1024 35 8697511247987525784866526224505474204292260357215616159982327587956883143
36214702887649442651668267755021942582700217489030967220321970093718777979705864
107549106608811204142046600066790196940691100768682518506600601481676686828742807
11088849408310989234142475694298520575977312478025518391 my personal key
1024 37 1140868200916227508775331982659387253607752793422843620910258618820621996
941824516069319525136671585267698112659690736259150374130846896838697083490981532
877352706061107257845462743793679411866715467672826112629198483320167783914580965
674001731023872042965273839192998250061795483568436433123392629 my work key

```

这两个公钥是 RSA 公钥：每一项中的第一个数字是密钥的位数，而第二个和第三个数字是 RSA 特有的参数，称为公共指数（exponent）和模数（modulus）。之后是任意多个字符组成的一个字符串，也就是公钥的内容。[8.2.1]

对 SSH2 来说，用户需要编辑两个文件：一个在客户端；一个在服务器。在客户端上，要创建或编辑文件 `~/.ssh2/identification` 并在其中插入一行，说明自己的私钥文件名：

```
IdKey id_dsa_1024_a
```

在服务器上，要创建或编辑文件 `~/.ssh2/authorization`，该文件中包含有关公钥的信息，每行一个。但是和 SSH1 的 `authorized_keys` 文件不同（`authorized_keys` 文件中含有公钥的拷贝），`authorization` 文件中只给出公钥的文件名：

```
Key id_dsa_1024_a.pub
```

最后，把 `id_dsa_1024_a.pub` 文件从本地机器中拷贝到远程 SSH2 服务器上，将其保存在 `~/.ssh2` 中。

不管用户使用的是哪种 SSH 实现产品，都要确保远程 SSH 目录及其相关文件都只有用户的账号才具有写入权限（注 5）：

```

# SSH1, OpenSSH
$ chmod 755 ~/.ssh
$ chmod 644 ~/.ssh/authorized_keys

# 仅对 OpenSSH
$ chmod 644 ~/.ssh/authorized_keys2

# 仅对 SSH2
$ chmod 755 ~/.ssh2

```

注 5：为了防止 NFS 出现问题，我们把这个文件的属性设置成全部可以读取，这个目录的属性设置成全部可以搜索。[10.7.2]

```
$ chmod 644 ~/.ssh2/id_dsa_1024_n.pub
$ chmod 644 ~/.ssh2/authorization
```

SSH服务器对文件和目录权限的要求十分严格，如果远程用户的SSH配置文件的权限设置不恰当，那么SSH服务器就可能拒绝进行认证。[5.4.2.1]

现在可以使用新的密钥来访问“pat”账号了：

```
* ssh1, ssh2, OpenSSH: 输出的是SSH1 的情况
$ ssh -l pat shell.isp.com
Enter passphrase for RSA key 'Your Name <you@local.org>': ****
Last login: Mon May 24 19:44:21 1999 from quincunx.nefertiti.org
You have new mail.
shell.isp.com>
```

如果一切顺利，就可以登录到远程账号中。图 2-2 给出了整个过程。

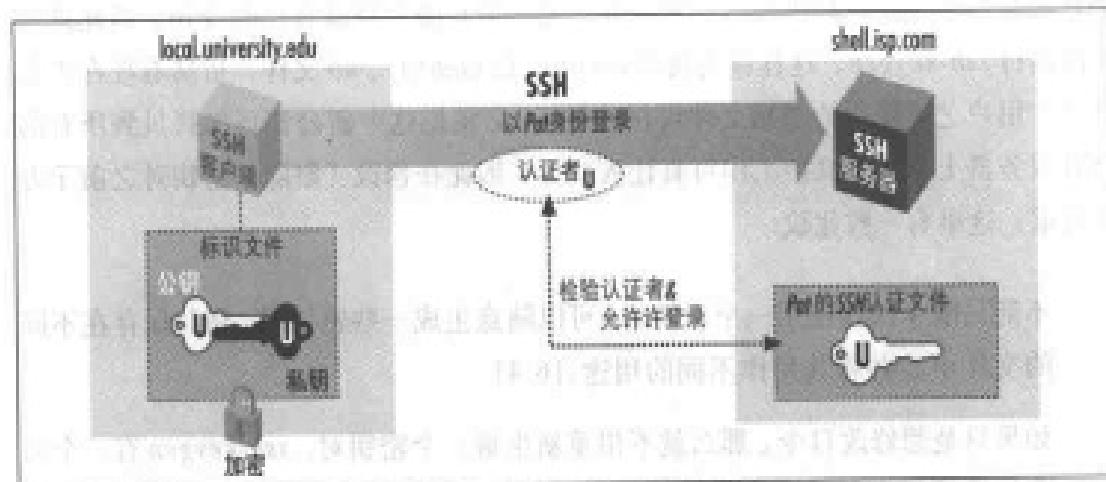


图 2-2：公钥认证

注意本例和前面使用密码进行认证的例子之间的相似之处。[2.2]从表面上来看，二者之间的唯一区别是用户为私钥提供的是口令，而不是登录密码。但是实际上二者之间的确有很大的不同。使用密码认证时，密码要传送到远程主机上。而使用密钥认证时，口令只用来解密私钥以创建认证者。[2.4.1]

公钥认证比密码认证更安全，这是因为：(1)公钥认证使用公钥加密，公钥是公开的；(2)公钥认证比密码认证更安全，这是因为：

- 公钥认证需要两个加密部分（磁盘上的identity文件和用户头脑中的口令），因此入侵者想要访问用户的账号就必须将二者全部截获。密码认证只需要一个部分，那就是密码，它可能更容易被窃取。

- 在公钥认证中，口令和密钥都不用发给远程主机，只要把前面讨论的认证者发给远程主机就可以了。因此，并没有什么秘密信息传出客户端。
- 机器生成的密钥是不可能猜测出来的；而人生成的密码使用一种称为字典攻击（dictionary attack）的密码猜测技术就可以按部就班地破解。字典攻击也可以对口令进行，但是这需要首先窃取私钥。

通过同时禁用密码认证并只允许使用密钥进行 SSH 连接可以极大提高主机的安全性。

2.4.4 如果要修改自己的密钥

假设已经生成了一个密钥对 *identity* 和 *identity.pub*，并把 *identity.pub* 拷贝到很多台 SSH 服务器中了。这都没什么问题。有一天，用户决定修改自己的身份，因此就要再次运行 *ssh-keygen*，这样就会覆盖 *identity* 和 *identity.pub* 文件。猜猜看现在怎么样了？用户之前使用的公钥文件现在无效了，必须把这个新公钥再次拷贝到所有的 SSH 服务器上。这种维护工作可真让人头疼，因此在修改（删除）密钥对之前千万要慎重。这里有一些建议：

- 不能局限于仅仅使用一个密钥对。可以随意生成一些密钥对，将其保存在不同的文件中，并将其用作不同的用途。[6.4]
- 如果只是想修改口令，那么就不用重新生成一个密钥对。*ssh-keygen* 有一个命令行选项可以替换现有密钥的口令：该选项对于 SSH1 和 OpenSSH 是 *-p*，[6.2.1]对于 SSH2 是 *-e*。[6.2.2]在这种情况下，因为私钥没有改变，所以公钥依然有效，只需要使用新口令对私钥进行解密就可以了。

2.5 SSH 代理

用户每次使用公钥认证运行 *ssh* 或 *scp* 时，都要重新输入口令。刚开始几次输入口令可能还没有什么，但最后这种反复输入口令的操作会让你觉得很讨厌。为什么不设计得更好点，只需要一次认证就可以让 *ssh* 和 *scp* 记住你的身份，一直到以后有事件发生时通知（例如，直到用户退出）为止都不提示再次输入口令呢？实际上，这正是 SSH 代理提供的功能。

代理是一个程序，它可以把私钥保存在内存中，并使用它来为SSH客户端提供认证服务。如果用户在开始一个登录会话时就使用一个代理，其中包含了很多私钥，那么SSH客户端就不会老提示输入口令；相反，它们会根据需要使用代理进行通信。代理的影响可以一直持续到用户结束代理为止，通常是在退出登录的前一瞬间。SSH1、SSH2 和 OpenSSH 使用的代理程序都称为 *ssh-agent*。

通常，用户都是在运行SSH客户端之前，先在自己的本地登录会话中运行 *ssh-agent*。虽然可以手工运行代理，但是人们通常都会编辑自己的登录文件（例如，*~/.login* 或 *~/.xsession*）来自动运行代理。SSH客户端和代理之间使用进程环境变量进行通信（注6），因此用户登录会话中的所有客户端（以及其他进程）都必须能访问该代理。要运行代理，请输入：

```
$ ssh-agent $SHELL
```

其中 *SHELL* 是包含用户的登录 shell 名的环境变量。另外，也可以指定其他 shell，例如，*sh*、*bash*、*csh*、*tcsh* 或 *ksh*。代理开始运行，然后调用给定的 shell，并将其作为一个子进程运行。直观效果只是出现了另外一个 shell 提示符，但是这个 shell 可以访问该代理。

代理一旦开始运行，就会使用 *ssh-add* 程序装入私钥。缺省情况下，*ssh-add* 会从缺省的标识文件中装入密钥：

```
$ ssh-add  
Need passphrase for /u/you/.ssh/identity ('Your Name <you@local.org>').  
Enter passphrase: *****  
Identity added: /u/you/.ssh/identity ('Your Name <you@local.org>').
```

现在不用 *ssh* 和 *scp* 一再提醒输入口令就可以连接到远程主机上了。图 2-3 说明了这个过程。

缺省情况下，*ssh-add* 从终端中读取口令，它也可以从标准输入中非交互式地读取密码。否则，如果用户正运行 X Window 系统，并设置了 DISPLAY 环境变量，而标准输入不是终端，那么 *ssh-add* 就使用一个图形化的 X 程序 *ssh-askpass* 来读取口令。在从 X 会话设置脚本中调用 *ssh-add* 时，这种功能就十分有用（注7）。

注 6： 在 Unix 中，SSH 客户端和代理之间使用命名管道进行通信，命名管道的文件名保存在一个环境变量中。[6.3.2]

注 7： 要强制 *ssh-add* 使用 X 来读取口令，请在命令行中输入 *ssh-add </dev/null*。

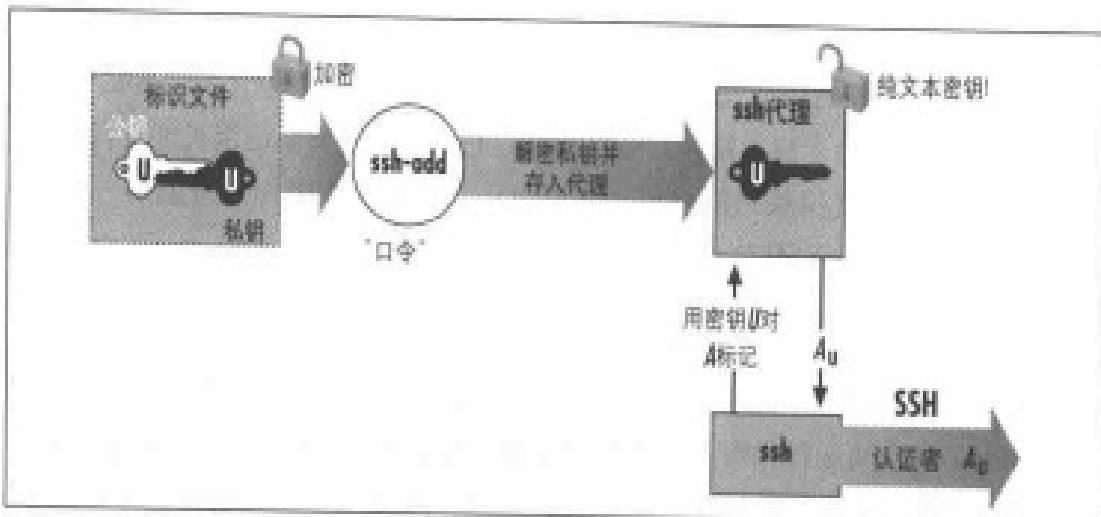


图 2-3: SSH 代理的工作方式

ssh-add 还有其他功能，特别是在 SSH2 中更是如此，它可以操作多个标识文件。[6.3.3] 现在我们给出几个有用的命令。要把一个非缺省的标识文件中的密钥装入代理中，可以给 *ssh-add* 提供一个文件名作为参数：

```
$ ssh-add my-other-key-file
```

可以显示代理现在持有的密钥：

```
$ ssh-add -l
```

从代理中删除一个密钥：

```
$ ssh-add -d name-of-key-file
```

删除代理中所有的密钥：

```
$ ssh-add -D
```

警告： 在运行 SSH 代理时，在登录期间不要放下终端不管。当用户的私钥被装载进代理之后，任何人都可以使用终端，而且不需要口令就可以通过这些私钥连接到任何可用的远程账号中！更糟糕的是，经验丰富的入侵者可以从正在运行的代理中提取出密钥并将其窃取。

如果用户使用代理，就要确保在登录期间离开终端时要将终端锁定。还可以使用 *ssh-add -D* 把已经装入代理的密钥清空。回来时再将其重新装入。另外，*ssh-agent2* 具有“锁定”特性，该特性可以防止未经认证的用户使用。[6.3.3]

2.5.1 代理的其他用途

由于 *ssh* 和 *rsh* 命令行的语法相似，用户自然想使用 *ssh* 来代替 *rsh*。假设有一个自动脚本，它使用 *rsh* 来运行远程进程。如果使用 *ssh* 替代 *rsh*，那么该脚本就会提示输入口令，这不便于实现自动操作。如果该脚本多次运行 *ssh*，那么反复输入口令既让人讨厌，还容易出错。但是如果运行一个代理，那么该脚本就可以不用输入口令而运行了。[\[11.1\]](#)

2.5.2 一个更复杂的口令问题

在前面的例子中，我们这样把一个文件从远程主机拷贝到本地主机：

```
$ scp pat@shell.isp.com:print-me imprime-moi
```

实际上，*scp* 可以把一个文件从远程主机 *shell.isp.com* 直接拷贝到运行 SSH 的第三方主机中，条件是在第三方主机上必须要有一个账号，如“psmith”：

```
$ scp pat@shell.isp.com:print-me psmith@other.host.net:imprime-moi
```

该命令不用把文件先拷贝到本地主机上，然后再拷贝到最终目的地，而是可以让 *shell.isp.com* 把该文件直接发给 *other.host.net*。然而，如果执行该命令，就会遇到下面的问题：

```
$ scp pat@shell.isp.com:print-me psmith@other.host.net:imprime-moi
Enter passphrase for RSA key 'Your Name <you@local.org>': *****
You have no controlling tty and no DISPLAY. Cannot read passphrase.
lost connection
```

这是怎么了？当用户在自己的本地机器上运行 *scp* 时，它连接到 *shell.isp.com* 上并间接调用另外一个 *scp* 命令执行拷贝工作。不幸的是，第二个 *scp* 命令也要求为密钥输入口令。由于远程主机上没有终端会话提示输入口令，第二个 *scp* 命令就会失败，从而导致最初的 *scp* 命令失败。SSH 代理可以解决这个问题：第二个 *scp* 命令只要简单地查询用户的本地 SSH 代理，这样就不需要提示输入口令了。

在本例中 SSH 代理还可以解决另外一个更棘手的问题。如果不使用代理，(*shell.isp.com* 上) 第二个 *scp* 需要访问用户的私钥文件，但是该文件在用户的本地机器上。因此用户只得将其拷贝到 *shell.isp.com*。这可并不是什么好主意；如果

`shell.isp.com` 不安全怎么办？还有，这种解决方案也没有什么伸缩性：如果用户有很多不同的账号，在所有的机器上维护这些私钥文件可是件头痛的事情。幸运的是，SSH 代理又充当了一次救世主。远程 `scp` 进程通过一个称为代理转发（agent forwarding）的进程简单地和本地 SSH 代理进行联系，认证和安全拷贝都可以连续成功执行。

2.5.3 代理转发

在前面的例子中，远程 `scp` 实例不能直接访问私钥，因为代理是在本地主机上运行的，而不是在远程主机上运行的。SSH 提供了代理转发机制[6.3.5]来解决这个问题。

当启用代理转发时（注 8），远程 SSH 服务器就会伪装成图 2-4 中的第二个 `ssh-agent`，它从 SSH 客户端进程中发出认证请求，将其通过 SSH 连接传回本地代理进行处理，并根据结果返回远程客户端。简而言之，远程客户端可以透明地访问本地 `ssh-agent`。由于通过远程服务器的 `ssh` 执行的任何程序都是该服务器的一个子进程，因此这些程序都可以访问本地代理，就仿佛这些程序都是在本地主机上运行的一样。

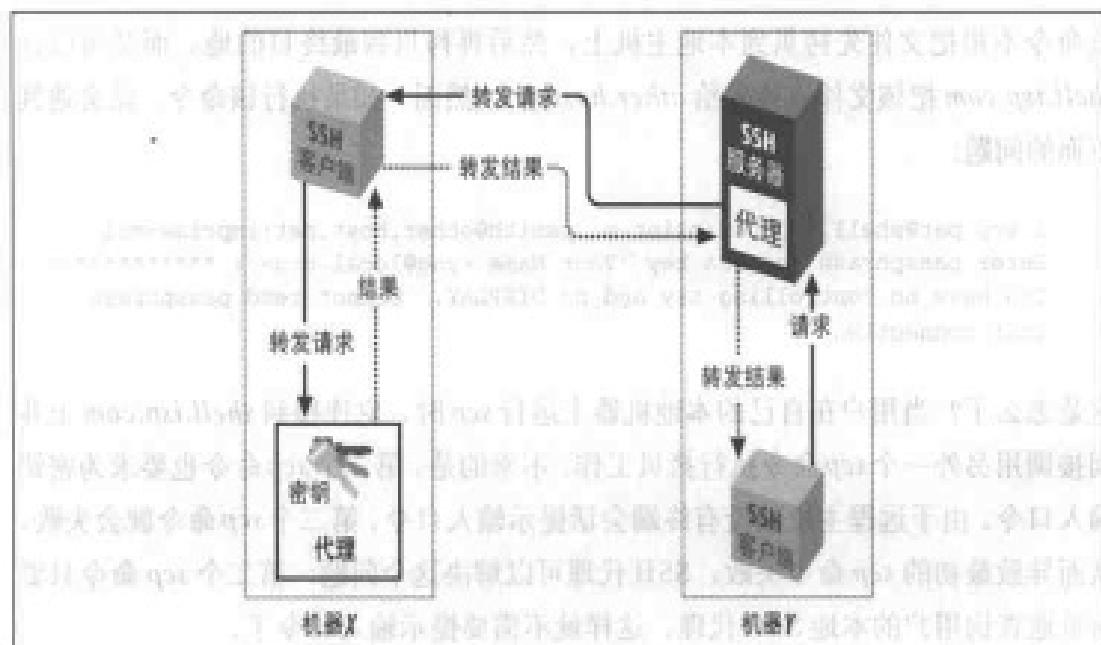


图 2-4：代理转发的工作方式

注 8：缺省情况下，代理转发在 SSH1 和 SSH2 中是启用的，而在 OpenSSH 中是禁用的。

在我们那个双远程 *scp* 例子中，当代理转发开始工作时会发生以下操作（请参看图 2-5）：

1. 在本地主机上运行命令：

```
$ scp pat@shell.isp.com:print-me psmith@other.host.net:imprime-moi
```

2. 这个 *scp* 进程与本地代理进行联系，并让用户和 *shell.isp.com* 进行认证。

3. 自动在 *shell.isp.com* 上执行第二个 *scp*，用来把文件拷贝到 *other.host.net* 上。

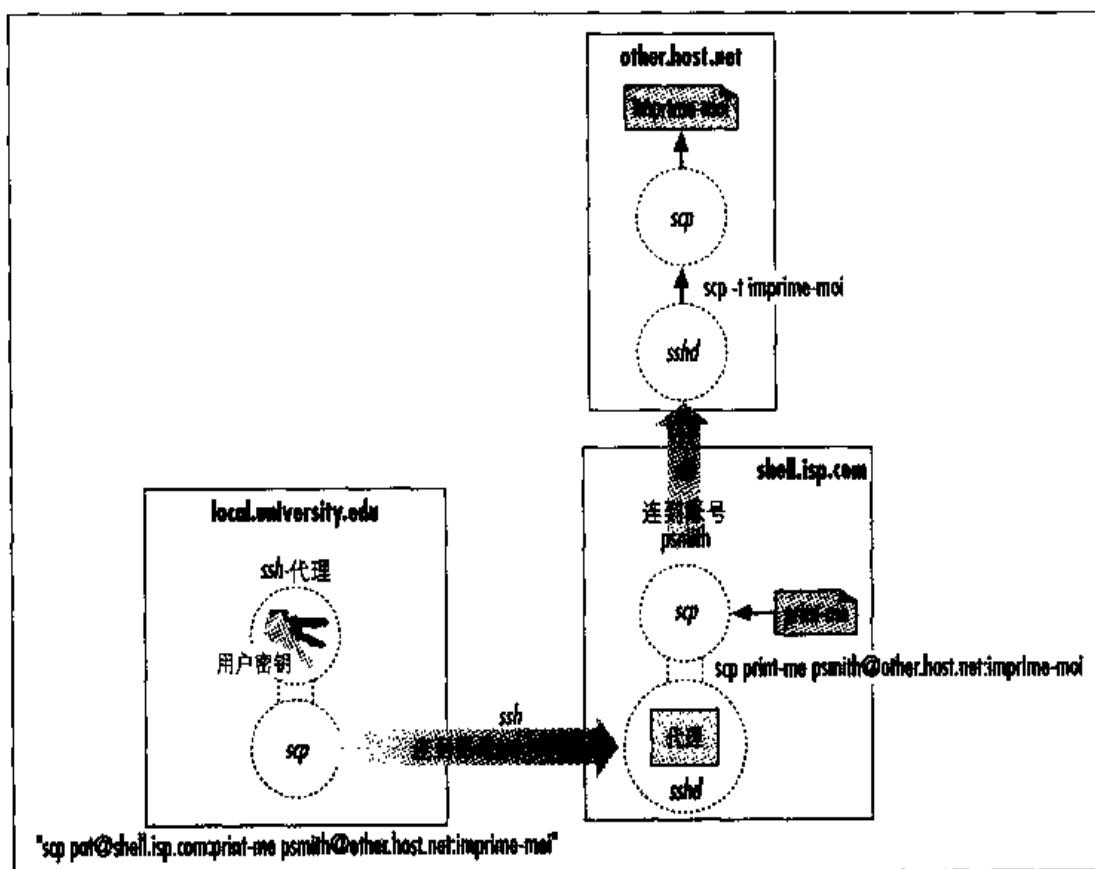


图 2-5：使用代理转发进行第三方 *scp* 操作

4. 由于启用了代理转发，因此 *shell.isp.com* 上的 SSH 服务器就充当一个代理。
5. 第二个 *scp* 进程通过联系 *shell.isp.com* 上的“代理”（实际上是 SSH 服务器），试图让用户和 *other.host.net* 进行认证。
6. *shell.isp.com* 上 SSH 服务器秘密与用户的本地代理进行通信，从而构建出一个认证者来提供用户的证书并将其传回服务器。

7. 服务器为第二个 *scp* 进程验证用户的身份，*other.host.net* 上的认证获得成功。
8. 开始拷贝文件。

代理转发通过一串多个连接进行工作，允许用户使用 *ssh* 从一台主机到另外一台主机，再到另外一台主机，整个过程都使用代理连接。这些主机的安全性可能会逐步减小，但是由于代理转发不会把私钥发送到远程主机上，而是把认证请求返回第一台主机上进行处理，因此密钥是安全的。

2.6 不用密码或口令进行连接

用户对 SSH 提出的最频繁的一个问题是：“我怎样才能不输入密码或口令就连接到远程主机上呢？”正如我们已经看到的一样，SSH 代理可以实现这个功能；还有其他一些方法也可以实现这个功能，每种方法都进行了一定的折衷。此处我们给出几种可用的方法，并说明我们会在哪一节对其进行讨论。

要想不输入密码或口令就使用 SSH 客户端进行交互会话，有几种选择：

- 使用代理的公钥认证。[\[2.5\]](#) [\[6.3\]](#)
- 可信主机认证。[\[3.4.2.3\]](#)
- Kerberos 认证。[\[11.4\]](#)

警告：另外一种实现无密码登录的方法是使用一个未使用口令加密过的私钥。虽然这种技术可以用于自动操作，但是千万不要使用这种方法进行交互操作。而要使用 SSH 代理，它提供了相同的便利，但是具有更好的安全性。不要使用未加密的密钥进行交互 SSH 操作！

另一方面，如果不用输入密码或口令，那么像 *cron* 任务或批处理脚本这种非交互式程序也可以方便地使用。在这种情况下，使用各种不同技术会使复杂程度提高，我们在后文中会讨论其优点和安全问题。[\[11.1\]](#)

2.7 各种客户端

除了 *ssh* 和 *scp* 之外，还有一些客户端：

- *sftp*, SSH2 的一个类 *ftp* 的客户端。
- *slogin*, 到 *ssh* 的一个链接, 类似于 *rlogin* 程序。
- 主机名链接 (到 *ssh*)。

2.7.1 sftp

虽然 *scp* 命令十分方便也非常有用, 但是很多用户早已非常熟悉 FTP (File Transfer Protocol, 文件传输协议), FTP 是 Internet 上广泛使用的一种文件传输技术 (注 9)。*sftp* 是在 SSH 之上的一个独立的文件传输工具, 它由 SSH Communications Security, Ltd. 开发, 最初只能在 SSH2 中使用, 但现在已经出现了其他一些实现版本 (例如, SecureFX 中的客户端、OpenSSH 中的客户端和服务器)。OpenSSH *sftp* 可以在 SSH-1 和 SSH-2 中运行, 而 SSH2 的 *sftp* 由于实现的细节问题只能在 SSH-2 上运行。

sftp 的优点来源于几个方面:

- 比较安全, 使用了一个受 SSH 保护的通道进行数据传输。
- 在一个 *sftp* 会话中可以调用多个命令进行文件拷贝和处理, 而 *scp* 每次调用时都要打开一个新会话。
- 可以使用熟悉的 *ftp* 命令编写脚本。
- 在后台运行 FTP 客户端的其他软件中, 可以使用 *sftp* 取代其他程序, 这样就可以增加该程序传输文件的安全性。

在测试 *sftp* 和其他类似的 FTP 工具时可能需要使用一个代理 (译注 1), 因为使用 FTP 的这些程序不能识别 *sftp* 提示用户输入口令的提示符, 或者它们都希望用户已经禁止显示 FTP 密码提示符 (例如, 使用 *.netrc* 文件)。

熟悉 FTP 的用户都会觉得使用 *sftp* 得心应手, 但是 *sftp* 还有其他一些值得注意的特性:

注 9: 根据 FTP 协议的特点, FTP 客户端很难使用 TCP 端口转发来增加安全性, 这一点和大部分基于 TCP 的客户端不同。[11.2]

译注 1: 此处的代理是指 *ssh-agent*, 而非 *ftp* 代理或其他传统意义上的代理。

- 使用类GNU Emacs风格的击键组合 (Control-B是退格键, Control-E是行尾, 等等) 进行命令行编辑。
- 文件名使用正则表达式进行匹配, 这在SSH2提供的`sshregex`手册页中有介绍, 请参看附录一。
- 几个命令行选项:

`-b filename`

从给定文件而不是从终端读取命令。

`-S path`

使用给定的路径定位`ssh2`程序。

`-h` 显示帮助消息并退出。

`-V` 显示该程序的版本号并退出。

`-D module=level`

打印调试输出。[5.8.2.2]

另外, `sftp`并不区分标准FTP的ASCII传输模式和二进制传输模式, 只使用二进制模式。所有文件都是逐字传输的。因此, 如果用户使用`sftp`在Windows和Unix之间拷贝ASCII文本文件, 那么就不能正确转换行结束符。通常, 标准FTP的ASCII模式会对Windows的“换行回车符”和Unix的换行符进行转换。

2.7.2 slogin

`slogin`是`ssh`的一个别名, 这就好像`rlogin`是`rsh`的一个别名一样。在Unix系统中, `slogin`只是连往`ssh`的一个符号链接。注意`slogin`链接只存在于SSH1和OpenSSH中, 在SSH2中则没有。为了保证一致性, 我们推荐只使用`ssh`, 它在所有的实现版本中都有, 而且短小、便于输入。

2.7.3 主机名链接

SSH1和OpenSSH的`ssh`对`rlogin`的模拟还有一个方面: 支持主机名链接。如果用户创建一个到`ssh`可执行文件的链接, 而使用的链接名不是`ssh`可以识别的标准名

(注 10)，那么 *ssh* 就会执行一些特殊的操作。它将链接名作为一个主机名处理，并试图连接到这个远程主机上。例如，如果用户创建了一个名为 *terpsichore.muses.org* 的链接，然后运行该链接：

```
$ ln -s /usr/local/bin/ssh terpsichore.muses.org
$ terpsichore.muses.org
Welcome to Terpsichore! Last login January 21st, 201 B.C.
terpsichore>
```

这和运行下面的命令效果是相同的：

```
$ ssh terpsichore.muses.org
Welcome to Terpsichore! Last login January 21st, 201 B.C.
terpsichore>
```

用户可以为所有常用的远程主机创建很多链接。注意在 SSH2 中已经取消了对主机名链接的支持。(我们自己并没有发现其用处，但是它的确在 SSH1 和 OpenSSH 中存在。)

2.8 小结

从用户的观点来看，SSH 包括几个客户端程序和一些配置文件。最常用的客户端是 *ssh* (用于远程登录) 和 *scp* (用于文件传输)。与远程主机之间的认证可以使用现有的登录密码或使用公钥加密技术实现。密码更直接更易用，而公钥认证更复杂更安全。*ssh-keygen*、*ssh-agent* 和 *ssh-add* 程序可以产生 SSH 密钥并对其进行管理。

注 10： *ssh* 可以识别的标准名有：*rsh*、*ssh*、*rlogin*、*slogin*、*ssh1*、*slogin1*、*ssh.old*、*slogin.old*、*ssh1.old*、*slogin1.old* 和 *remsh*。

第三章

SSH 内幕

本章内容：

- SSH 特性概述
- 密码学基础
- SSH 系统框架
- SSH-1 内幕
- SSH-2 内幕
- 伪装用户访问权限
- 随机数
- SSH 和文件传输 (scp 和 sftp)
- SSH 使用的算法
- SSH 可以防止的攻击
- SSH 不能防止的攻击
- 小结

SSH 可以使数据在网络上传输时得到保护，但它究竟是如何实现的呢？在本章中，我们将紧紧围绕技术方面的内容来介绍 SSH 的内幕。现在让我们卷起袖子大干一场，一点一滴地来挖掘 SSH 的内幕吧。

本章适用于系统管理员、网络管理员和安全专家。本章的目的是向读者介绍有关 SSH 的足够多的知识，从而使读者可以对 SSH 的使用作出理智的、技术合理的、英明的决定。由于 SSH-1 和 SSH-2 之间有很大的区别，因此我们将分别对其进行介绍。

当然，有关 SSH 的最权威的参考文献还是其协议标准和具体实现的源代码。我们不可能完整地分析协议，也不可能详细介绍软件执行的每个操作步骤。相反，我们将对这些内容进行总结，从而从技术上对其操作进行概要地介绍。如果需要了解更多细节内容，就应该参考那些标准的文档。在 IETF 标准中，SSH 版本 2 协议现在还是一个 IETF 草案；可以在这里找到这些内容：

<http://www.ipsec.com/tech/archive/secsh.html>

<http://www.ietf.org/>

SSH1 和 OpenSSH1 所实现的老版本协议是 V1.5，SSH1 源码包中有一个名为 RPC 的文件就包含了相应的文档。

3.1 SSH 特性概述

SSH 协议的主要特性和优点如下：

- 使用强加密 (strong encryption) 技术来保证数据的私密性 (privacy)。
- 通信的完整性，确保通信不会被修改。
- 认证 (authentication)，即发送者和接受者的身份证明。
- 授权 (authorization)，即对账号进行访问控制。
- 使用转发或隧道技术对其他基于 TCP/IP 的会话进行加密。

3.1.1 数据私密性（加密）

数据的私密性是指保护数据不会泄密。典型的计算机网络都不能保证数据私密性；任何可以访问网络硬件的用户或连接到网络上的主机都可以读取 (或监听) 网络上上传输的所有数据。虽然现在的交互式网络已经把这个问题限制在局域网范围之内，但这仍然是个十分严重的问题；使用这种监听攻击一般都可以窃听到密码。

SSH 通过对网络上上传输的数据进行加密，从而提供了私密性。这种端到端的加密的基础是随机密钥，随机密钥为会话进行安全协商，在会话结束后被丢弃。SSH 支持很多种对会话数据进行加密的加密算法，包括诸如 ARCFour、Blowfish、DES、IDEA 和 3DES 之类的标准加密算法。

3.1.2 完整性

完整性指数据从网络连接的一端传递到另外一端而不会被修改。SSH 的底层传输协议 TCP/IP 本身具有完整性检查，可以检测到网络问题 (如，电子噪声，由于网络负载过重而所造成的报文丢失，等等) 对数据造成的破坏。然而，这些方法对于蓄意攻击来说用处并不大，聪明的攻击者可以把这些保护措施玩弄于股掌之上。使用 SSH 对数据流进行加密，攻击者就不能轻易修改选定的内容达到特殊目的；即便如

此，也不可能只使用TCP/IP的完整性检查来防止攻击者蓄意向会话中添加各种垃圾信息。

更复杂的一个例子是重放攻击 (replay attack，或称为再现攻击)。假设攻击者 Attila 正在监视你的SSH会话，无异于他就在你的肩膀后面盯着你(可能他就站在你身后，也可能是在监视你终端的击键序列)。在你工作时，Attila 看到你在 一个目录中输入了命令 `rm -rf *`。虽然他无法读取加密过的SSH会话数据的内容，但是他可以监听你输入命令所使用的连接的动作，并截获包含加密过的命令的报文。然后，当你在自己的主目录中工作时，Attila 就可以在你的SSH会话中插入刚才截获的报文，最终你会错误地删除自己的所有文件！

Attila 的重放攻击之所以能够成功，是因为他插入的报文是有效的；虽然他不能自己生成这些报文(因为这些报文是加密过的)，但是他可以拷贝这段报文并在以后重新发送。TCP/IP的完整性检测只能以报文为单位进行，因此不能检测出 Attila 的攻击。显然，完整性检测必须把数据流作为一个整体进行，从而确保数据位到达目的地时和发送时一样：顺序相同而且无冗余。

SSH-2 协议使用密码完整性检测，目的有两个：确保所传输的数据未经改动，确保数据确实来自于连接的另一端。SSH-2 为此使用基于 MD5 和 SHA-1 的加密 hash 算法：这是一个很出名也很权威的算法。而 SSH-1 则不同，它使用的方法没这么可靠：它对每个报文中未加密的数据进行 32 位的循环冗余校验 (CRC-32)。[3.9.3]

3.1.3 认证

认证就是对用户身份进行验证。假设我宣称自己是 Richard Silverman，而你想对这个声明的真伪进行鉴别。如果没什么证据，你只好相信我的话。如果你有点头脑，就可以要求看我的驾驶执照或其他有照片的证件。如果你是银行的职员，要确定是否给我打开保险箱，就可以再要求我提供一把钥匙，等等。这都取决于你对我的身份进行确认的程度。高科技认证技术工业一直以来都不断发展，包括DNA测试微处理芯片、视网膜和指纹扫描仪以及声音打印分析仪。

每个 SSH 连接都涉及双向认证：不仅客户端要验证 SSH 服务器的身份（服务器认证），服务器也要验证发起访问请求的用户的身份（用户认证）。服务器认证确保SSH 服务器是真实无误的，而不是一个冒牌货，这样可以防止攻击者把你的网络连接重定向到另外一台机器。服务器认证还可以防止中间人攻击（man-in-the-middle

attack)，此时攻击者隐身于你和服务器之间，他在一端假装成客户端，而在另一端假装成服务器，从而欺骗双方并读取你整个会话过程中的所有通信！

有关服务器认证的粒度的大小问题存在不同的观点：其粒度大小是应该可以区分不同的服务器主机，还是应该可以区分 SSH 服务器上的单个实例呢？也就是说，在一台特定主机上运行的所有 SSH 服务器是都必须使用相同的主机密钥，还是可以使用不同的主机密钥呢？当然，术语“主机密钥”本身就反映了对于第一种观点的偏爱，SSH1 和 OpenSSH 正是采用这种方式：其已知名主机列表中只能给一个特定的主机名关联一个密钥。而 SSH2 则不同，它使用第二种方法：“主机密钥”实际上和一个监听套接字进行关联，从而允许每个主机可以有多个密钥。这可以反映出实际的需要，而不是只考虑理论上的变化。当 SSH2 刚出现时，只支持 DSA 主机密钥，而 SSH-1 只支持 RSA 密钥。因此，由于实现上的原因，一个主机不可能同时运行 SSH-1 和 SSH2 服务，也不能让 SSH-1 和 SSH-2 共享同一个主机密钥。

传统的用户认证是使用密码实现的，不幸的是这种认证模式十分脆弱。要证明你的身份，必须提供密码，这样就使密码可能被偷窃。另外，为了能记住密码，人们喜欢使用短小有意义的密码，这样就使第三方更容易猜测密码。对于长密码来说，有些人选择母语中的单词或句子，这些密码都很容易被破解。从信息论的观点来看，合乎文法的句子包含的信息量（技术上称为熵，*entropy*）少：通常少于英文文章中的每个字符两位的信息量，更远少于计算机编码中的每个字符 8~16 位的信息量。

SSH 支持密码认证，并且在网络上传输时会对密码进行加密。这对于其他通用的远程访问控制协议（Telnet, FTP）来说是个极大的改进，原来的这些工具在网络上传输时都是把密码以明文传送的（也就是未加密），因此具有足够网络访问权限的用户都可以窃取密码！然而，密码认证还是太过简单，因此 SSH 提供了功能更强大、更易管理的机制：每个用户使用一个公钥签名，使用改进过的 *rlogin* 风格的认证，并使用公钥对主机身份进行验证。另外，不同的 SSH 实现都支持其他一些系统，包括 Kerberos、RSA 安全公司的 SecurID 令牌、S/Key 一次性密码以及可插入认证模块（PAM）系统。SSH 客户端和服务器要根据其配置进行协商，并确定使用哪种认证机制。SSH2 甚至可以要求使用多种认证方法。

3.1.4 授权

授权就是确定哪些人有权操作，哪些人无权操作。授权是在认证之后进行的，因为在你知道用户是谁之前，无法确定用户的权限。SSH 服务器有很多种方法来限制客

客户端可以执行的操作。虽然对交互式登录会话、TCP 端口和 X 窗口转发、密钥代理转发等的访问都可以进行控制，但是并不是所有的 SSH 实现中都提供了这些功能，而且它们也并不像你想象的那样通用或灵活。授权可以在服务器范围的层次上（例如，对于 SSH1 来说是 */etc/sshd_config* 文件）进行控制，也可以在每账号的层次上（例如，每个用户的 *~/.ssh/authorized_keys*、*~/.ssh2/authorization*、*~/.shosts*、*~/.k5login* 等文件）进行控制，这要取决于所使用的认证方法。

3.1.5 转发（隧道）

转发或隧道是在一个 SSH 会话中封装另外一个基于 TCP 的服务，例如，Telnet 或 IMAP。这样就可以把 SSH 的安全特性（私密性、完整性、认证、授权）应用到其他基于 TCP 的服务上。例如，普通的 Telnet 连接传输用户名、密码和登录会话的其他内容都是以明文方式进行的。使用 SSH 对 *telnet* 进行转发，所有这些数据都可以自动加密并进行完整性检查，用户可以使用 SSH 证书对其进行鉴别。

SSH 支持三种转发。通常 TCP 端口转发可以用于前面提到的任何一种基于 TCP 的服务。^[9.2] X 转发包含有另外一些增强 X 协议（例如，X Windows）安全性的特性。^[9.3] 第三种是代理转发，它允许 SSH 客户端使用远程主机上的 SSH 私钥。^[6.3.5]

3.2 密码学基础

到现在我们已经介绍了 SSH 的基本特性。现在我们将注意力转移到密码学上来，从宏观上介绍一些重要的术语和理论。有关密码学和实际应用有很多很好的参考文献，我们在本书中不再详细地介绍。（要想了解更多细节内容，请参考 Bruce Schneier 的一本好书，《Applied Cryptography》，John Wiley & Sons 出版。）我们在本书中会介绍有关加密（*encryption*）、解密（*decryption*）、明文（*plaintext*）、密文（*ciphertext*）、密钥（*key*）、密钥（*secret-key*）加密、公钥（*public-key*）加密以及散列函数的内容，既要介绍基本知识，还要介绍其在 SSH 中的应用。

加密是为了防止未授权者会读取数据而对数据增加不规则性的过程。加密算法（*encryption algorithm*，或 *cipher*）是增加数据不规则性所采用的方法；现在流行的加密算法有 RSA、RC4、DSA 和 IDEA。最初的可读数据称为明文，加密后的数

据称为相应的密文。要把明文转换成密文，可以对数据使用一定的加密算法（使用密钥作为参数，密钥是一个字符串，通常只有你才知道密钥的内容）。如果任何人没有密钥就“根本不可能”读取（解密）密文，那么我们就认为这个加密算法必须是安全的。没有密钥而试图解密数据的行为称为密码分析（cryptanalysis）。

3.2.1 怎样才够安全？

我们要充分理解上一段中“根本不可能”的含义。现在大部分流行的安全加密算法都容易受到蛮力攻击（brute-force attack）：如果能遍历所有可能的密钥，那么最终就能成功解密密文。然而，当可能的密钥的范围十分巨大时，蛮力攻击的遍历操作就需要大量的时间和计算机资源才能完成。根据现在计算机硬件的处理能力和具体的加密算法，密钥的范围如果足够大，我们就不可能完全遍历所有可能的密钥，这样攻击者进行蛮力攻击时就不可能遍历搜索。但是，究竟什么数量级才是不可能的呢？这和数据的有用程度、要达到的安全时间以及攻击者的投入程度以及资源配置情况有关。使数据对一个攻击者保密几天是一回事情；使数据对整个世界组织保密10年则是另外一回事情。

当然，这都有一个前提，就是你必须能确认唯一可以破解这种加密算法的方法就是蛮力攻击。加密算法有一定的结构，可以进行数学分析。随着时间的推移，原来认为是安全的加密算法现在都已经被密码分析证明并不安全。现在不可能证明一个实际加密算法是安全的。加密算法要经过严格的数学分析和密码分析之后才能获得认可。如果一个新加密算法具有良好的设计规则，而且很多著名的研究者对其研究一段时间之后而没能找出一种实际可用的比蛮力攻击更快的方法来破解，那么人们就会认为这种加密算法是安全的（注1）。

注 1：数学家 Claude Shannon 对信息论和加密进行了开拓性的研究，他为加密算法的安全性定义了一种模型，并在这种模型下给出了一个绝对安全的算法：就是所谓的 *one-time pad*。这种算法相当安全：攻击者不能从加密过的数据中看到任何有关原明文的内容。密文可以解密成任何明文，而且概率相同。使用 *one-time pad* 的问题是它的使用非常麻烦而且脆弱。它要求密钥和要保护的信息一样长，而且要完全随机地生成，并且永远不会重用。如果这些条件中有一个不满足，那么 *one-time pad* 就不安全了。以 Shannon 的观点来看，现在使用的加密算法都不够安全，但是这些算法的好处是对蛮力攻击免疫。

3.2.2 公钥加密和密钥加密

到现在为止我们介绍的加密算法称为对称 (symmetric) 加密或密钥 (secret-key) 加密：加密和解密所使用的是同一个密钥。这种算法有 Blowfish、DES、IDEA 和 RC4。这种方法马上就引出了密钥分布 (key-distribution) 的问题：怎样才能获得接收者的密钥呢？如果你能和每个人都见一面并和他交换密钥，这当然没什么问题，但是对于现在通过计算机网络进行动态通信的这种模式来说，这是根本不可能的。

公钥（或非对称，*asymmetric*）加密使用一对相关密钥（一个公钥，一个私钥）来代替一个密钥。公钥和私钥使用一种很聪明的数学方法关联在一起：使用公钥加密过的数据可以使用对应的私钥解密；虽然私钥的分布是不可能的，但是公钥分布是可能的。保管好你的私钥，千万不要泄漏给别人；公钥可以给任何想要的人，不用担心会泄漏任何机密。理想地来说，你可以像电话本一样把它印到自己名字后面。当有人想要给你发秘密消息时，他可以使用你的公钥对数据进行加密。其他人也可能有你的公钥，但是他们不能解密该消息；只有你可以，因为你有相应的私钥。公钥加密在解决密钥分布问题上已经经历了很长一段历程（注 2）。

公钥方法也是数字签名的基础：数字签名是电子文档中附加的信息，它可以证明某个人已经阅读过文档并同意文档的内容，这就像是手写签名对书面文档的作用一样。所有的非对称加密算法（RSA、ElGamal、Elliptic Curve 等）都可以用作数字签名，反之则不行。例如，SSH-2 协议中的密钥使用 DSA 算法，它是只能用于签名的一种公钥模式，而不能用于加密（注 3）。

密钥加密算法和公钥加密算法还有一点不同：性能。所有通用的公钥加密算法的速度都要比密钥加密算法的速度慢得多——可能要慢几个数量级。使用公钥加密大量数据显然是不可能的。由于这个原因，现在的数据加密都是结合使用这两种方法。假设你想给朋友 Bob Bitflipper 安全发送数据，下面是现代加密程序执行的操作：

1. 生成一个随机密钥，称为 *bulk key*，用于快速密钥算法，例如 3DES（也就是 *bulk* 算法）。

注 2：这里仍然有一个严重的问题，就是如何判断哪个公钥是谁的；但是这涉及公钥结构或 PKI 系统的问题，是一个更广泛的主题。

注 3：其思想就是如此，尽管有人指出，可以方便地为 RSA 和 ElGamal 加密使用一个基本的 DSA 实现。然而这并不是它本来的设计目标。

2. 使用这个 bulk key 加密明文。
3. 使用 Bob Bitflipper 的公钥加密 bulk key，从而提高其安全性，这样只有 Bob 才能将其解密。由于密钥很小（最多也只有几百位），因此公钥加密算法的速度并不是什么问题。

要逆向这个操作，Bob 的解密程序首先解密 bulk key，然后使用它来解密密文。这种方法综合了两种加密技术的优点，实际上 SSH 就是使用这种技术。通过 SSH 连接传输的用户数据都是使用快速密钥加密算法加密过的，用于这种用途的密钥在客户端和服务器之间使用公钥加密的方法共享。

3.2.3 散列 (hash) 函数

在密码学（以及计算机和网络技术的其他领域）中，了解数据是否发生了变化是十分有用的。当然，一方可以同时发送（或简单地保留）原始数据进行比较，但这样会造成时间和存储空间的极大浪费。解决这个问题的通用方法称为散列函数。SSH-1 使用散列函数进行完整性检测（在密码学中散列函数还有一些其他用途，在此我们就不讨论了）。

散列函数简单地把一个大数据集映射到一个小数据集上。例如，散列函数 H 可以输入一个最多长达 50000 位的字符串，而统一产生一个 128 位的输出。其思想是当我给 Alice 发消息 m 时，同时要发一个散列值 $H(m)$ 。Alice 自行计算一下 $H(m)$ ，并将结果和我发的 $H(m)$ 进行比较；如果二者不同，那么她就知道该消息在传输过程中被修改了。

这种简单技术并不是一定有效的。由于散列函数的值域远远小于其定义域，因此很多消息都可能具有相同的散列值。一个散列函数 H 如果可用，必须具有这种特性：消息在传输过程中可能会遇到的修改必须会使消息的散列值也会变化。换而言之：给定一个消息 m 和一个对 m 修改过的 m' ，不能出现这种情况： $H(m) = H(m')$ 。

因此散列函数必须要根据用途进行筛选。散列函数的一个通用的用途是用于网络中：网络上传输的数据报通常会包含一个消息散列值，该值用来检测由于硬件失效或软件错误而产生的传输错误。另外一种用途是在密码学中用来实现数字签名。对大量数据进行签名的代价是很昂贵的，因为除了要使用速度很慢的公钥操作之外，还要发送完全加密过的数据。实际的解决方式是首先对文档应用散列函数，产生一个较

小的散列值，然后对该值进行签名，并发送这个签名过的散列值。验证者会自行计算散列值，然后才能使用适当公钥加密过的签名解密，并对二者进行比较。如果二者相同，他就认为签名（非常可能）有效，数据从用所有者的私钥签名之后未经修改过。

但是，这两种用法的要求不同，一个散列函数适用于检测由于线路噪声而引起的传输错误，但未必也适合于检测由入侵者所带来的蓄意修改。密码学中的散列函数必须确保不能通过计算找到两个不同的消息具有相同的散列值，也不能找到一个消息有其他散列值。这种函数称为防冲突的（*collision-resistant*，或称为*collision-proof*，虽然这有点容易误解）、防前映像的（*pre-image-resistant*）。用来检测意外的数据修改（例如，传输以太网帧时）所通用的循环冗余校验散列函数就是一个非防冲突的散列函数。我们很容易就可以找到一个 CRC-32 hash 冲突，SSH-1 的插入攻击就是根据这个原理进行的。^[3.10.5]密码学中的强散列函数有 MD5 和 SHA-1。

3.3 SSH 系统框架

SSH 有很多特有的交互作用的组件，这些组件产生了我们前面介绍的那些功能。^[3.1]图 3-1 对这些主要组件及其相互关系进行了说明。

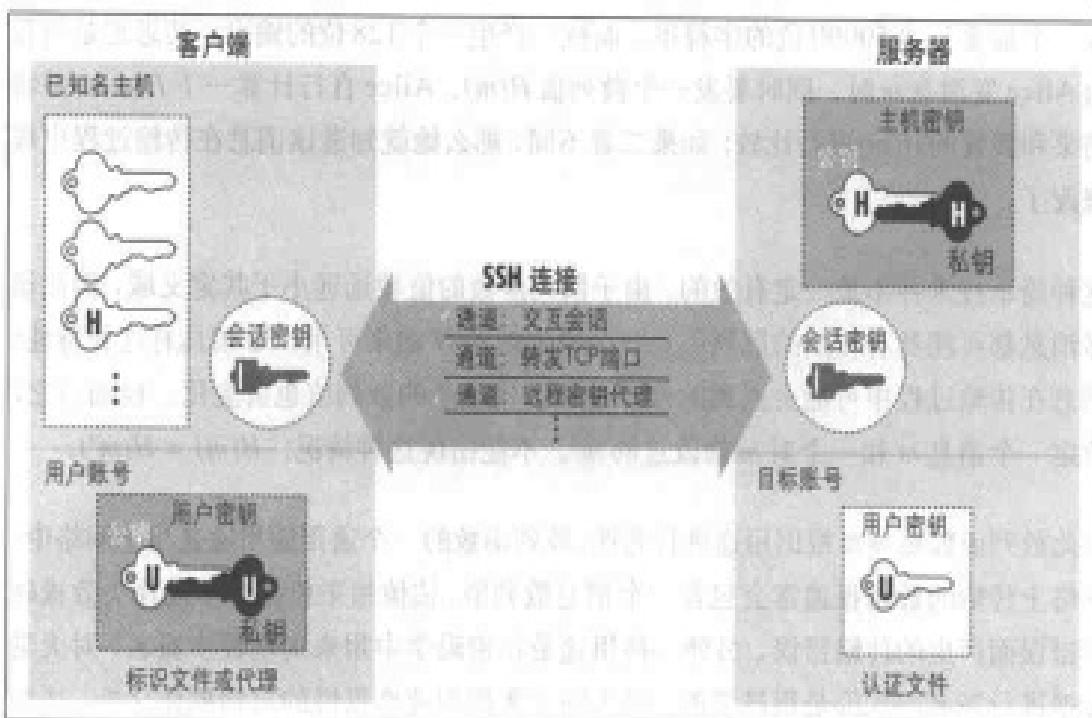


图 3-1：SSH 框架

我们使用“组件”这个词并不是说它一定是一个“程序”：SSH还有密钥、会话以及其他有趣的内容。在本节中我们会提纲挈领地对这些组件进行介绍，这样读者就可以对 SSH 大概有个印象了：

服务器

服务器是一个程序，它负责接收到达的 SSH 连接、进行认证、授权工作等等。

在大部分 Unix 的 SSH 实现中，服务器都是 *sshd*。

客户端

客户端是一个程序，它连接到 SSH 服务器上并发起请求（例如，“请让我登录”或“拷贝这个文件”）。在 SSH1、SSH2 和 OpenSSH 中，最主要的客户端是 *ssh* 和 *scp*。

会话

会话是客户端和服务器之间正在使用的连接。会话的始点是客户端和服务器成功实现认证，终点是连接结束。会话可以是交互式的，也可以是批处理式的。

密钥

密钥很小，通常只有几十位到一、二百位，用作诸如加密或消息认证之类的加密算法的参数。密钥的使用把算法操作和密钥所有者以某种方式绑定在一起：在加密中，它确保只有持有相应的密钥的用户可以对消息进行解密；在认证中，它允许你对实际给消息签名的密钥所有者进行验证。密钥有两种：一种是对称密钥或密钥；另外一种是非对称密钥或公钥。[3.2.2]非对称密钥有两部分：公钥和私钥。SSH 可以处理四种密钥，我们在表 3-1 中进行了总结，并在后面进行了说明。

表 3-1：密钥

名称	生存期	创建者	类型	用途
用户密钥	永久	用户	公钥	向服务器标识用户身份
会话密钥	一次会话	客户端（和服务器）	密钥	保护通信
主机密钥	永久	管理员	公钥	标识服务器 / 主机
服务器密钥	一小时	服务器	公钥	加密会话密钥（仅对 SSH1）

用户密钥

用户密钥是客户端用来证明用户身份的一个永久性的非对称密钥。（一个用户可以有多个密钥 / 身份标识。）

主机密钥

主机密钥是服务器用来证明自己身份的一个永久性的非对称密钥，这就和客户端在证明自己的主机身份时用于可信主机认证一样。[3.4.2.3]如果一台机器只运行一个SSH服务器，那么主机密钥还惟一标识这台主机的身份。（如果一台机器运行了多个SSH服务器，那么每个SSH服务器都可以有一个不同的主机密钥，或者它们共享一个主机密钥。）主机密钥很容易和服务器密钥混淆。

服务器密钥

服务器密钥是SSH-1协议中使用的一个临时的非对称密钥。每隔一定的间隔（缺省是一小时），它都在服务器上重新生成，用来保护会话密钥（稍后给出定义）。服务器密钥很容易和主机密钥混淆。服务器密钥从不保存在磁盘上，其私有部分也不会以任何形式通过连接传输；它为SSH-1会话提供了“完美的转发安全性（perfect forward secrecy）”。[3.4.1]

会话密钥

会话密钥是一个随机生成的对称密钥，用于对SSH客户端和服务器之间的通信进行加密。会话密钥由进行会话的双方在SSH连接建立时以安全的方式进行共享，因此窃听者无法窃取到会话密钥。然后会话双方就都有了这个会话密钥，它们用会话密钥加密通信。当SSH会话结束时，该密钥就被销毁。

注意：SSH-1使用一个会话密钥，而SSH-2使用多个会话密钥：每个方向的会话（服务器到客户端和客户端到服务器）都有几个密钥用于加密，还有几个用于完整性检查。在我们的讨论中，我们把所有的SSH-2会话密钥作为一个单位，为方便起见我们称之为“会话密钥”。如果上下文中需要，我们会说明我们指的是哪个密钥。

密钥生成器

密钥生成器是为SSH创建永久性密钥（用户密钥和主机密钥）的程序。SSH1、SSH2和OpenSSH的密钥生成器都是*ssh-keygen*。

已知名主机数据库

已知名主机数据库是主机密钥的集合。客户端和服务器根据这个数据库的内容彼此进行认证。

代理

代理是一个程序，它把用户密钥暂存在内存中，这样用户就不需要老是重复输入口令了。代理对有关密钥操作的请求进行响应，例如对认证者进行签名，但

是它不会泄漏密钥。这是一个十分方便的特性。SSH1、SSH2 和 OpenSSH 都有代理程序 *ssh-agent* 和 *ssh-add*，后者可以装载或清空密钥缓存。

签名者

签名者是对基于主机的认证报文进行签名的程序。我们将在讨论可信主机认证时详细介绍。[\[3.4.2.3\]](#)

随机数种子

随机数种子是 SSH 组件用来实现软件伪随机数生成器的一个随机池。

配置文件

配置文件是那些用来定制 SSH 客户端或服务器行为的设置的集合。

在一种 SSH 实现产品中这些组件并非全是必须的。当然服务器、客户端和密钥都是必须的，但是很多实现产品中并没有代理，有些甚至没有密钥生成器。

3.4 SSH-1 内幕

现在我们已经了解了 SSH 的主要特性和主要组件，接下来让我们深入研究一下 SSH-1 协议的细节问题。SSH-2 会在[\[3.5\]](#)中另外介绍。SSH-1 的框架在图 3-2 中进行了总结。我们在本节中将介绍以下内容：

- 安全会话是如何建立的。
- 密码认证、公钥认证和可信主机认证。
- 完整性检查。
- 数据压缩。

3.4.1 安全连接的建立

在进行有意义的交互会话之前，SSH 客户端和服务器必须首先建立一条安全连接。该连接可以允许双方共享密钥、密码，最后可以相互传输任何数据。

现在我们介绍 SSH-1 协议是如何确保网络连接的安全性的。SSH-1 客户端和服务器从开始经过很多个步骤，协商使用加密算法，生成并共享一个会话密钥，最终建立起一条安全连接：

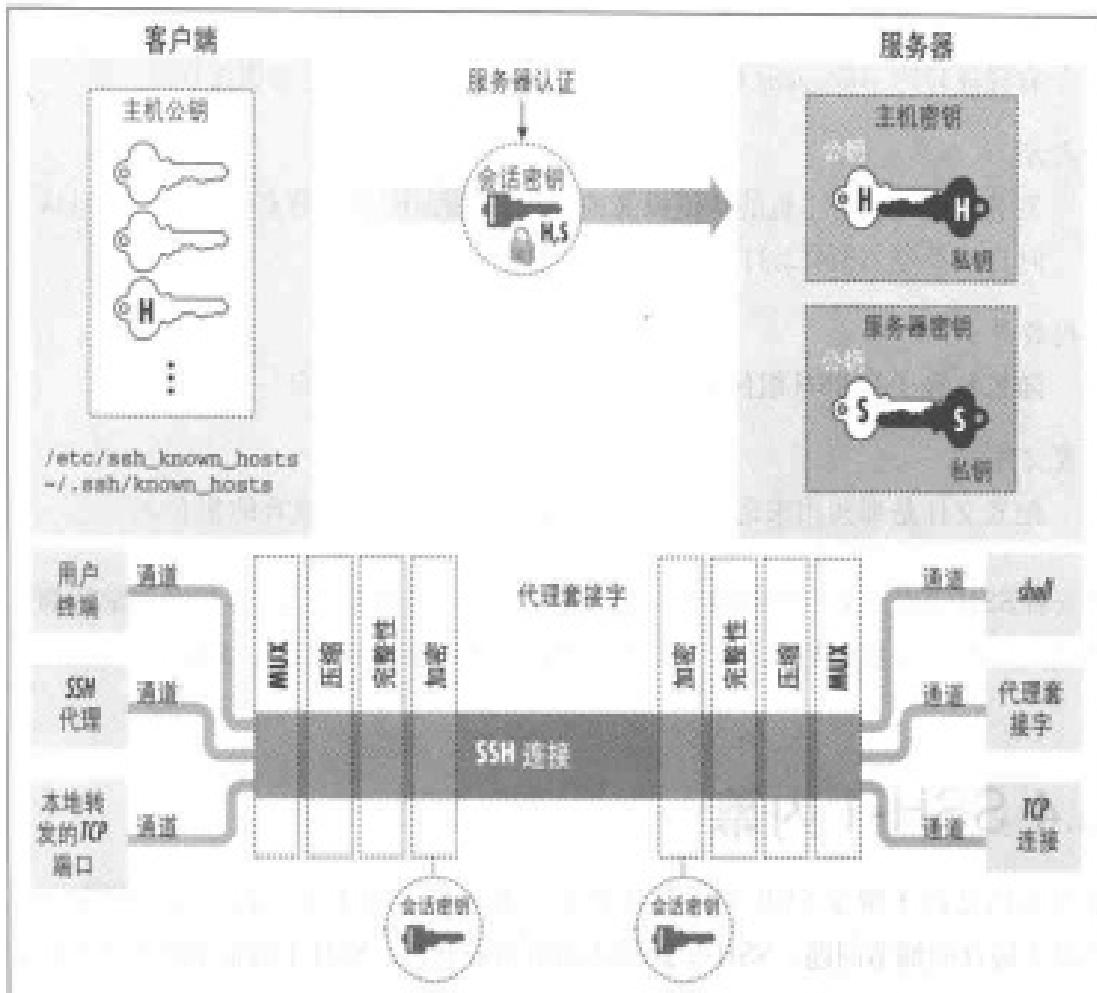


图 3-2: SSH-1 框架

1. 客户端连接到服务器。
2. 客户端和服务器交换自己支持的 SSH 协议版本号。
3. 客户端和服务器切换到基于报文的协议。
4. 服务器向客户端提供自己的身份证明和会话参数。
5. 客户端给服务器发送一个（会话）密钥。
6. 双方启用加密并完成服务器认证。
7. 建立安全连接。

现在，客户端和服务器就可以通过加密消息进行通信了。让我们详细看一下各个步骤；完整的处理过程如图 3-3 所示。

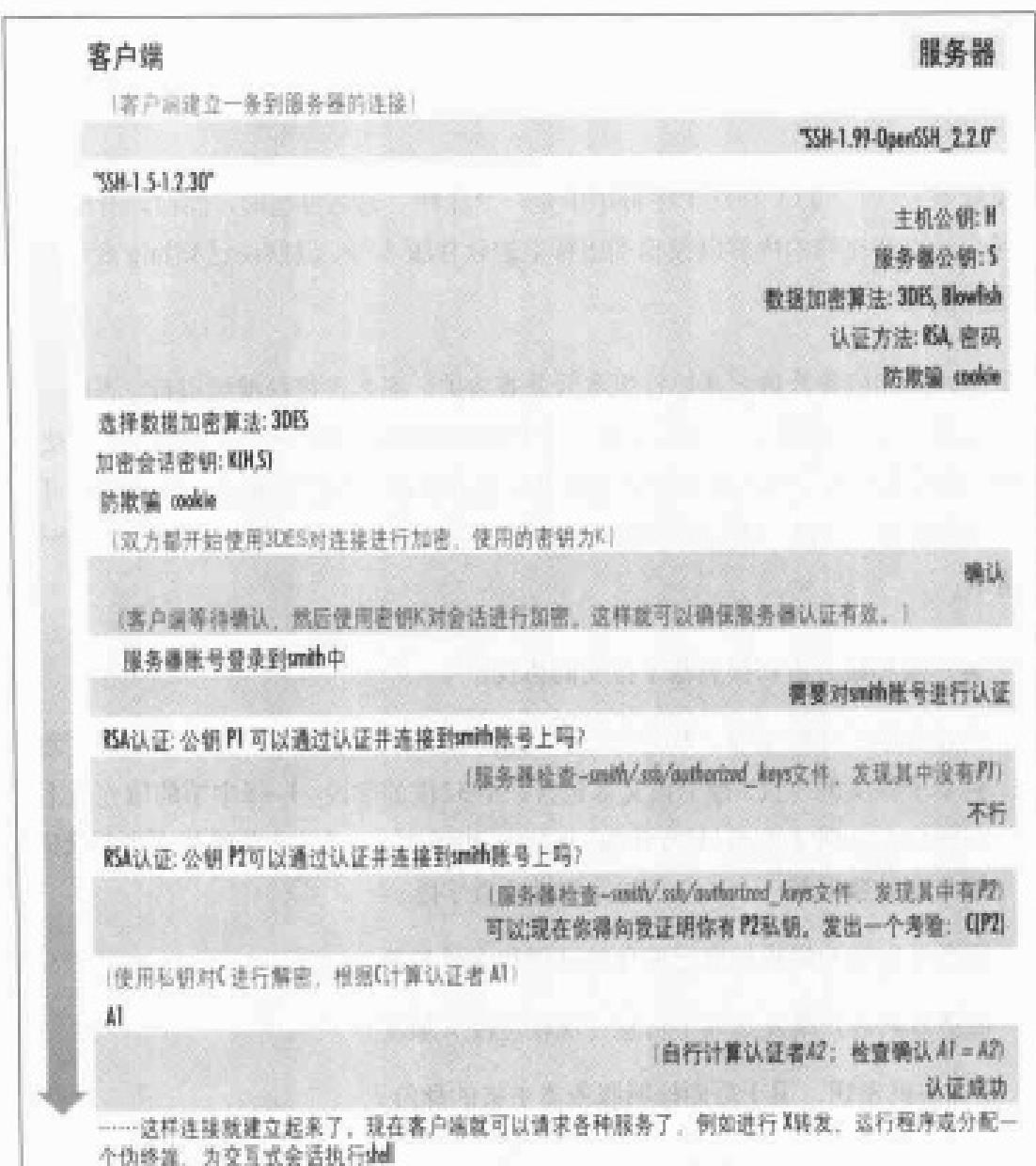


图 3-3: SSH-1 协议交换

1. 客户端连接到服务器上。

这个步骤没什么好说的，就是向服务器的 TCP 端口（约定是 22）发送连接请求。

2. 客户端和服务器交换自己支持的 SSH 协议版本号。

这些协议以 ASCII 字符串表示，例如，“SSH-1.5-1.2.27”，其意义为 SSH 协议版本号是 V1.5，SSH1 实现版本是 1.2.27。可以使用 Telnet 客户端连接到一个 SSH 服务器端口上看到这个字符串：

```
$ telnet server 22
Trying 192.168.10.1
Connected to server (192.168.10.1).
Escape character is '^]'.
SSH-1.5-1.2.27
```

实现版本号（1.2.27）在这个字符串中是一个注释，它是可选的。然而，有些 SSH 实现会检查该注释的内容以便识别出特定的软件版本号，以解决已知 bug 或不兼容问题（注 4）。

如果客户端和服务器确定其协议版本号是兼容的，那么连接就继续进行；否则，双方都可能决定中断连接。例如，如果一个只使用 SSH-1 的客户端连接到一个只使用 SSH-2 的服务器上，那么客户端就会断开连接并打印一条错误消息。实际上还可能执行其他操作：例如，只使用 SSH-2 的服务器可以调用 SSH-1 服务器来处理这次连接请求。

3. 客户端和服务器切换到基于报文的协议。

协议版本号交换过程一旦完成，客户端和服务器都立即从下层的 TCP 连接切换到基于报文的协议。每个报文都包含一个 32 位的字段，1~8 字节的填充位 [用来防止已知明文攻击 (known-plaintext attack)]，一个 1 字节的报文类型代码，报文有效数据和一个 4 字节的完整性检查字段。

4. 服务器向客户端提供自己的身份证明和会话参数。

服务器向客户端发送以下信息（现在还没有加密）：

- 主机密钥，用于后面证明服务器主机的身份。
- 服务器密钥，用来帮助建立安全连接。
- 8 个随机字节序列，称为检测字节 (check bytes)。客户端在下一次响应中必须包括这些检测字节，否则服务器就会拒绝接收响应信息。这种方法可以防止某些 IP 伪装攻击 (IP spoofing attack)。
- 该服务器支持的加密、压缩和认证方法。

注 4：有些系统管理员会把注释信息删除，不说明自己软件包的版本号，以免为攻击者提供线索。

此时，双方都要计算一个通用的128位会话标识符，它在某些协议中用来惟一标识这个SSH会话。该值是主机密钥、服务器密钥和检测字节一起应用MD5散列函数得到的结果。

当客户端接收到主机密钥时，它要进行询问：“之前我和这个服务器通信过吗？如果通信过，那么它的主机密钥是什么呢？”要回答这个问题，客户端就要查阅自己的已知名主机数据库。如果新近到达的主机密钥可以和数据库中以前的一个密钥匹配，那么就没有问题了。但是，此时还有两种可能：已知名主机数据库中没有这个服务器，也可能有这个服务器但是其主机密钥不同。在这两种情况中，客户端要选择是信任这个新近到达的密钥还是拒绝接受该密钥。[\[7.4.3.1\]](#)此时就需要人的指导参与了：例如，客户端用户可能被提示要求确定是接受还是拒绝该密钥。

如果客户端拒绝接受这个主机密钥，那么连接就中止了。让我们假设客户端接受该密钥，现在继续介绍。

5. 客户端给服务器发送一个（会话）密钥。

现在客户端为双方都支持的bulk算法[\[3.2.2\]](#)随机生成一个新密钥，称为会话密钥。其目的是对客户端和服务器之间发送的数据进行加密和解密。所需要做的工作是把这个会话密钥发送给服务器，双方就可以启用加密并开始安全通信了。

当然，客户端不能简单地把会话密钥发送给服务器。此时数据还没有进行加密，如果第三方中途截获了这个密钥，那么他就可以解密客户端和服务器之间的消息。此后你就和安全性无缘了。因此客户端必须安全地发送会话密钥。这是通过两次加密实现的：一次使用服务器的公共主机密钥，一次使用服务器密钥。这个步骤确保只有服务器可以读取会话密钥。在会话密钥经过两次加密之后，客户端就将其发送给服务器，同时还会发送检测字节和所选定的算法（这些算法是从第四步中服务器支持的算法列表中挑选出来的）。

6. 双方启用加密并完成服务器认证。

在发送会话密钥之后，双方开始使用密钥和所选定的bulk算法对会话数据进行加密。但是在开始继续发送其他数据之前，客户端要等待服务器发来一个确认消息，该消息（以及之后的所有数据）都必须使用这个会话密钥加密。这是最后一步，它提供了服务器认证：只有目的服务器才可以解密会话密钥，因为它

是使用前面的主机密钥（这个密钥已经对已知名主机列表进行了验证）进行加密的。

如果没有会话密钥，假冒的服务器就不能解密以后的协议通信，也就不能生成有效的通信，客户端会注意到这一点并中断连接。

注意服务器认证是隐含的；并没有显式交换来验证服务器主机密钥。因此客户端在继续发送数据之前，必须等待服务器使用新会话密钥作出有意义的响应，从而在处理之前验证服务器的身份。虽然 SSH-1 协议在这点上并没有什么特殊，但是 SSH-2 需要服务器认证时显式地交换会话密钥。

使用服务器密钥对会话密钥再进行一次加密就提供了一种称为完美转发安全性的特性。这就是说不存在永久性密钥泄漏的可能，因为它不会危害到其他部分和以后 SSH 会话的安全性。如果我们只使用服务器主机密钥来保护会话密钥，那么主机私钥的泄漏就会危害到以后的通信，并允许解密原来记录下来的会话。使用服务器密钥再加密一次就消除了这种缺点，因为服务器密钥是临时的，它不会保存到磁盘上，而且会周期性地更新（缺省情况下，一小时更新一次）。如果一个入侵者已经获取了服务器的私钥，那么他必须还要执行中间人攻击或服务器欺骗攻击才能对会话造成损害。

7. 建立安全连接。

由于客户端和服务器现在都知道会话密钥，而其他人都不知道，因此他们就可以相互发送加密消息（使用他们一致同意的 bulk 算法）并对其进行解密了。而且，客户端还可以完成服务器认证。我们现在就已经准备好开始客户端认证了。

3.4.2 客户端认证

安全连接一旦建立起之后，客户端就试图向服务器认证自己的身份。客户端此时会试验所有的认证方法，直到成功为止，否则等所有的方法都试过之后就认为失败。例如，SSH-1.5 协议中定义了六种认证方法，按照 SSH1 实现中进行认证所尝试的次序依次为：

1. Kerberos (注 5)

注 5：缺省情况下不能使用这种认证方法；要想使用这种认证，必须在编译时启用。

2. Rhosts
3. RhostsRSA
4. Public-key
5. TIS (注 5)
6. 密码 (次序: 主机登录密码、Kerberos、SecureID、S/Key 等等)

F-Secure 的 Windows 平台上的 SSH 客户端 (请参看第十六章) 尝试的次序依次为:

1. 公钥
2. 密码

了解客户端所使用的认证方法的次序是个好主意, 这有助于在认证失败或发生一些意外的情况时进行错误诊断。

3.4.2.1 密码认证

在密码认证过程中, 用户向 SSH 客户端提供一个密码, 客户端会把这个密码通过加密过的连接安全地发送给服务器。然后服务器就检查给定的密码是否能被目标账号接受, 如果可以就允许这次连接。在最简单的情况下, SSH 服务器通过主机操作系统所固有的密码认证机制来实现密码检查。

密码认证十分方便, 因为它不需要额外对用户进行设置。用户不需要生成一个密钥, 在服务器机器上创建一个`~/.ssh` 目录, 也不需要编辑任何配置文件。特别是对于第一次使用 SSH 的用户和经常出差并未携带自己的私钥的用户来说, 这尤其显得方便。你可能不想在其他机器上使用自己的私钥, 也可能根本就没有法子在要用的机器上获得私钥。如果你经常出差, 而你的 SSH 实现支持一次性密码, 那么就应该考虑使用一次性密码来设置自己的 SSH, 从而提高密码机制的安全性。[\[3.4.2.5\]](#)

从另外一方面来讲, 密码认证又很不方便, 因为每次连接到 SSH 服务器上时都要输入密码。而且, 密码认证的安全性不如公钥认证, 因为敏感的密码要传出客户端主机。它可以防止网络上的伪装欺骗攻击, 但是当它到达已经被攻击过的服务器时就很容易被截获。这和公钥认证不同, 后者即使被攻击过的服务器也不会通过协议

获得私钥。因此，在选择密码认证之前，应该衡量一下客户端和服务器的可信度，因为一旦私钥丢失，那么攻击者就获得了通往你的电子王国的钥匙。

虽然密码认证的概念很简单，但是不同的 Unix 变体保存密码和验证密码的方法都不同，这就导致了很多复杂性。OpenSSH 缺省情况下使用 PAM 进行密码认证，这必须要仔细配置。^[4.3]大部分 Unix 系统使用 DES（通过 `crypt()` 库程序）对密码进行加密，但是现在有些系统已经开始使用 MD5 散列算法，这也产生了一些配置问题。^[4.3]如果我们在 SSH 服务器中启用了 Kerberos^[5.5.1.7]或 SecureID^[5.5.1.9]的支持，那么密码认证的行为也会改变。

3.4.2.2 公钥认证

公钥认证使用公钥来验证客户端的身份。要访问 SSH 服务器上的一个账号，客户端必须证明自己有密钥：具体地说，就是已认证过的公钥的私有部分。如果认证文件（例如，`~/.ssh/authorized_keys`）含有一个密钥的公共部分，那么这个密钥就是认证过的。操作步骤为：

1. 客户端向服务器发送一个请求，请求使用一个特定的密钥进行公钥认证。该请求包含密钥的模数和一个标识符（注 6）。
密钥隐含地使用 RSA 算法；SSH-1 协议规定专门使用 RSA 算法进行公钥操作。
2. 服务器读取目标账号的认证文件，并搜索一个可以匹配该密钥的项。如果没有匹配项，那么这个认证请求就失败了。
3. 如果有匹配项，服务器就重取这个密钥并注意密钥使用的限制。之后服务器就可以根据限制规则立即拒绝请求了，例如，如果该密钥不能来自这台客户端主机。否则，处理过程就继续。
4. 服务器产生一个 256 位的随机字符串，将其作为一个“考验”，使用客户端的公钥对其进行加密并把结果发送给客户端。
5. 客户端接受这个“考验”并使用相应的私钥将其解密。然后将这个考验和会话标识符合并在一起，对结果应用 MD5 散列函数，并把散列值返回给服务器，作

注 6：一个 RSA 密钥由两个部分组成：指数（exponent）和模数（modulus）。模数是一个长数字，保存在公钥（.pub）文件中。

为客户端对考验的响应。会话标识符已经混合在其中，用来把认证者绑定到当前会话中，从而防止重放攻击利用这个缺点，或在创建这个考验时危害到随机数的生成。

此处的散列操作用来防止通过协议误用客户端的私钥，包括选择明文攻击 (chosen-plaintext attack) (注 7)。如果客户端简单地返回解密过的挑战，那么已被攻击的服务器就可以向客户端发送任何使用客户端的公钥加密过的数据，可怜的客户端还是忠心耿耿地把数据解密之后将其发回服务器。攻击者所截获的可能是解密 email 消息所使用的一个数据加密密钥。还有，请回想一下在 RSA 中，使用私钥“解密”数据实际上和对数据进行“签名”所执行的操作是类似的。因此服务器可以向客户端提供一些选定的、未加密的数据作为考验，这些数据要使用客户端的私钥进行签名——就仿佛一个文档全是“OWAH TAGU SIAM”甚至更难懂的东西。

6. 服务器对考验和会话 ID 计算相同的 MD5 散列函数值；如果客户端的响应和该值可以匹配，那么认证就成功了。

通常来讲，公钥方法在 SSH 中是最安全的认证方法。首先，客户端需要两个密文进行鉴别：私钥和解密私钥所使用的口令。二者泄漏一个都不会危及目标账号的安全（假设口令足够保险）。私钥很难猜测，而且永远不会离开客户端主机，因此偷窃私钥要比偷窃密码困难得多。保险的口令即使使用蛮力攻击也很难猜测得出来，而且如果必要，可以不用改变相关密钥就可以修改口令。还有，公钥认证不会信任客户端主机提供的任何信息；评判客户端是否可信的惟一标准就是私钥的所有权证明。这刚好和 RhostsRSA 认证形成了鲜明的对比，后者中服务器把部分的认证职权委托给客户端主机：在服务器对客户端的身份和优先级进行验证之后，就相信客户端软件不会对用户的身份撒谎了。[3.4.2.3]如果有人模仿一台客户端主机，那么他不用从用户手中偷取任何东西就可以模仿这台主机上的所有用户。这在公钥认证中是不可能发生的（注 8）。

注 7： 在选择明文攻击中，密码破译者可以把他选择的明文，和这个明文经他要破译的密钥加密形成的密文作比较。RSA 算法对这种攻击的抵御能力特别低，所以使用 RSA 的协议都要力图避免受到选择明文攻击。

注 8： 不过，请不要把模仿一台机器与入侵一台机器混为一谈。如果你真的攻破某台主机，危害到它的安全，那么说什么都没用了；你就可以偷走那台机器上所有用户的密钥、密码等。SSH 无法保护主机泄密。

公钥认证也是 SSH 中对认证进行控制最灵活的方法。在认证成功之后，可以对公钥进行适当的标记，从而反映出公钥以后使用的限制：哪些客户端主机可以连接，可以运行哪些命令，等等。^[8.2]与其说这是公钥方法的内在优点，倒不如说这是 SSH 实现的一个重要细节（注 9）。

如果我们向下比较，公钥认证也有一些不便之处，它比其他方法都复杂。公钥认证需要用户生成并维护自己的密钥和认证文件，这就可能导致出现一些错误：*authorized_keys* 中的语法不对，SSH 目录或文件中权限不正确，私钥文件的丢失需要新密钥并更新所有的目标账号，等等。SSH 并没有为密钥在大范围内的分布和管理提供任何管理机制。用户可以结合使用 SSH 和 Kerberos 认证系统（后者提供了这种管理机制），从而同时利用二者的优势。^[11.4]

警告：在连接 RSAref 加密库时就表现出了公钥认证的一个缺陷。^[3.9.1.1] RSAref 支持的密钥长度最大为 1024，而 SSH 内部的 RSA 软件可以支持更长的密钥。如果在 SSH/RSAref 中使用更长的密钥，那么就会出错。在用户和主机密钥中都会出现这个问题，如果你最近才开始使用 RSAref，原因可能在用户或者是从原来运行的非 RSAref 版本的 SSH 转换过来的密钥上。在这些情况中，只能使用更短的密钥来替换这些密钥。

3.4.2.3 可信主机认证 (*Rhosts* 和 *RhostsRSA*)

密码和公钥认证都需要客户端使用隐秘手段证明自己的身份：可能是密码，也可能是和服务器上特定的账号关联在一起的私钥。也就是说，客户端的位置（运行 *ssh* 计算机）和认证无关。

可信主机 (Trusted-Host) 认证则不同（注 10），它不用让用户向每个访问的主机都证明自己的身份，而是在主机之间建立信任关系。如果你使用用户 *andrew* 登陆到主机 A 上，同时使用 SSH 连接到主机 B 的账号 *bob* 上，主机 B 使用可信主机认证，那么主机 B 上的 SSH 服务器就不会直接检查你的身份。相反，主机 B 要检查主机 A 的

注 9： 我们倒是希望换一种方式实现。与其把认证和授权的功能这样搅在一起，不如实现一种与认证方法无关的控制，让 SSH 能对任何连接做任何限制。不过据我们所知还没有哪种 SSH 产品能使认证和授权完全无关。

注 10： 可信主机是我们自己的说法；它指的是一组相关的方法，包括 *Rhosts*、SSH-1、*RhostsRSA* 和 SSH-2 的基于主机的认证方法。

身份，确认主机 A 是一台可信主机。然后主机 B 还要进一步检查连接来自于主机 A 上的一个可信程序，该程序是由系统管理员安装的，它不会对 andrew 的身份撒谎。如果连接可以通过这两个测试，那么服务器就会告诉 A “你已经被证实是 andrew”，然后继续进行认证检测是否允许 andrew@A 访问 bob@B。

下面让我们一步一步地详细介绍一下整个认证过程：

1. SSH 客户端请求和 SSH 服务器建立连接。
2. SSH 服务器使用自己的本地名字服务来查找客户端网络连接的源地址的主机名。
3. SSH 服务器查阅服务器本地文件中的认证规则，判断特定的主机是否可以信任。如果服务器找到了主机名的匹配项，那么认证就继续进行；否则就失败。
4. 服务器按照原来的 Unix 特权端口的约定验证远程程序不是可信的程序。基于 Unix 的 TCP 和 UDP 栈把 1 到 1023 端口保留用作特权端口，只允许以 root 身份运行进程监听这些端口并在本地连接中使用这些端口。服务器简单地检查连接的源端口是否在特权端口范围内。假设客户端主机是安全的，只有超级用户可以调度程序发起这种连接，那么服务器就相信自己正在和一个可信程序进行通信。
5. 如果一切顺利，现在认证就成功了。

这个过程在 Berkeley 的 r- 命令族 (*rsh*、*rlogin*、*rcp*、*rexec* 等) 中已经应用了很多年了。不幸的是，现代网络中的认证方法实在太脆弱了。IP 地址可以伪装，名字服务可以篡改，特权端口在 PC 中也没什么特权，因为在 PC 桌面操作系统中终端用户通常都具有超级用户（系统管理员）特权。实际上，有些桌面操作系统缺少用户的概念（例如，MacOS），还有些操作系统根本就没有实现特权端口的约定（Windows），因此任何用户都可以访问空闲端口。

然而，可信主机认证也有一些优点。首先是它非常简单，不用输入密码或口令，也不用生成、分布并管理密钥。可信主机还为自动操作提供了便利。如果脚本中需要密钥、口令或密码，或者将其保存在受保护文件或内存中，那么诸如 *cron* 任务之类的无人参与的进程要使用 SSH 就有些困难。这不仅存在安全风险，而且维护也是件可怕的事。如果认证者发生了变化，那么就必须穷追不舍，把硬盘上的所有备份全

部修改掉，稍后我们会看到一种由于这个原因而引起漏洞的情形。可信主机认证巧妙地避免了这个问题。

由于可信主机认证的思想十分有用，SSH1以两种方式对其提供支持。Rhosts 认证简单地按照前面介绍的步骤 1 到步骤 5 执行，这和 Berkeley 的 r- 命令一样。缺省情况下这种方法是禁用的，因为它不够安全，然而这对于 rsh 来说还是一个很大的提高，因为它提供了服务器主机认证、加密和完整性。更重要的是，SSH1 提供了一种更安全的可信主机认证方法，称为 RhostsRSA 认证，它使用客户端的主机密钥改进了前面的步骤 2 和步骤 4。

步骤 2 中要对客户端主机的完整性执行更严格的检查。SSH 不需要依赖于源 IP 地址和 DNS 之类的名字服务，而使用公钥加密。回想一下安装 SSH 的每个主机都有一个非对称“主机密钥”对自己的身份进行标识。主机密钥负责在建立安全连接时向客户端认证服务器。在 RhostsRSA 认证中，客户端的主机密钥负责向服务器认证客户端主机。客户端主机提供自己的主机名和公钥，接着通过考验——响应交换证明自己拥有相应的私钥。服务器维护了一个已知名主机及其公钥的列表，使用这个列表可以判断客户端是否是已知名的可信主机。

步骤 4 负责验证服务器正和一个可信程序通信，我们可以使用客户端的主机密钥对这个步骤进行改进。私钥的保存非常隐秘，因此只有具有特殊权限（例如，setuid 为 root）的程序可以读取私钥。因此，如果客户端可以访问自己的本地主机密钥（这必须执行步骤 2 中的整个认证过程），就必须具有特殊的权限。客户端是由系统管理员在可信主机上安装的，因此也是可信的。SSH1 中保留了特权端口检测，这个特性不能禁用（注 11）。SSH2 彻底舍弃了这种检测，因为它没有增加任何有用的内容。

可信主机访问控制文件。在 SSH 服务器上有一对文件用来为可信主机认证提供访问控制，这两个文件控制的强弱程度不同：

- */etc/hosts.equiv* 和 *~/.rhosts*
- */etc/shosts.equiv* 和 *~/.shosts*

注 11：SSH1 有一个 *UsePrivilegedPort* 配置关键字，但它的意思是让客户端不要使用特权端口作为源 socket 端口，否则会导致不能使用 rhosts 和 RhostsRSA 认证。该特性的目的是解决防火墙可能会阻塞来自特权端口的连接的问题，并请求使用其他认证方法。

目标账号主目录中的文件是特定于该账号的，而`/etc`中的文件的作用域是整台机器。`hosts.equiv`和`shosts.equiv`语法相同，`.rhosts`和`.shosts`的语法也相同，缺省情况下，系统对这四个文件都要进行检测。

警告：如果这四个访问文件中有一个文件中允许某个特定的连接，即使其他文件中禁止这个连接，那么这个连接也是允许的。

`/etc/hosts.equiv`和`~/.rhosts`文件起源于不安全的`r-`命令。为了保持向后兼容性，SSH也使用这两个文件进行可信主机认证的决策。然而，如果同时使用`r-`命令和SSH，那么你恐怕不想让两个系统使用相同的配置。而且，由于`r-`命令的安全性不好，通常我们都在`inetd.conf`文件中禁用`r-`命令的服务器，并删除相应的程序。在这种情况下，你可能不想再使用这两个传统的控制文件，这是为了防范攻击者再重新启用这些服务。

为了把这两个文件和`r-`命令分离开，SSH采用了另外两个文件：`/etc/shosts.equiv`和`~/.shosts`，其语法与`/etc/hosts.equiv`和`~/.rhosts`文件相同，但是是专门为SSH使用的。如果只使用SSH专用的文件，就可以使用SSH可信主机认证，完全可以丢弃`r-`命令使用的这两个文件（注12）。

这四个文件的语法都相同，SSH对这四个文件的解释也和`r-`命令相似，当然并不是完全相同。请仔细阅读以下的内容确保你正确理解了这种行为。

控制文件的细节。这四个可信主机控制文件具有相同的格式。文件中的每项都是一行，其中包含一段或两段内容，使用制表符或空格分隔开。注释以`#`开始，直到行尾为止，可以放在任何地方；文件中允许出现空行和注释行。

```
* 控制文件条目样例
[+-][@]hostspec [+-][@]userspec # comment
```

上面的两段内容分别代表主机和用户；其中`userspec`可以省略。如果出现了`@`符号，那么这段内容就被解释成一个网络组（`netgroup`，请参看后文中的“网络组”），并

注12：不幸的是，并不能把服务器配置成只查看一对文件，而不查看另外一对文件。如果SSH服务器要使用`~/.shosts`，那么它也会使用`~/.rhosts`文件，另外还会使用两个全局文件。

调用 `innetgr()` 函数进行查找，结果会给出用户名和主机名。否则，这段内容就被解释成一个主机或用户名。主机名必须符合服务器主机上 `gethostbyaddr()` 的约定；否则主机名就不能正常使用。

网络组 (netgroup)

网络组定义了一组（主机名、用户名、域名）三元组。网络组用来定义用户、主机或账号的关系列表，通常用于访问控制；例如，我们通常可以使用网络组来规定允许哪些主机可以装载（mount）NFS 文件系统（例如，在 Solaris 的 `share` 命令或 BSD 的 `exportfs` 中）。

虽然不同风格的 Unix 对于网络组的实现也不同，但有一点是相同的：只有系统管理员才能定义网络组。网络组定义的可能来源包括：

- 正文文件，例如，`/etc/netgroup`。
- 不同格式的数据库文件，例如，`/etc/netgroup.db`。
- 信息服务，例如，Sun 的 YP/NIS。

在大部分现代的 Unix 中，网络组的信息源可以使用 Network Service Switch 工具进行配置；请参看 `/etc/nsswitch.conf` 文件。注意在 SunOS 和 Solaris 的有些版本中，网络组只能在 NIS 中定义；如果你在 `nsswitch.conf` 文件中指定“文件”作为网络组的信息源，也是可以的，但是这不能正常工作。虽然现在的 Linux 系统可以支持 `/etc/netgroup`，但是 `glibc 2.1` 之前版本的 C 库函数只支持使用 NIS 定义网络组。

典型的网络组定义如下：

```
* 定义两个主机为一组：主机名 "print1" 和 "print2"。  
* 在（可能是 NIS）域 one.foo.org 和 two.foo.com 中  
print-servers      (print1,,one.foo.com) (print2,,two.foo.com)  
# 三个登录服务器的列表  
login-servers     (login1,,foo.com) (login2,,foo.com)  
(login1,,foo.com)  
# 使用现有的网络组来定义所有的主机  
# 以及 another.foo.com  
all-hosts          print-servers login-servers (another,,foo.com)
```

— 待续 —

```
# 用于控制访问的用户列表。Mary 可允许从 foo.com  
# 域的任意地方来，Peter 则只能来自一台主机。  
# Alice 完全可从任意地方来访问  
allowed-users (.mary,foo.com) (login1,peter,foo.com) (,alice,)
```

在确定一个网络组内部的成员关系时，我们通常都将正在匹配的内容解释成一个三元组。如果在网络组 N 中存在一个三元组 (a, b, c) 和 (x, y, z) 匹配，那么我们就说 (x, y, z) 可以和网络组 N 匹配。同理，我们可以这样定义两个三元组匹配，当且仅当满足以下条件：

$x = a$, 或 x 为 null, 或 a 为 null

并且

$y = b$, 或 y 为 null, 或 b 为 null

并且

$z = c$, 或 z 为 null, 或 c 为 null

也就是说三元组中的空元素可以和任意通配符匹配。我们使用“null”来表示该元素不存在的情况，也就是说，在三元组 $(, user, domain)$ 中，主机部分的元素为 null。注意这和空字符串 $("", user, domain)$ 不同。后面这个三元组中，主机部分并不为 null；它实际上是一个空字符串，这个三元组只能和其他主机部分也是空字符串的三元组匹配。

当 SSH 要在网络组中匹配用户 U 时，它应该匹配三元组 $(, U,)$ ；同理，SSH 要匹配主机 H 时，它应该匹配三元组 $(H, ,)$ 。你可能会想也许可以使用 $(, U, D)$ 和 $(H, , D)$ 这样的三元组，其中 D 是主机域名；但是这样是不允许的。

如果这两段内容中有一段或者两段前面都有一个减号（-），那么该项整个就认为应该取否定值。不管哪个段前面出现减号；效果都是一样的。在解释详细规则之前让我们先看几个例子。

如果远程用户名和本地用户名相同，那么下面的 *hostspecs* 可以允许来自 *fred.flintstone.gov* 的所有用户登录到系统中：

```
# /etc/shosts.equiv
fred.flintstone.gov
```

如果远程用户名和本地用户名相同，那么下面的 *hostspec*s 可以允许来自网络组“trusted-hosts”的所有主机上的用户登录到系统中，但是禁止来自 *evil.empire.org* 主机的用户登录，即使该主机在 trusted-hosts 网络组中也不行。

```
# /etc/shosts.equiv
-evil.empire.org
@trusted-hosts
```

下面的 *hostspec* 和 *userspec* 允许 *mark@way.too.trusted* 登录到本地任何账号中。即使某个用户在 *~/.shosts* 文件中指定了 *-way.too.trusted mark*，也不能阻止 *mark@way.too.trusted* 访问本账号，因为全局文件的优先级比较高。你可能永远都不会这样使用。

```
# /etc/shosts.equiv
way.too.trusted mark
```

另一方面，下面的内容允许来自 *sister.host.org* 主机上除 *mark* 之外的任何用户都可以使用相同的账号名连接，*mark* 不能访问本地账号。但是要记住，如果我们在 *~/.shosts* 文件中加入 *sister.host.org mark*，那么目标账号的设置就会覆盖这种限制。而且正如我们前面已经说过的一样，否定行只能放在前面；如果顺序颠倒了，设置就无效。

```
# /etc/shosts.equiv
sister.host.org -mark
sister.host.org
```

下面的 *hostspec* 允许 *fred.flintstone.gov* 上的用户 *wilma* 登录到本地的 *wilma* 账号中：

```
# ~wilma/.shosts
fred.flintstone.gov
```

这段内容允许用户 *fred.flintstone.gov* 上的用户 *fred* 登录到本地的 *wilma* 账号中，但其他人都不行，即使是 *wilma@fred.flintstone.gov* 也不行。

```
# ~wilma/.shosts
fred.flintstone.gov fred
```

下面的设置允许 *fred.flintstone.gov* 上的 *fred* 和 *wilma* 都可以登录到本地 *wilma* 账号中：

```
# ~wilma/.shosts
fred.flintstone.gov fred
fred.flintstone.gov
```

现在我们已经看过了一些例子，接下来让我们讨论详细的规则。假设客户端用户名是 C，SSH 命令的目标账号是 T。那么：

1. 当 T=C 时，*userspec* 为 空的 *hostspec* 项表示允许来自所有 *hostspec* 主机的连接用户访问。
2. 当 C 为 *userspec* 用户名之一时，在每个账号的文件（*~/.rhosts* 或 *~/.shosts*）中，一个 *hostspec userspec* 项表示允许来自 *hostspec* 主机的连接访问 *userspec* 中包含的用户账号。
3. 当 C 为 *userspec* 用户名之一时，在全局文件（*/etc/hosts.equiv* 或 */etc/shosts.equiv*）中，一个 *hostspec userspec* 项表示允许来自任何 *hostspec* 主机的连接访问任何本地账号。
4. 对于否定项来说，以上三条规则中的“允许”应改为“禁止”。

仔细看一下规则 3，你永远都不希望在自己的机器上留下这样一个安全漏洞吧？这条规则的唯一一个用处就是在否定项中使用，这样就可以禁止一个特定的远程账号访问所有的本地账号。我们稍后会给出一些例子。

这些文件的检查顺序（如果文件不存在就简单地将其跳过，这对认证决定没什么影响）为：

1. */etc/hosts.equiv*
2. */etc/shosts.equiv*
3. *~/.shosts*
4. *~/.rhosts*

当目标账号是 root 时，SSH 要进行一些特殊的处理：此时它不会检查全局文件。对于 root 账号的访问只能通过 root 账号的 *.rhosts* 和 *.shosts* 文件进行授权。如果使用 *IgnoreRootRhosts* 服务器标志禁用了这两个文件，就要通过可信主机认证有效地控制对 root 账号的访问。

在检测这些文件时，要记住两条规则。第一条规则是：先接受的行优先。也就是说，如果有两个网络组：

```
set      (one,,) (two,,) (three,,)
subset   (one,,) (two,,)
```

那么下面的`/etc/shosts.equiv`文件只允许来自主机`three`（译者注，指第三台主机）的连接访问：

```
-@subset
@set
```

但是下面的配置则可以允许来自这3台主机的连接都可以访问：

```
@set
-@subset
```

第二行无效，因为第二行中的所有主机已经被第一行接受了。

第二条规则是：如果任何文件可以接受一个连接，那么就允许这个连接访问。也就是说，如果`/etc/shosts.equiv`文件禁止一个连接，而目标账号的`~/.shosts`文件接受这个连接，那么这个连接就是可以接受的。因此，系统管理员不能依赖全局文件阻塞连接。同理，如果你的每账号文件中禁止了一个连接，那么这个连接也可能被一个接受该连接的全局文件的设置覆盖。在使用可信主机认证时要牢记这两条规则（注13）。

网络组中的通配符。你可能已经注意到规则语法中没有通配符；之所以这样决定是经过深思熟虑的。`r-`命令可以识别纯“+”和“-”字符，分别将其作为肯定和否定的通配符。很多攻击者可以秘密地在用户的`.rhosts`文件中加上“+”，这样立即就允许任何用户都可以以这个用户的身份登录。因此SSH经过仔细考虑之后丢弃了这些通配符。如果你碰到了这种通配符，就会在看到它对服务器调试输出的影响，所显示的信息为：

```
Remote: Ignoring wild host/user names in /etc/shosts.equiv
```

然而，还有一种方法可以利用通配符：在网络组中使用通配符。一个空网路组：

注13：把服务器的`IgnoreRhosts`关键字设置成`yes`，就可以让服务器彻底忽略每账号文件，转而只使用全局文件。

```
empty # 无内容
```

不能匹配任何内容。但是，这个网络组：

```
wild (,,)
```

可以匹配所有的三元组。实际上，一个网络组在任何地方包含 (,,)，都可以匹配任何三元组，不管这个网络组中的其他内容如何都是如此。因此这样的配置：

```
# ~/.shosts
@wild
```

只要远程用户名和本地用户名可以匹配，那么它就允许来自所有主机的连接都可以访问（注 14）。这个配置：

```
# ~/.shosts
way.too.trusted @wild
```

允许 *way.too.trusted* 中的所有用户都可以登录到这个账号中，而这个配置：

```
# ~/.shosts
@wild @wild
```

允许来自所有主机的用户都可以访问。

知道了这个通配符的操作，我们在定义网路组时就应该非常谨慎了。创建一个通配符网络组要比想像的简单得多。包括一个空三元组 (,,) 是很明显的一种方法。但是，要注意网络组三元组中元素的顺序是 (*host,user, domain*)。假设这样定义一个“oops”网络组：

```
oops      (fred,,) (wilma,,) (barney,,)
```

你本来是希望把这作为一个用户组使用，但是在 *host* 位置放入了用户名，而把 *username* 保留为 null 了。如果把这个组用作一条规则的 *userspec*，那么它就变成一个通配符了。因此这个配置：

```
# ~/.shosts
home.flintstones.gov @oops
```

注 14： 如果现在使用的是强加密可信主机认证，那么这就表示任何主机都要使用公钥对服务器的已知名主机数据库进行验证。

允许 `home.flintstones.gov` 上的所有用户都可以登录到你的账号上来，不只是你的三个朋友。千万要小心。

小结。 可信主机认证对于用户和管理员来说都非常方便，因为它可以根据用户名对应关系和主机交互信任关系在主机之间设置自动进行认证。这减少了用户重复输入密码的负担，也减小了密钥管理的工作量。然而，可信主机认证要严重依赖于系统配置的正确性和所涉及的主机的安全性；一台可信主机被攻克就给了攻击者自动访问其他主机上所有账号的权利。而且，访问控制文件所使用的规则非常复杂而且脆弱，很容易用错而危及系统的安全。在对窃听和泄漏的危险性要求比主动攻击（active attack）更高的环境中，更理想的方法是使用 RhostsRSA（SSH-2 “基于主机的”）认证进行普通用户的认证。虽然它可以满足特殊目的的用户的有限需要，但在安全性要求更高的环境中（例如无人值守的批处理任务），这还不够。^[11.1.3]

我们不推荐在 SSH1 和 OpenSSH/1 中使用这种脆弱的（“Rhosts”）可信主机认证；它不够安全。

3.4.2.4 Kerberos 认证

SSH1 和 OpenSSH 支持基于 Kerberos 的认证；SSH2 则不行。^[11.4]（注 15）表 3-2 对这些产品支持的功能进行了总结。

表 3-2：SSH 中对 Kerberos 认证的支持

产品	Kerberos 版本号	许可证	密码认证	AFS	转发
SSH1	5	支持	支持	不支持	支持
OpenSSH	4	支持	支持	支持	只有 AFS 支持

下面我们将对上表中的内容进行解释：

许可证（ticket）

执行标准的 Kerberos 认证。客户端获得一个许可证，用于服务器上的“host”（V5）或“rcmd”（V4）服务，并将其发给 SSH 服务器作为身份证明；服务器对许可证进行标准的验证。SSH1 和 OpenSSH 都要进行 Kerberos 相互认证。严

注 15： 在 SSH2 2.3.0 发布时，集成了 Kerberos 的实验性支持。

格地讲这没什么必要，因为 SSH 在建立连接时已经对服务器进行了认证，但是再进行一次检查毕竟没什么坏处。

密码认证

选择密码认证可以使用 Kerberos 执行服务器端的密码认证。密码认证不使用操作系统的账号数据库检查密码，而是由 SSH 服务器为目标账号获得 Kerberos 的原始证书（“可更新许可证”，或 TGT）。如果这个步骤能成功执行，那么用户就通过了认证。服务器还把 TGT 存储起来供会话使用，这样用户就可以访问这个 TGT，从而就不需要使用 *kinit* 了。

AFS

Andrew 文件系统 (<http://www.faqs.org/faqs/afs-faq/>)，或称为 AFS，使用 Kerberos-4 以一种特殊的方法进行认证。OpenSSH 也可以支持 AFS 证书的获取与转发，这在使用 AFS 共享文件的环境中是十分重要的。在进行认证之前，*sshd* 必须要读取目标账号的用户主目录，例如检查 `~/.shosts` 或 `~/.ssh/authorized_keys` 文件。如果用户主目录是通过 AFS 进行共享的，那么根据 AFS 权限的设置，*sshd* 可能只有在获得账号所有者的有效 AFS 证书之后，才可以读取用户主目录。OpenSSH AFS 代码提供了这种功能，把源用户的 Kerberos-4 TGT 和 AFS 证书转发给远程主机的 *sshd* 使用。

转发

Kerberos 证书通常都只能在生成这些证书的机器上使用。Kerberos-5 协议允许用户把 Kerberos 证书从一台机器转发到另外一台自己已经认证过的机器上，从而避免了反复调用 *kinit* 的麻烦。SSH1 使用 *KerberosTgtPassing* 选项来支持这个功能。Kerberos-4 不能执行证书转发，因此 OpenSSH 也没有提供这种特性——除非 OpenSSH 使用了 AFS，也对 Kerberos-4 实现进行了修改，从而提供了一种证书转发功能。

注意：OpenSSH 只有在使用 SSH-1 协议时才支持 Kerberos。

3.4.2.5 一次性密码

密码认证十分方便，因为它在任何地方都可以方便地使用。如果你经常旅行，经常使用别人的计算机，那么密码就是你使用 SSH 认证的最好方式。然而，这却恰巧碰

到了这种情况：你最关心是否有人可能窃取你的密码——可能是通过监视已经被攻击过的计算机上的键盘操作进行，也可能是通过原来的“肩窥(shoulder-surfing)”攻击进行。一次性密码，或称为 OTP 系统，既方便了密码访问的使用，同时还减轻了密码被窃取的风险：用户每次登录时都需要提供一个不同的、不可预知的密码。以下是 OPT 系统的一些特性：

- 使用免费的 S/Key 软件 OTP 系统，可以打印出一个密码列表，或者计算出在使用掌上电脑或 PDA 上的软件所需要的下一个密码。
- 使用 RSA Security, Inc. 的 SecureID 系统，用户可以获得一个带 LCD 屏幕的小硬件认证器（大小和信用卡或钥匙链差不多），它可以显示一个频繁修改的密码并和 SecureID 服务器保持同步，SecureID 服务器负责对密码进行验证。
- Trusted Information Systems, Inc. (TIS) 的 OTP 系统是一个“考验 - 响应(challenge-response)”的变种：服务器显示一个考验，用户要输入自己的软件密码或硬件显示的密码。它负责提供相应的响应，用户在其中提供内容用来进行认证。

SSH1 把 SecureID 当作密码认证行为的一个变种，而将 TIS 作为使用 TISAuthentication 配置关键字（正如我们前面已经介绍过的一样，这在 SSH-1 协议中实际上是一种单独的认证类型）的另外一种方法。OpenSSH 不支持 TIS，但是它在 SSH-1 协议中重用了 TIS 消息类型来实现 S/Key。这种方法之所以能正常工作是因为 TIS 和 S/Key 都符合同一个考验 / 响应模型。

对这些系统的使用包括获得必需的库文件和头文件，使用适当的配置开关编译 SSH，启用正确的 SSH 认证方法，还要根据用途设置系统。如果你正在使用 SecureID 或 TIS，那么必需的库文件和头文件就应该和软件本身一起提供的，也可以从提供商那里获得。S/Key 可以广泛地在网络上使用，但是它又分成很多版本，我们不知道它是否有一个标准站点。我们可以在 Wietse Venema 的 *logdaemon* 包中找到一个通用的实现；请参看 <http://www.porcupine.org/wietse/>。这些包的具体内容通常已经超出了 SSH 的范围，因此我们就不进行深入介绍了。

3.4.3 完整性检查

SSH-1 协议使用了一种弱完整性检查：32 位的循环冗余校验，或称为 CRC-32。这

种检查对于数据偶然造成的修改来说是足够的，但是它不能有效地检测出数据是否被蓄意修改了。实际上，Futoransky 和 Kargieman 的“插入攻击（insertion attack）”就是专门利用了 SSH-1 中的这种缺陷。[\[3.10.5\]](#) 使用 CRC-32 进行完整性检测的工作是从 SSH-1 中继承下来的一个严重缺陷，这也促使 SSH-2 对其进行改进，后者使用强加密安全检测，完全可以防止这种攻击。

3.4.4 压缩

SSH-1 协议支持使用 GNU 的 *gzip* 工具 (<ftp://ftp.gnu.org/pub/gnu/gzip/>) 的“deflate”算法对会话数据进行压缩。每个方向上的报文数据都是单独进行压缩的，每个都被当成是一个大数据流，而不用考虑报文边界。

虽然在 LAN 和快速 WAN 链路中通常并不需要压缩，但是在慢速链路（例如 modem 线路）中使用压缩可以显著地提高传输速度。特别是对于文件传输、X 转发以及在终端会话中运行的 *curses* 风格的程序（例如，文本编辑器）来说尤其有效。而且，由于压缩是在加密之前进行的，因此使用压缩，就可以减小加密的延时。这在 3DES 的情况下尤其有效，因为它非常之慢。

3.5 SSH-2 内幕

在本节中，我们讨论 SSH-2 的设计和内部细节，不再重复这两种协议的共同内容，而是重点介绍 SSH-2 和 SSH-1 的不同和改进之处，我们还会对 SSH1 和 SSH2 的产品、二者的软件实现的差异以及所支持的协议进行比较。图 3-4 对 SSH-2 的框架进行了总结。

SSH1 和 SSH2 之间最重要的区别是它们支持不同的而且不兼容的 SSH 协议版本：SSH-1.5 和 SSH-2.0。[\[1.5\]](#) 这些产品的实现也有很大的不同，部分原因是由于协议的不同，而更多是由于 SSH2 对代码进行了完全重写。

3.5.1 (SSH-1 和 SSH-2 之间的) 协议差异

SSH-1 是一个单一的大模块，在一个协议中包含了很多功能。SSH-2 则不同，它被划分成很多模块，包括三个协同工作的协议：

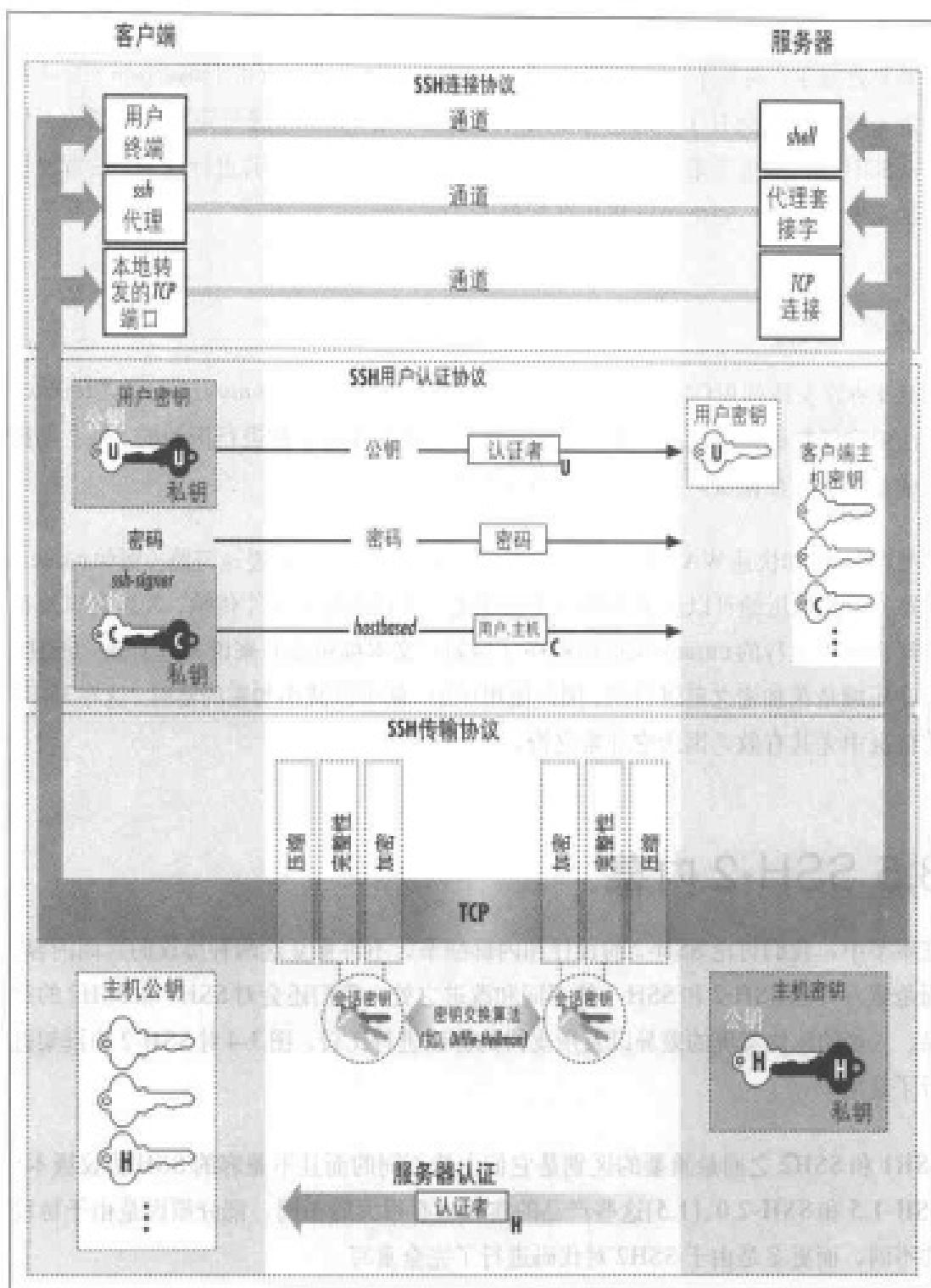


图 3-4: SSH-2 的框架

- SSH 传输层协议 (SSH-TRANS)
- SSH 认证协议 (SSH-AUTH)

- SSH 连接协议 (SSH-CONN)

这三个协议都有单独的文档进行定义，还有第四个文档，也就是 SSH 协议框架 (SSH-ARCH)，它在对这三个单独的规范进行深入分析的基础上描述了 SSH-2 协议的完整框架。

图 3-5 概要介绍了这些模块之间的分工和彼此间的联系方式，以及与应用程序和网络之间的联系方式。SSH-TRANS 是基础，它提供了原始连接、报文协议、服务器认证以及基本加密和完整性服务。在建立 SSH-TRANS 连接之后，应用程序就有一个到已认证端的单独的、安全的、全双工的字节流。

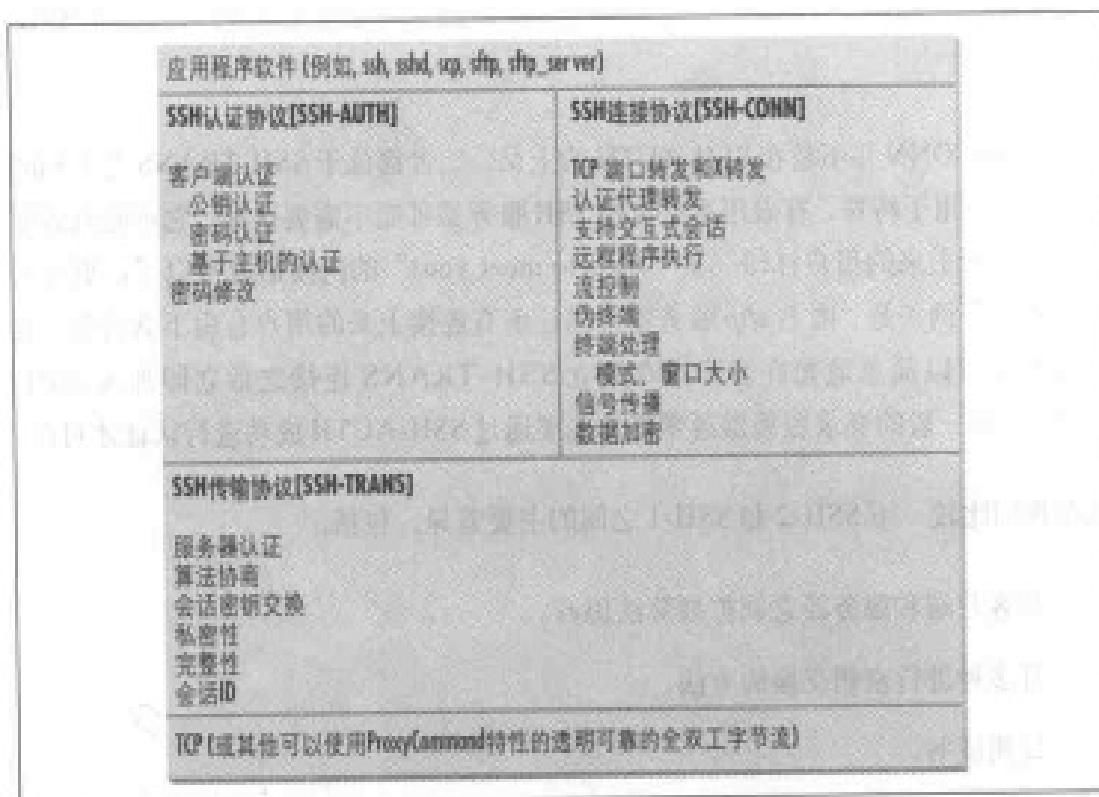


图 3-5: SSH-2 协议族

然后，客户端要使用 SSH-AUTH 在 SSH-TRANS 连接上向服务器认证自己。SSH-AUTH 定义了三种认证方法：公钥认证，基于主机的认证和密码认证。公钥认证和 SSH-1 的“RSA”方法类似，但却更通用，可以使用任何公钥签名算法。标准的公钥认证只需要一个算法 DSA，因为到目前为止 RSA 的使用还受到专利许可的限制。

(注 16)。基于主机的认证和 SSH-1 的 RhostsRSA 方法类似，它使用加密来确信客户端主机的身份，从而提供了可信主机认证。密码认证和 SSH-1 的密码认证相同；它还提供了修改用户密码的功能，但是我们现在还没有看到哪种实现支持这个特性。原来 SSH-1 脆弱的、不安全的 Rhosts 认证已经被丢弃了。

最后，SSH-CONN 协议通过由 SSH-TRANS 提供的单个管道为客户端提供了丰富的服务。这包括要支持多个交互会话或非交互会话所需要的所有内容：经过底层管道的多路数据流（或通道）；管理 X、端口和代理转发；通过连接转发应用程序信号（例如 SIGWINCH，说明终端窗口的大小改变了）；终端处理；数据压缩以及远程程序执行。和 SSH-1 不同，SSH-CONN 可以处理在同一个连接上的零个或多个交互会话。这就是说 SSH-2 不需要一个单独的终端会话就可以支持 X 或端口转发，而 SSH-1 则不行。

注意 SSH-CONN 并不是在 SSH-AUTH 的上层；二者都位于 SSH-TRANS 之上相同的一层中。用于特殊、有限用途的专用 SSH 服务器可能不需要认证。它可能只需要给每个连接上来的用户打印一条“Nice to meet you!”的问候语就可以了。更实际一点的一个例子是，匿名 *sftp* 服务器可以让所有连接上来的用户自由下载内容。这种服务器可以简单地允许客户端在建立 SSH-TRANS 连接之后立即加入 SSH-CONN，而一般的登录服务器通常都首先要通过 SSH-AUTH 成功进行认证才可以。

现在我们比较一下 SSH-2 和 SSH-1 之间的主要差异。包括：

- 在客户端和服务器之间扩展算法协商。
- 有多种进行密钥交换的方法。
- 公钥证书。
- 认证更灵活，包括部分认证。
- 使用加密进行强完整性检测。
- 周期性替换会话密钥（“rekeying”）。

注 16： 经过若干年的专利保护之后，RSA 在 2000 年 9 月开始用于公共领域。

3.5.1.1 算法的选择和协商

SSH-1的一个很好的特性是算法协商，此时客户端从服务器支持的加密算法中选择出一些来。然而，SSH-1中的其他算法的编码不好也不够灵活。SSH-2通过在客户端和服务器之间可以进行其他算法协商对其进行改进：主机密钥、消息认证、散列函数、服务器密钥交换和数据压缩使用的算法都可以协商。SSH-2需要为每一类都支持一种方法以便确保他们的协同工作能力，并定义了其他几种推荐方法和可选方法。^[3.9]

SSH-2的另外一个改进是扩展了算法的命名空间。SSH-1使用一个数码来代表要协商的算法，而没有为其设置值以用于本地的其他用途。SSH-2与此不同，其算法（以及协议、服务和密钥/证书格式）使用字符串进行命名，也支持本地定义。从SSH-ARCH来看：

那些不包含有@符号的名字保留给IANA (Internet Assigned Numbers Authority, Internet号码分配机构)进行分配。例子包括3des-cbc、sha-1、hmac-sha1和zlib。这种格式的名字可以在IANA进行注册，如果还没有在IANA注册就不能使用。已经注册过的名字中一定不能含有@符号或逗号(,)。任何人都可以使用name@domainname格式的名字来定义其他算法，例如，ourcipher-cbc@ssh.fi。@符号之前内容的格式没有详细规定；但是它必须完全由除@或逗号(,)之外的US-ASCII字符组成。@符号之后的内容必须是一个有效的完整的Internet域名[RFC-1034]，该域名由定义者进行控制。每个域名都决定了自己如何管理本地命名空间。

这种格式允许为内部使用添加新的非标准算法，而不会影响它与其他SSH-2实现的协同工作能力，甚至也不会影响其他本地内容。OpenSSH使用这种功能定义了一种称为hmac-ripemd160@openssh.com的完整性检查算法。

3.5.1.2 会话密钥交换和服务器密钥

回想一下会话密钥是大块数据加密算法使用的共享对称密钥——用来直接对SSH连接之上传输的数据进行加密。^[3.3]在SSH-1中，这个密钥是由客户端生成的，然后使用服务器密钥和服务器的主机密钥进行双重加密后安全地传送给服务器。服务器密钥的目的是提供完美的转发安全性。^[3.4.1]

为了和设计保持一致，SSH-2 引入了一种更通用的机制来适应多密钥交换的方法，使用这种方法可以协商要使用的交换方法。所选定的方法会生成一段共享密文，它不会直接用作会话密钥，而是用于后面生成会话密钥的过程。这种附加的处理过程可以确保哪一方都不能完全确定会话密钥（不管使用哪种交换方法），并且对重放攻击提供了保护。^[3.1.2] SSH-2 协议的密钥交换过程还要负责服务器认证，这和 SSH-1 是相同的。

SSH-2 现在只定义了一种密钥交换方法，就是 diffie-hellman-group1-sha1；所有的 SSH-2 实现都必须支持这种方法。顾名思义，它是一个基于固定群的 Diffie-Hellman 密钥协商算法（注 17），采用 SHA-1 hash 函数。Diffie-Hellman 算法自身预先采取了保密措施，因此不需要再使用服务器密钥。而且 Diffie-Hellman 算法和刚才介绍的过程无关，它可以单独保证任何一方都不能确定共享密文。

由于 diffie-hellman-group1-sha1 本身已经预先提供了保密机制，因此 SSH-2 实现使用该算法就不需要服务器密钥了。因为我们还可以为 SSH-2 定义其他密钥交换方法，所以我们可以可信地实现 SSH-1 的密钥交换算法，这需要一个服务器密钥或类似的方法来提供完美的转发安全性。但是现在还没有定义这种方法，因此只有在 SSH1 和 OpenSSH/1 中才需要服务器密钥。因此，只使用 SSH-2 的服务器更应服从 *inetd* 的控制，因为它避免了启动时生成服务器密钥的负载。^[5.4.3.2] 例子有 SSH2，还有禁用了 SSH-1 的 OpenSSH。

3.5.1.3 密钥 / 身份绑定

在任何一种公钥系统中，一个至关重要的问题是需要对密钥所有者进行验证。假设你希望和自己的朋友 Alice 共享密文，但是入侵者 Mallory 欺骗你把他自己的一个公钥当作 Alice 的公钥。现在你（本来想）给 Alice 加密的信息对于 Mallory 来说都是可以读取的。当然，当 Alice 发现自己不能解密你的消息时，你们就会很快发现这个问题，但是到这时损失已经造成了。

密钥属主的问题源于对一种称为公钥证书技术的使用。证书是一个数据结构，它说明一个可信第三方为密钥输出的身份提供担保。更确切地说，该证书证明公钥和特

注 17：群（group）是和 Diffie-Hellman 过程有关的一个数学抽象；如果读者对此感兴趣，请参看群论、数论或抽象代数。

定身份（一个个人或公司名、email 地址、URL 等等）或能力（访问数据库、修改文件、登录账号等权限）之间的绑定关系。这种证明使用一个第三方的数字签名来表示。因此在我们的例子中，你和 Alice 可以协商使用第三方 Pete 来对你们各自的公钥进行签名，因此你就不会相信 Mallory 那个伪造的未签名的密钥了。

这种方法当然很好，但是谁来为担保者提供担保呢？你又怎样才能知道对 Alice 的密钥进行签名的人真的是 Pete 呢？这个问题还可以继续递归，因为你需要签名者的公钥，还需要密钥所使用的证书，依此类推。这种证书链的问题可以划分成一种层次化模型，其根是知名的证书权威 (Certificate Authorities)；也可以将其划分成一种分布式网络模型，就是所谓使用 PGP 的“可信网络 (web of trust)”。这种划分或信任模型是公钥基础体系 (public-key infrastructure, PKI) 的根本。

在 SSH 中，密钥属主的问题在对主机名和主机密钥进行绑定时也开始浮出水面。在现在所有的 SSH 实现中，这个问题是使用主机名、地址和密钥的简单数据库来解决的，该数据库必须由用户或系统管理员对其进行维护并负责分布。这种系统的伸缩性很差。SSH-2 允许在公钥中包括证书，这样就打开了使用 PKI 技术的大门。现在 SSH-2 的规范定义了 X.509、SPKI 和 OpenSSH 证书的格式，但是现在的 SSH 实现还不支持使用这些证书。

从理论上来说，证书也可以用于用户认证。例如，证书可以把用户名和公钥绑定在一起，而且 SSH 服务器可以接受有效的证书为私钥持有者进行认证，从而允许访问账号。这种系统提供了基于主机的认证的优点，而避开了单纯对主机安全性的脆弱依赖关系。如果 PKI 以后更通用了，那么就可能出现这种特性。

3.5.1.4 认证

为了进行认证，SSH-1 客户端就要一一尝试从服务器所允许的认证方法集合中所选定的一系列认证方法（公钥认证、密码认证、可信主机认证等等），直到有一种方法成功或所有方法都失败为止。这种方法是一种要么全有要么全无 (all-or-nothing) 的尝试；服务器无法要求使用多种认证方法，因为一旦一个方法成功之后，认证过程就结束了。

SSH-2 协议更加灵活：服务器在交换过程的任何时机都可以通知客户端可用的认证方法，而不是和 SSH-1 一样只能在连接开始时马上确定认证方法。因此，SSH-2 服

务器可以在两次尝试失败之后决定禁用公钥认证，以后只允许使用密码认证方法。这种功能的一个用途十分有趣，值得我们注意。SSH-2客户端通常首先使用一个特殊的方法“none”发起一个认证请求。这通常都会失败并返回服务器所允许的实际认证方法。如果你在浏览SSH日志时看到方法“none”已经“failed(失败)”而迷惑不解，那么现在你就知道为什么会这样了（这是正常的）。

SSH-2服务器还可以说明部分成功的情况：也就是一种特定的方法已经成功了、但是还需要进行进一步的认证。因此服务器就可能需要客户端登录时要通过多种认证测试，也就是说既需要密码认证，也需要基于主机的认证。SSH2服务器配置关键字 RequiredAuthentications 可以控制这个特性，OpenSSH/2现在还不支持这种特性。[5.5.1]

3.5.1.5 完整性检查

SSH-2对SSH-1使用CRC-32这种脆弱的完整性检测方法进行了改进，它使用强加密的消息认证代码（Message Authentication Code, MAC）算法来保证完整性，并提供数据源的担保。每个方向上使用的密钥（它和会话加密密钥是分离的）和MAC方法是在协议的密钥交换阶段确定的。SSH-2定义了几个MAC算法，需要对hmac-sha1的支持，hmac-sha1是一个160位的散列值，它使用SHA-1算法构造标准密钥HMAC。（请参看RFC-2104，“HMAC: Keyed-Hashing for Message Authentication”）。

3.5.1.6 基于主机的认证

一个SSH服务器需要一些客户端主机标识符来执行基于主机的认证。具体说来，需要执行两步操作：

- 查找客户端主机密钥。
- 在通过基于主机的控制文件(*hosts.equiv*, 等等)进行认证时匹配客户端主机。

我们称这些操作为HAUTH过程。现在，在协议1和协议2中的可信主机认证之间有一个重要的区别：在SSH-2中，认证请求包含客户端的主机名，而在SSH-1中则不需要。这就是说SSH-1要被强迫使用客户端的IP地址，或通过名字服务获得的一个名字作为标识符。由于SSH-1服务器有关客户端主机身份的思想是绑定客户端的

网络地址，因此 RhostsRSA 认证在以下这些通常的情况下不能正常工作（至少有时不行）：

- 移动客户端，IP 地址会经常修改（例如，随身携带并连接到不同网络上的掌上电脑）。
- 客户端位于网络代理（例如，SOCKS）之后。
- 客户端具有多个网络地址的客户端，除非相应的 DNS 项有特殊的排列顺序。

另一方面，SSH-2 协议并不会利用这种限制：基于主机的认证过程原则上是和客户端的网络地址是独立的。SSH-2 服务器的客户端标识符有两种选择： N_{auth} 和 N_{net} ，前者是认证请求中的名字，后者是通过客户端的网络地址查找到的名字。它也可以简单地忽略 N_{net} ，在 HAUTH 中使用 N_{auth} 。当然，已知名主机列表和基于主机的认证文件必须使用命名空间进行维护。实际上， N_{auth} 可以从任何标识符空间中选取，而根本不必局限于网络名字服务。为了清楚起见，应该可以继续使用客户端的规范主机名。

正如目前的实现一样，SSH2 不能这样处理。*sshd2* 的操作和 *sshd1* 相同，它们都在 HAUTH 中使用 N_{net} ， N_{auth} 只是用于完整性检查。如果 $N_{net} \neq N_{auth}$ ，那么 *sshd2* 就认为该次认证失败。这是真正向后兼容的，从而减小了基于主机的认证的用途，因为它不能在前面提到的那种情况下工作。作者建议 SCS 在 HAUTH 中使用 N_{auth} ，并在每个主机选项中都实现 $N_{net} = N_{auth}$ 检查。这在知道客户端主机地址不会变化的情况下可以增加一点安全性。这和公钥认证类似，后者和客户端主机地址也是独立的，但是它允许在适当的时候（使用 "hosts=" *authorized_keys* 选项）增加一些基于源地址的限制。

3.5.1.7 会话密钥重新生成 (rekeying)

使用一个密钥加密的数据越多，就越可能进行分析，攻击者解密的机会也就越大。因此如果正在对大量数据进行加密，周期性地修改密钥是个英明的决定。这并不是非对称密钥的问题，因为非对称密钥通常都只用于加密少量数据，对散列值进行数字签名或加密对称密钥。然而，例如，如果正在传输大文件或进行保护性备份，SSH 连接中大量数据加密所使用的密钥可能会对几百兆数据进行加密。SSH-2 协议为 SSH 连接的每一端替换会话密钥都提供了一种方法。这就导致客户端和服务器要对

新的会话密钥进行协商并在以后使用。SSH-1并没有为会话提供修改大量密文的密钥的方法。

3.5.1.8 SSH-1/SSH-2：小结

表 3-3 对 1.5 版本和 2.0 版本的 SSH 协议之间的重要区别进行了总结。

表 3-3: SSH-1 和 SSH-2 的区别

SSH-2	SSH-1
传输、认证和连接协议彼此独立	一个集成协议
强加密完整性检测	弱 CRC-32 完整性检测
支持密码修改	N/A
每个连接可以有任意个会话通道（也可以没有）	每个连接只能刚好有一个会话通道 (需要执行远程命令，即使不想执行也不行)
对模块加密和压缩算法都要进行协商，包括大块数据加密、MAC 和公钥	只对大块密码进行协商；其他内容都是既定的
连接的每个方向上的加密、MAC 和压缩都要使用独自的密钥单独进行协商	连接的两个方向上都使用相同的算法和密钥（但是 RC4 使用独自的密钥，因为该算法的设计要求不能重用密钥）
算法 / 协议命名模式的可扩展性允许在确保协同工作能力的同时进行本地扩充	既定的编码禁止使用协同操作的内容
用户认证方法： • 公钥认证 (DSA、RSA、OpenPGP) • 基于主机的认证 • 密码认证 • (Rhosts 由于不安全而被舍弃)	支持的用户认证方法更多： • 公钥认证（只能使用 RSA） • RhostsRSA • 密码认证 • Rhosts (rsh 风格) • TIS • Kerberos
由于使用 Diffie-Hellman 密钥协定，不需要再使用服务器密钥	在会话密钥上使用增加转发保密的服务器密钥
支持公钥证书	N/A

表 3-3: SSH-1 和 SSH-2 的区别 (续)

SSH-2	SSH-1
用户认证交换更灵活，允许访问时请求使用多种认证方法	每个会话只允许使用一种认证方法
基于主机的认证原则上是和客户端的网络地址无关的，因此可以使用代理、移动客户端等等，请参看 [3.5.1.6]	Rhost RSA 认证和客户端的主机地址紧密地绑定在一起，这样就限制了它的用法
周期性替换会话密钥	N/A

3.5.2 实现差异

目前 SSH-1 和 SSH-2 的实现产品中有很多差异。有些差异是由于协议不同而直接造成的，例如能否请求多种认证方法或能否支持 DSA 公钥算法；还有些是和协议无关的特性差异。这些都是由于软件作者的设计不同而造成的。现在我们讨论一下 OpenSSH、SSH1 和 SSH2 之间和协议无关的一些设计差异和特性差异：

- 主机密钥。
- 认证失败时不倒退回 *rsh*。
- setuid 客户端。
- SSH-1 的向后兼容性。

3.5.2.1 主机密钥

SSH 主机密钥是一种长期非对称密钥，用来区分并标识运行 SSH 的主机或一台 SSH 服务器上的多个 SSH 实例，这要根据 SSH 的实现而确定。在 SSH 协议中，有两个地方会用到主机密钥：

1. 服务器认证向连接上来的客户端验证服务器的身份。每个 SSH 连接都会执行这个步骤（注 18）。

注 18： 在 SSH-1 中，主机密钥也会对传输使用的会话密钥进行加密。但是，这种用法实际上是用于服务器认证，而不是用来保护每次传输的数据；然后服务器就通过显示成功解密了会话密钥而证明自己的身份。会话密钥的保护是通过使用服务器密钥两次加密实现的。

2. 客户端主机对服务器的认证；只在RhostsRSA或基于主机的用户认证中使用。

不幸的是，“主机密钥”这个术语很容易引起混淆。它似乎是说只有一个这种密钥可以属于一台给定的主机。这对于客户端认证来说是对的，但是对于服务器认证来说则不然，因为在一台机器上可以运行多个SSH服务器，每个SSH服务器都用一个不同的密钥来标识。这种所谓的“主机密钥”实际上是标识SSH服务器程序正在运行的一个实例，而不是一台机器。

SSH1维护了一个既用于服务器认证又用于客户端认证的数据库。这个数据库就是系统的`known_hosts`文件（`/etc/ssh_known_hosts`）和用户的`~/.ssh/known_hosts`文件的一种综合，后者可以位于源机器上（对于服务器认证），也可以位于目标机器上（对于客户端认证）。该数据库把主机名或地址映射成一组密钥，用于对该主机名或地址的机器进行认证（注19）。一个主机名可以关联多个密钥（稍后我们就会更详细介绍这一点）。

另一方面，SSH2则不同，它为此目的维护了两个单独的映射：

- 服务器主机认证使用的`hostkeys`映射。
- 客户端主机认证使用的`knownhosts`映射。

哇！这些术语更让人摸不着头脑了。实际上，我们在这里套用一个格式稍有不同的术语“已知名主机”（`knownhosts`和`known_hosts`相比较而言），但是目的却不同。

SSH1把主机密钥当成几项记录，保存在一个文件中，而SSH2将其存储在一个文件系统目录中，每个密钥使用一个文件，由文件名进行索引。例如，一个`knownhosts`目录如下所示：

```
$ ls -l /etc/ssh2/knownhosts/
total 2
-r--r--r-- 1 root      root      697 Jun  5 22:22 wynken.sleepy.net.ssh-dss.pub
-r--r--r-- 1 root      root      697 Jul 21 1999 blynken.sleepy.net.ssh-dss.pub
```

注意文件名的格式为`<hostname>.<key type>.pub`。

另外一个映射`hostkeys`不但要对主机名/地址进行映射，而且还要对服务器监听的

注19：假设服务器彼此兼容，如果希望让它们共享相同的密钥，也是可以的。

TCP端口号进行映射；也就是说，它要对TCP socket进行映射。这允许每个主机都有多个密钥，方式比前面的更特殊。此时，文件名的格式变成了`key_<port number>_<hostname>.pub`。下面的例子给出了一些公钥，其中一个SSH服务器正在`wynken`的22端口上运行，而另外两个在`blynken`的22端口和220端口上运行。另外，我们又建立了一个符号链接，给`wynken:22`上的服务器起了一个别名：“`nod`”。终端用户可以把这些密钥（可以手工添加，也可以由客户端自动添加）加入`~/.ssh2/knownhosts`和`~/.ssh2/hostkeys`目录，从而添加这些映射。

```
$ ls -l /etc/ssh2/hostkeys/
total 5
-rw-r--r-- 1 root      root      757 May 31 14:52 key_22_blynken.sleepy.net.pub
-rw-r--r-- 1 root      root      743 May 31 14:52 key_22_wynken.sleepy.net.pub
-rw-r--r-- 1 root      root      755 May 31 14:52 key_220_wynken.sleepy.net.pub
lrwxrwxrwx 1 root      root      28 May 31 14:57 key_22_nod.pub ->
                           key_22_wynken.sleepy.net.pub
```

虽然SSH2允许每个主机使用多个密钥（multiple keys per host），但是它仍然丢失了SSH1的一个很有用的特性：每个主机名使用多个密钥（multiple keys per name）。二者听起来似乎是一回事，但是它们有一点细微的差别：主机名可以指多个主机。一个常见的例子是一个主机名后面可能有一组负载共享的登录服务器。一个大学可能有这样三台机器来处理普通的登录访问，每个都有自己的主机名和地址：

```
login1.foo.edu → 10.0.0.1
login2.foo.edu → 10.0.0.2
login3.foo.edu → 10.0.0.3
```

另外，还有一个通用的主机名包括这三个地址：

```
login.foo.edu → {10.0.0.1, 10.0.0.2, 10.0.0.3}
```

该大学的计算中心告诉用户只要登录`login.foo.edu`即可，其域名服务会轮询（round-robin）处理这三个地址（例如，使用轮询DNS）以便在这三台机器之间共享负载。SSH的缺省设置在此时就会碰到问题。每次连接到`login.foo.edu`上时，正连接的机器就有2/3的可能和上次登录的机器不同，其主机密钥也不同。SSH会反复抱怨`login.foo.com`的主机密钥已经改变了，并产生一条警告说有人正在攻击客户端。这个问题很快就会让人头大。使用SSH1，你可以编辑`known_hosts`文件为通用主机名关联所有的主机密钥，可以把：

```
login1.foo.edu 1024 35 1519086808544755383...
login2.foo.edu 1024 35 1508058310547044394...
login3.foo.edu 1024 35 1087309429906462914...
```

修改成：

```
login1.foo.edu,login.foo.edu 1024 35 1519086808544755383...
login2.foo.edu,login.foo.edu 1024 35 1508058310547044394...
login3.foo.edu,login.foo.edu 1024 35 1087309429906462914...
```

但是，使用 SSH2 则没有什么通用的方法可以很好地实现这个功能；因为数据库是由目录中的项进行索引的，每个文件有一个密钥，所以每个主机名不能有多个密钥。

似乎这样处理会丢失一些安全性，但是我们并不这样认为。实际发生的事情是对于一个特定主机名的解析在不同的时间可能会指向不同的主机，这样如果一个连往该主机名的连接通过了一个给定的密钥的认证，那么就可以告诉 SSH 信任这个连接。大多数情况下，这组密钥的个数都是 1，也就可以告诉 SSH，“当我连接到这个主机名时，我想确保我正连到这个特定主机上。”为一个主机名使用多个密钥，也可以这样说，“当我连接到这个主机名上时，我想确保我可以连到下面这组主机上。”这样做是件十分有效而且有用的事情。

解决这个问题的另外一种方法是让 *login.foo.com* 的系统管理员在这三台机器上都安装相同的主机密钥。但是这样就影响了 SSH 在这些主机上分布的能力，即使希望这样处理也是如此。我们推荐使用前一种方法。

3.5.2.2 认证失败时不倒退回 rsh

SSH1 不但可以支持 *rsh* 风格的认证，而且如果远程主机现在没有运行 SSH 服务器，*ssh* 还可以自动调用 *rsh*。由于 SSH2 可以支持 Rhosts 认证，这种特性就从 SSH2 中取消了，这主要是因为 *rsh* 的安全性太差。[\[7.4.5.8\]](#)

3.5.2.3 setuid 客户端

为了使用 RhostsRSA 认证，SSH1 客户端需要被安装成 setuid root。之所以这样处理有两个原因：主机密钥访问和特权源端口。对于客户端特权端口的限制源于 *rsh* 风格认证，并没有增加 RhostsRSA 的安全性，在 SSH2 基于主机的认证中已经舍弃了这种要求。[\[3.4.2.3\]](#)

设置客户端 setuid 的另外一个原因是对私有主机密钥文件的访问权限。因为主机密钥是加密后存储的，所以 SSH 不需要用户输入口令就可以访问主机密钥。因此，我们必须保护包含私有主机密钥的文件，不能让普通用户可以读取该文件。SSH 服务器由于其他一些原因通常都是作为 root 运行的，因此可以读取任何文件。但是客户端是作为普通用户运行的，因此必须具有访问私有主机密钥的权限，这样才能进行可信主机认证。该文件通常都被安装成只能由 root 用户读取，因此客户端必须被 setuid 成 root。

现在在基本安全的环境中，只要可能，我们就应该避免安装 setuid 的程序——特别是那些 setuid 成 root 的程序。任何这种程序都必须仔细编写以防误用。更确切地说，setuid 的程序应该尽可能短小、简单，尽量少和用户交互。像 SSH 客户端这种大型的复杂程序，经常会和用户进行通信，也经常会和其他机器通信，这样使用的确不安全。

SSH2 通过引入 *ssh-signer2* 程序避开了这个问题。*ssh-signer2* 为那些需要访问私有主机密钥的客户端单独使用一个程序。该程序按照自己的标准输入/输出和 SSH 报文协议通信，并用作要签名的基于主机的认证请求的输入。它会对请求仔细检查有效性；最通常的情况是，它要检查请求中的用户名是否是运行 *ssh-signer2* 的用户，以及主机名是否是当前主机的规范名。如果请求有效，*ssh-signer2* 就使用主机密钥对请求进行签名并返回。

由于 *ssh-signer2* 很小而且非常简单，因此比较容易确定该程序编写是否安全，是否可以安全执行 setuid。这样，SSH 客户端本身就不需要再 setuid 了；当它需要对一个基于主机的认证请求进行签名时，就把 *ssh-signer2* 当成一个子进程运行来获得签名。

SSH2 的安装过程虽然使得私有主机密钥只能被 root 读取，但却把 *ssh-signer2* setuid 成 root，因此这样使用时就不再真正需要使用 root 账号了，实际上出于任何原因考虑都不需要了。这样足以创建一个新的、非特权用户用于这种特殊目的，比如说“*ssh*”。该用户应该是一个锁定账号，没有密码，无法登录，账号信息应该存储在本地文件中（例如，*/etc/passwd*, */etc/group*）而不是 NIS 中。然后就应该把主机密钥修改成只能由这个账号读取，并调用 *ssh-signer2* 执行 setuid，并修改其所有权限。例如：

```
# chown ssh /etc/ssh_host_key
# chmod 400 /etc/ssh_host_key
# chown ssh /usr/local/bin/ssh-signer2
# chmod 04711 /usr/local/bin/ssh-signer2
```

这样的效果和缺省的安装相同，而且风险更小，因为它不会调用一个 setuid 成 root 的程序。

此后可以达到和 *ssh1* 一样的效果，但是不能再使用可信主机认证了，因为服务器需要特权源端口才能采用 *RhostsRSA* 机制。

3.5.2.4 SSH-1 向后兼容性

如果在同一台机器中也安装了完整的 SSH1 包，那么 SSH2 还为 SSH-1 协议提供了向后兼容性。当用户连接到旧版本的协议上来时，SSH2 客户端和服务器就简单运行自己的 SSH1 副本。这可相当讨厌。这样既浪费，速度又慢，因为每个 *sshd1* 都需要生成自己的服务器密钥，如果不使用 SSH1，只需要一个主服务器每小时重新生成一次。这样会对系统的可用信息量（或称为熵，entropy）造成浪费，有时会浪费系统宝贵的资源，可能在启动 SSH-1 连接到 SSH2 服务器上时会造成显著的延时。而且，这在管理上也是件头疼的问题，还有安全性问题，因为用户必须维护两个单独的 SSH 服务器配置，并要试图确保所有期望的限制管用，并且在两台服务器上都完全设置好。

从 2.1.0 开始，OpenSSH 用一组程序同时支持 SSH-1 和 SSH-2，虽然这些支持还不像 SSH2 中那样完整。（例如，不支持基于主机的认证；但是这不会影响到 SSH-2 的兼容性，因为这种支持本身就是可选的。）这种技术避免了在 SSH2 机制中所继承下来的问题。SSH-1 协议仍然被认为是用户的首选；如果你正和一个支持这两种协议的服务器进行联系，那么 OpenSSH 客户端就使用 SSH-1。你可以使用 -2 开关或“protocol 2”配置语句强制 OpenSSH 使用 SSH-2。

3.6 伪装用户访问权限

SSH 服务器通常都是作为 root 用户运行的（在某些情况中客户端也是如此）。SSH 在很多地方都会需要访问源账号和目标账号的文件。root 账号特权会覆盖大部分访问控制权限，但并不是所有的都可以。例如，NFS 客户端上的 root 账号并不需要对

远程文件系统中的文件具有特殊的访问权限。另外一个问题是 POSIX 访问控制列表 (ACL)；只有文件的属主才能修改文件的 ACL，root 不能覆盖这种限制。

在 Unix 中，有一种方法可以让一个进程利用另外一个用户的身份，而不是使用当前用户的 ID：setuid 系统调用。root 用户可以使用这种机制“变成”任何用户。然而，这个调用在进程执行过程中是不可取消的；一个程序不能恢复到原来的权限并在 SSH 中取消 setuid。有些 Unix 实现中有一个可以取消的 setuid（设置有效用户的 ID），但是这并不通用，也不符合 POSIX 标准（注 20）。

为了保证可移植性，SSH1 和 SSH2 使用这种广泛可用的 setuid 系统调用。它们第一次需要作为普通用户访问一个文件时，就开始执行一个子进程。这个子进程调用 setuid 切换（不可取消）成所希望的 uid，但是 SSH 主程序继续作为 root 运行。然后，当 SSH 需要作为其他用户访问文件时，主程序就向这个子进程发送一个消息，要求它执行所需要的操作并返回结果。这种技术在内部称为 *userinfo* 模块。

在使用 SunOS 中的 *trace*、Solaris 中的 *truss*、Linux 中的 *strace* 以及其他进程调试器调试 SSH 进程时要记得这一点。缺省情况下，这些程序都只跟踪最顶层的进程，因此要永远记得还要跟踪子进程。（请参看调试器的手册页中的适当选项，通常是 *-f*。）如果忘记这样做了，那么对文件进行访问时就会出现问题，你可能不会看到问题所在，因为 userinfo 子进程执行了文件访问系统调用（*open*、*read*、*write*、*stat* 等等）。

3.7 随机数

加密算法和协议都需要很好的随机位，或称为熵（entropy，平均信息量）。随机数有很多用途：

- 生成数据加密使用的密钥。
- 作为正文填充位，在加密算法中初始化向量，这有助于增加密码分析的难度。
- 用于协议交换中的检测字节或 cookie，将其作为防止报文欺骗攻击的一种措施。

注 20：实际上，POSIX 的确有相同的特性，不过名字不同，而且不是所有的地方都会出现。

随机数的获得可能比想像的困难；实际上，就算是对随机数给出一个定义也是很困难的（或者给一种特定的情况选定一个确切的定义也很难）。例如，在统计模型中使用得非常好的“随机”数可能对于密码学来说就远远不够了。这些应用程序每个都需要其输入具有相当程度的随机性，例如满足一个均匀分布（even distribution）。特别是密码学需要不可预言，这样攻击者即使读取了数据也无法猜测出密钥。

纯随机数（从完全不可预言角度来讲）是不能由计算机程序产生的。任何程序输出所产生的位序列最终都会重复。对于纯随机数来说，只能使用一些物理过程，例如湍流或放射性物质衰变产生的量子。即使在这些情况中，也必须注意测量措施不能引入一些不必要的结构。

但是，有些算法可以产生实用的不可预言的长序列输出，该序列具有很好的统计学随机特性。这些序列对于很多密码学应用程序来说也是足够的，这种算法称为伪随机数生成器（pseudo-random number generator），或 PRNG。PRNG 需要少量的随机输入，称为随机数种子，这样它就不会总产生相同的输出。使用随机数种子，PRNG 可以产生一个大字符串，这是一个可以接受的随机数输出；实际上，这可以说是一个随机数的“延伸器”。因此使用 PRNG 的程序需要获得一些很好的随机位，只要很少的位就可以，但是这些随机位也最好是不可预言的。

由于很多程序都需要随机位，因此有些操作系统就内建了一些程序来产生随机位。有些 Unix 变种（包括 Linux 和 OpenBSD）都有一个设备驱动程序（通过 /dev/random 和 /dev/urandom 进行访问），在它被当作文件打开或读取时可以提供随机位。这些位源自于各种方法，其中有些方法相当精巧。例如，对磁盘访问时间进行正确地测量过滤可以反映出由于磁头周围的空气湍流所造成的波动。另外一种技术是查看来自于未用的麦克风端口的噪声的最低位。当然，也可以跟踪诸如网络报文到达时间、键盘事件、中断等不确定事件。

虽然 SSH 实现利用了这些随机性，但是这个过程对于终端用户来说是不可见的。下面我们介绍一下内部到底发生了什么。例如，SSH1 和 SSH2 都使用一个基于内核的随机源（如果有这种随机源），同时使用它们不确定系统参数样本，后者是通过运行诸如 ps 或 netstat 之类的程序得到的。SSH 使用这些来撒播 PRNG 随机数种子，由此一石激起千层浪，每次都获得更随机的随机数。由于获得随机数的代价很高，SSH 在两次调用该程序的间隙把随机位缓冲池存储在文件中，如下表所示：

	SSH1	SSH2
服务器	<code>/etc/ssh_random_seed</code>	<code>/etc/ssh2/random_seed</code>
客户端	<code>~/.ssh/random_seed</code>	<code>~/.ssh2/random_seed</code>

我们应该对这些文件进行保护，因为这些文件中包含有敏感的信息，如果泄漏给攻击者会减弱SSH的安全性，而SSH也采取了一些措施来降低这种可能性。随机数种子的信息通常都是和一些尚未使用的内容一起保存，缓冲池只有一半保存到了磁盘上，这样即使被窃取了也可以减小它的可预言性。

在SSH1和SSH2中，所有这些操作都是自动执行的，对于终端用户来说都是不可见的。在没有`/dev/random`的平台上编译OpenSSH时，要作出一个选择。如果已经安装了一个附加(add-on)随机源，例如OpenSSH推荐的“Entropy Gathering Daemon”(EGD，<http://www.lothar.com/tech/crypto/>)，那么就可以告诉OpenSSH使用EGD(通过指定`--with-egd-pool`开关)。如果没有指定缓冲池，那么OpenSSH就使用内部的熵搜集机制。可以通过编辑`/etc/ssh_prng_cmds`文件自行定制运行哪些程序来搜集熵，以及这些程序要达到的随机性究竟有多少。还有，注意OpenSSH的随机数种子是保存在`~/.ssh/prng_seed`文件中的，而不是后台进程中，后者只是root用户的随机数种子文件。

3.8 SSH 和文件传输 (scp 和 sftp)

要理解有关SSH和文件传输，首先要明白SSH并不执行文件传输。

现在我们已经引起了你的注意，但是我们这样做是想干什么呢？毕竟本书有完整的章节专门介绍如何使用`scp1`、`scp2`和`sftp`传输文件。我们的目的是：在SSH协议中没有任何传输文件的内容，SSH通信者不能请求对方通过SSH协议来发送或接收文件。我们刚才提到的程序实际上并没有自己真正实现SSH协议，也根本没有融合什么安全特性。实际上，它们只是在一个子进程中运行SSH客户端，从而连接到远程主机上并在此执行另外一半文件传输过程。SSH对这些程序并没有什么特别的地方；它们使用SSH的方式和我们前面介绍的其他程序非常类似，例如，CVS和Pine。

它必须和`scp1`一起首先出现的惟一一个原因是：没有广泛被使用的、通用的文件传输协议可以在由SSH远程程序执行所提供的一个全双工字节流连接上执行。如果现

有的FTP实现可以方便地在SSH上操作，那么`ssh`就不需要其他内容了；但是正如我们看到的一样，FTP完全不适合这样操作。^[11.2]因此Tatu Ylönen就编写了`scp1`，并将其作为SSH1的一个部分。它使用的协议（让我们称之为“SCP1”）仍未有完整的文档，即使到Tatu Ylönen编写第一份RFC来描述SSH-1协议时仍是如此。

后来，当SSH Communications Security开始编写SSH2时，他们想继续在SSH2中包含一个文件传输工具。他们选定了SSH上的这种模型，但是决定全部重新实现。这样，他们就把“`scp1`协议”替换成“SFTP协议”，这一点已经众所周知了。SFTP协议也是一种在单个可靠全双工字节流连接上执行双向文件传输的简单方法。其基础报文协议恰巧和SSH连接协议的底层相同，大概是为了方便起见。实现者早已有一个工具可以通过字节管道发送面向记录的消息，因此就重用了这个工具。SFTP在发表时也没有文档介绍所用的协议，现在IRTF SECSH工作组正在开展一些工作，来记录SFTP并对其进行标准化。

SFTP这个名字实在是不怎么好，因为它会误导很多人。大部分人都认为SFTP代表“安全的FTP（Secure FTP）”。但是实际上，正如`scp1`一样，首先作为一种协议来说它根本就不是安全的；实现者通过把该协议的通信架构在一个SSH连接之上而获得安全性。其次，它怎样说都和FTP协议没什么关系。认为可以使用SFTP安全地和FTP服务器安全地通信也是一个普遍的错误——如果我们从名字来推测意思，经常都会得到这样的结论。

SSH2中文件传输的另外一点混淆之处是`scp2`、`sftp`以及协议之间的关系。在SSH1中，有一种单独的文件传输协议SCP1和一个单独的程序`scp1`来实现它。在SSH2中，也有一个单独的新文件传输协议：SFTP；但是却有三个单独的程序来实现它，而且有两个不同的客户端。服务器端是`sftp-server`程序。两个客户端是`scp2`和`sftp`。`scp2`和`sftp`只是用途相同的两种不同的外表而已：它们都在子进程中运行SSH2客户端开始执行并和远程主机上的`sftp-server`进行通信。它们仅仅提供了不同的用户接口而已：`scp2`类似于传统的`rcp`，而`sftp`更像是FTP客户端。

即使在安装SSH1和SSH2时可以使用符号链接，可以使用“`scp`”、“`sftp`”这样的通用名来代替“`scp1`”和“`ssh2`”，这也不能简化这些易混淆的术语的使用。当我们提及这两种SSH有关文件传输的协议时，我们称之为SCP1和SFTP协议。SCP1有时只简单称为“`scp`”协议，这在技术上有些含糊，但是通常还可以理解。我们假设你

倾向于把SFTP叫做“scp2协议”，但是我们还没有听过这种说法，如果想确保无误，我们也不建议这样使用（注 21）。

3.8.1 scp1 细节

当运行 *scp1* 把一个文件从客户端拷贝到服务器上时，它会这样调用 *ssh1*：

```
ssh -x -a -o "FallBackToRsh no" -o "ClearAllForwardings yes" server-host scp ...
```

这会在远程主机上运行另外一个 *scp* 的拷贝。该拷贝是使用没有在文档中给出说明的 *-t* 和 *-f* 开关（分别对应“to”和“from”）调用的，它会转入 SCP1 服务器模式。下面这个表给出了一些例子；图 3-6 给出了具体细节。

客户端的 <i>scp</i> 命令：	运行这个远程命令：
<i>scp foo server:bar</i>	<i>scp -t bar</i>
<i>scp server:bar foo</i>	<i>scp -f bar</i>
<i>scp *.txt server:dir</i>	<i>scp -d -t dir</i>

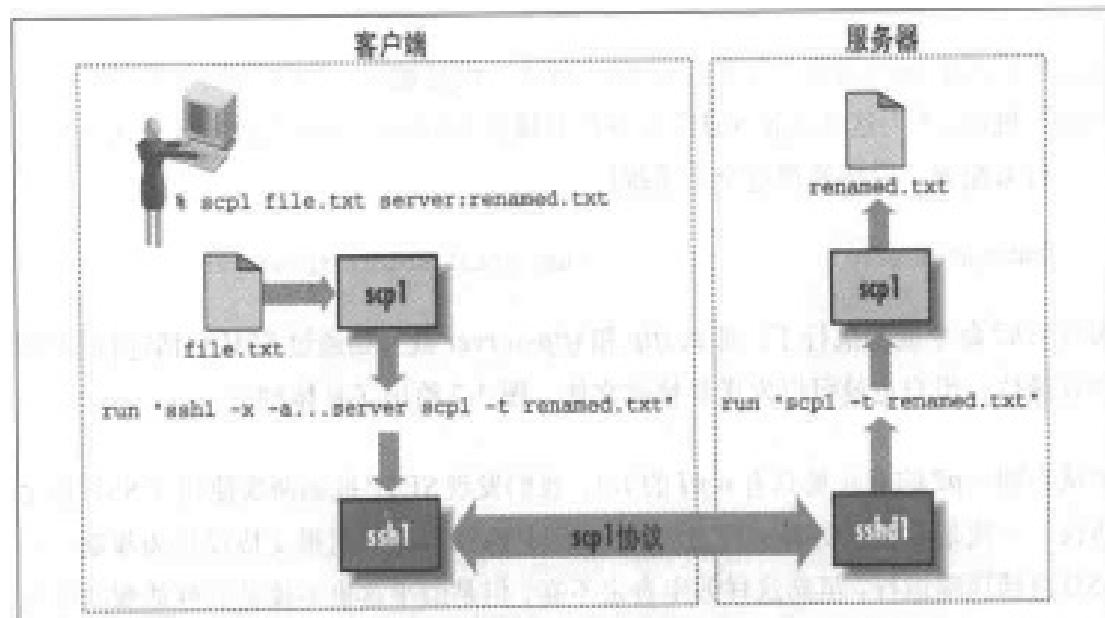


图 3-6: *scp1* 的操作

注 21：特别是由于 *scp2* 可以运行 *scp1*，这样可以与 *SSH1* 兼容！

如果运行 *scp1* 在两台远程主机之间拷贝一个文件，那么它会在源主机上简单地执行另外一个 *scp1* 客户端把该文件拷贝到目的主机上。例如这个命令：

```
scp1 source:music.au target:playme
```

在后台运行：

```
ssh1 -x -a ... as above ... source scp1 music.au target:playme
```

3.8.2 *scp/sftp* 细节

当运行 *scp2* 或 *sftp* 时，这两个程序会使用下面的命令在后台运行 *ssh2*：

```
ssh2 -x -a -o passwordprompt "%U@%H's password:"  
      -o "nodelay yes"  
      -o "authenticationnotify yes"  
      server host  
      -s sftp
```

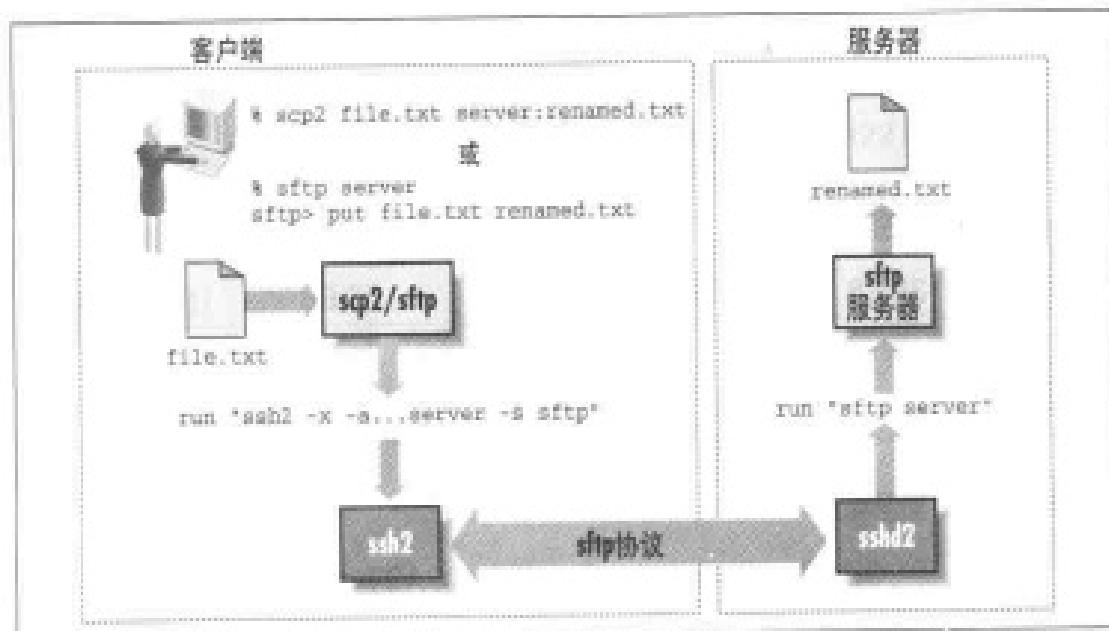
和 *scp1* 不同，这里的命令不会根据文件传输的方向和类型而变化；所有必要的信息都包含在 SFTP 协议内部。

注意它们不会使用远程命令来启动 *sftp-server*，而是通过 *-s sftp* 选项采用 SSH2 “子系统”机制。^[5.7]这就是说 SSH2 服务器必须在 */etc/ssh2/sshd2_config* 使用下面这样一行来配置，以便处理这个子系统：

```
subsystem-sftp           /usr/local/sbin/sftp-server
```

假设 *ssh2* 命令成功执行了，那么 *sftp* 和 *sftp-server* 就开始通过 SSH 会话进行 SFTP 协议通信，用户也就可以发送并接收文件。图 3-7 给出了具体细节。

测试表明 *scp2* 的吞吐量只有 *scp1* 的 1/4。我们发现 SFTP 机制两次使用了 SSH 报文协议，一次是封装在另外一次之内的：SFTP 协议本身使用报文协议作为基础，在 SSH 会话顶端运行。虽然这样效率肯定不高，但是似乎这也不像是其性能骤减的原因；可能是还存在一些可以修正的简单实现问题，例如在协议代码的不同层之间进行缓冲时交互处理不好。我们还没有自己深入研究代码来分析速度下降的原因。

图 3-7: `scp2/sftp` 操作

3.9 SSH 使用的算法

表 3-4 至表 3-6 对 SSH 协议机器实现中可以使用的加密算法进行了总结。必须使用的算法以黑体给出；推荐使用的算法以斜体给出；其他算法都是可选的。圆括号说明该算法不是在协议中定义的，而是由某些实现所提供的。这些项的含义为：

- x* 实现支持该算法，缺省选项中包含了该算法。
- a* 实现支持该算法，但是缺省选项中不包含该算法（该算法必须在编译时专门启用）。
- 实现不支持该算法。

表 3-4: SSH 协议中的算法

	SSH-1.5	SSH-2.0
公钥	RSA	DSA , DH
散列函数	MD5 , CRC-32	SHA-1 , MD5
对称密钥	3DES , <i>IDEA</i> , <i>ARCFOUR</i> , DES	3DES , <i>Blowfish</i> , <i>Twofish</i> , CAST-128, IDEA, ARCFOUR
压缩	zlib	zlib

注意，表 3-4 简单地给出了两种协议规范中所使用的不同种类的算法，但是没有给出用途。例如，SSH1 既使用 MD5，又使用 CRC-32，但是二者的用途不同；这个表中并没有说明 SSH-1 可以使用 MD5 算法进行完整性检测。

表 3-5：SSH-1 加密算法

	3DES	IDEA	RC4	DES	(Blowfish)
SSH1	x	x	o	o	x
OpenSSH	x	-	-	-	x

表 3-6：SSH-2 加密算法

	3DES	Blowfish	Twofish	CAST-128	IDEA	RC4
SSH2	x	x	x	-	-	x
F-Secure SSH2	x	x	x	x	-	x
OpenSSH	x	x	-	x	-	x

为什么有些算法在某些程序中不支持呢？在 SSH-1 中，由于 DES 的安全性不足而经常不会支持。舍弃 RC4 是由于它在 SSH-1 协议中的使用方式有问题，会带来一些缺陷而引发网络层主动攻击；这个问题在 SSH-2 中已经修正了。OpenSSH 和非商业的 SSH1、SSH2 产品都不支持 IDEA，因为它是一种专利，商业使用需要支付专利税。OpenSSH 中不支持 twofish，因为它还不是 OpenSSH 所使用的 OpenSSL 一个工具包的一部分。CAST-128 是免费的，因此我们不知道非商业的 SSH2 产品为什么都不支持它。

SSH2 的免费版本只支持 DSA 用于公钥认证，而商业化的 F-Secure SSH2 服务器还支持为 RSA 密钥用于用户认证。[\[6.2.2\]](#)如果主机密钥是 RSA，那么 F-Secure 服务器就启动，并报告说已经成功读取密钥。然而，它仍然宣称自己的密钥交换消息中宣扬自己的主机密钥类型是 DSA，然后提供 RSA 密钥，这样导致客户端在试图读取所提供的密钥时产生错误。当然，整个问题掩盖了客户端是否可以处理 RSA 主机密钥的问题，即使已经正确标识也是如此。OpenSSH/2 根本就不支持 RSA，但是现在 RSA 的专利已经超期了，因此 ssh-rsa 密钥类型很快就会加入到 SSH-2 协议中，各种支持应该还会稍晚一些。

现在我们小结一下前面提到的各种算法；但是不要认为这些总结是全面的分析。不

应该不考虑其他部分就单纯地从单个算法的特征来（肯定地或否定地）推断整个系统。这样安全性就更强了。

3.9.1 公钥算法

3.9.1.1 Rivest-Shamir-Adleman (RSA)

Rivest-Shamir-Adleman公钥算法 (RSA) 是应用最广泛的一种非对称加密算法。其安全性由把一个大整数因式分解成两个大小差不多的素数的乘积的难度而产生。因式分解被广泛认为是非常难处理的（也就是说是不可实行的，没有什么有效的省时的解决方法），但是这还没有得到证明。RSA 既可以用于加密，也可以用于签名。

直到2000年9月为止，RSA在美国一直都是Public Key Partners, Inc. (RSA Security, Inc. 是这家公司的股东) 的专利。（该算法现在可以用于公用领域了。）由于专利的保护，PKP宣称自己负责控制RSA算法在美国的使用，所有未经授权的实现都是非法的。直到90年代中期，RSA Security 提供了一种免费的参考实现 RSAref，同时提供一个许可证允许教育和广泛的商业用途使用（只要软件本身不销售获利即可）。虽然它很通用，但是该公司已经不再支持或发行这个工具包了。由于现在RSA已经在公用领域使用了，因此没有理由再使用RSAref了：该工具包已经不再被支持了，有些版本包含安全漏洞，另外也有一些更好的实现；我们不推荐再使用RSAref。

SSH-1 协议明确地指定使用 RSA。SSH-2 可以使用多种公钥算法，但是只定义了 DSA。^[3.9.1.2]SECSH 工作组现在计划为 SSH-2 增加 RSA 算法，因为其专利权已经过期了。在此期间，只有 F-Secure 的 SSH2 服务器在 SSH2 中使用全局密钥格式标识符“ssh-rsa”实现了 RSA 密钥。但是这并非草案标准的一部分：要保证技术上正确，它必须使用本地化的名字，例如 `ssh-rsa@datafellows.com`。^[3.5.1.1]然而，这并不会引起什么实际问题。这种特性对于使用 SSH1 密钥向 SSH2 服务器来说非常重要，因为这样你就不必创建一个新 (DSA) 密钥。

3.9.1.2 数字签名算法 (DSA)

数字签名算法是由美国国家安全局 (NSA) 开发并由美国国家标准技术研究所 (NIST) 作为数字签名标准 (Digital Signature Standard, DSS) 的一个部分进行发布的。DSS 在 1994 年 5 月作为一份联邦信息处理标准 FIPS-186 颁布。它是一种公

钥算法，基础是 Schnorr 和 ElGamal 方法，其可靠性依赖于在有限空间内计算离散对数的难度。最初它被设计成只能用于签名的一种技术，而不能用于加密，但是完整的基本实现都可以很容易用来执行 RSA 和 ELGANAL 加密。

从 DSA 诞生开始，就陷入了争论的泥潭中。NIST 最初宣称自己设计了 DSA，最终却证明是 NSA 设计了 DSA。由于有充足的历史原因，很多人都对 NSA 的动机和道德水平提出了置疑（注 22）。研究者 Gus Simmons 发现 DSA 中有一个后门，让实现者可以泄漏信息（例如，所有签名的密钥位）（注 23）。由于该算法要在政府的 Capstone 项目中作为一个硬件实现存储在智能卡中，因此很多用户都对此提出了质疑。最终结果是 NIST 将 DSA 免费供给所有用户使用。此前它是 David Kravitz 的专利（专利号 #5,231,668），他后来供职于 NSA，并将该专利所有权捐献给美国政府。但是，也有报道声称 DSA 侵犯了已有的密码专利，包括 Schnorr 专利。就我们所知，这种观点还没有在法庭中得到证实。

SSH2 协议使用 DSA 作为主机识别所需要（也是目前惟一定义的）的公钥算法。

3.9.1.3 Diffie-Hellman 密钥协议

Diffie-Hellman 密钥协议算法是最原始的公钥系统，由 Whitfield Diffie、Martin Hellman 和 Ralph Merkle 在 1976 年发明，并在 1977 年获得专利（颁布于 1980 年，专利号 #4,200,770）；该专利现在已经过期了，其算法已经在公共领域中广泛使用。和 DSA 一样，其可靠性来自于解决离散对数问题的难度，它允许双方通过一条开放的通道安全地获得共享密钥。也就是说，双方交换信息之后，最终共享私钥。窃听者不可能仅仅通过监视交换报文就确定出共享密钥。

SSH-2 使用 Diffie-Hellman 算法作为必须的（也是目前惟一定义的）密钥交换方法。

注 22：请参看 James Bamford 的《the Puzzle Palace》(Penguin) 一书，其中研究了 NSA 的历史。

注 23：G. J. Simmons, "The Subliminal Channels in the U.S. Digital Signature Algorithm(DSA),"《Proceedings of the Third Symposium on: State and Progress of Research in Cryptography》，Rome: Fondazione Ugo Bordoni, 1993, 第 35 ~ 54 页。

3.9.2 密钥算法

3.9.2.1 国际数据加密算法 (IDEA)

国际数据加密算法 (International Data Encryption Algorithm, IDEA) 是由 Xuejia Lai 和 James Massey 在 1990 年设计的 (注 24)，它已经经历了几次修订、改进甚至重命名才变成现在的样子。虽然它还比较新，但却是相当安全的；著名的密码学家 Bruce Schneier 在 1996 年认为“它是目前公众可用的最安全的算法”。

IDEA 在欧洲和美国都由瑞士公司 Ascom-Tech AG 申请了专利 (注 25)。Ascom-Tech 的商标就是“IDEA”。随着时间的推移，Ascom-Tech 对于该专利的态度以及 IDEA 在美国的使用已经发生了变化，尤其是它本来要嵌到 PGP 中的态度。现在它免费供给非商业用途使用。政府或商业用途都需要授权，此处的“商业用途”包括商业组织内部的使用，而不仅仅指销售其实现版本或从使用中获利。这里有两个站点有更详细的信息：

<http://www.ascom.ch/infosec/idea.html>

<http://www.it-sec.com/idea.html>

3.9.2.2 数据加密标准 (DES)

数据加密标准 (Data Encryption Standard, DES) 是一种历经沧桑的对称加密算法。DES 是 IBM 的研究者在 20 世纪 70 年代早期以 Lucifer 的名义设计的，美国政府于 1976 年 9 月 23 日将 DES 列为一项标准 (FIPS-46)。其专利权属于 IBM，但是 IBM 又授权全世界免费使用。从此之后，它在公共加密和私有加密领域就得到了广泛的应用。DES 已经接受了若干年密码分析的考验，表现良好，只是由于它使用的 56 位的密钥对于现代计算机的计算能力来说实在太小了，所以现在被认为已经过时了。现在已经出现了很多很好的专用“DES 解密机”，其价格越来越能被政府和大公司所接受，至少 NSA 似乎有这种设备。由于这些缺点，NIST 现在已经开始继续开发下一代 DES，称为高级加密标准 (Advanced Encryption Standard, AES)。

注 24：X. Lai 和 . Massey, “A proposal for a New Block Encryption Standard,”《Advances in Cryptology - EUROCRYPT '92 Proceedings》，Springer-Verlag, 1992, 第 389 – 404 页。

注 25：美国专利号：#5,214,703, 1993 年 5 月 25 日；国际专利号：PCT/CH91/00117, 1991 年 11 月 28 日；欧洲专利号：EP 0 482 154 B1。

3.9.2.3 Triple-DES (3DES)

Triple-DES（或称为3DES）是DES的一个变种，目的是要通过增加密钥长度来增加安全性。现在已经证明DES函数不会产生密钥收敛（注26），这就是说使用单独密钥多次加密可以增加安全性。3DES使用三个单独的密钥应用三次DES算法对明文进行加密。3DES的有效密钥长度是112位，这对于普通的DES的56位的密钥来说是一个很大的改进。

3.9.2.4 ARCFOUR (RC4)

Ron Rivest在1987年为RSA Data Security, Inc. (RSADSI) 设计了RC4加密算法；这个名字在不同地方可能分别代表“Riverst算法”或“Ron密码”。由于RSADSI的商业机密，RC4没有申请专利，它广泛用于使用RSADSI许可证的很多商业产品中。但是在1994年，在Internet上匿名出现了声称是实现RC4的源代码。实验很快证明这些代码实际上的确可以兼容RC4，这段代码是不小心泄漏出来的。由于RC4从来都没有申请过专利，因此很快就进入了公共应用领域。这并不是说RSADSI不会控告别人在商业产品中使用RC4，因为与其打官司来解决还不如稍微花点钱去获得一份许可证呢。我们并没有对此进行任何测试。由于“RC4”是RSADSI的商标，人们就生造出一个词“ARCFOUR”来，用来指该算法泄漏到公共领域的版本。

ARCFOUR速度非常快，但是并没有像其他算法一样经受仔细的研究。它使用大小可变的密钥；SSH-1为SSH会话的每个方向都使用单独的128位密钥。每个方向上单独使用密钥在SSH-1中是一个特例，而且也相当关键：ARCFOUR实际上是使用一个伪随机数生成器的结果作为一层铺垫。因此它不能重用一个密钥，因为这样就使得密码分析变得非常容易。如果考虑到ARCFOUR所带来的法律诉讼的因素，尽管没有多少公共分析结果，但是很多人都认为ARCFOUR是相当安全的。

3.9.2.5 blowfish 算法

blowfish算法是由Bruce Schneier在1993年设计的，当时作为过时的DES的一种过渡品。其速度比DES和IDEA都快得多，但是还不如ARCFOUR快，也没有申请专利，任何用户都可以免费使用。它被特别设计用于实现大型的现代通用微处理器，

注26：K. W. Campbell和M. J. Wiener, “DES Is Not a group”, 《Advances in Cryptology -- CRYPTO '92 Proceedings》, Springer-Verlag, 第512~520页。

以及用于那些密钥变化很少的情况。它不太适合诸如智能卡之类的低端环境。它采用长度可变的密钥（从 32 位到 448 位）；SSH-2 使用 128 位的密钥。blowfish 已经经受了大量密码分析审查，已经证明不会受到目前攻击手段的损害。这方面的信息可以在 Counterpane, Schneier 的安全构建公司中找到：

<http://www.counterpane.com/blowfish.html>

3.9.2.6 twofish 算法

twofish 算法是由 Bruce Schneier 和 J. Kelsey、D. Whiting、D. Wagner、C. Hall 以及 N. Ferguson 一起设计的。它于 1998 年被当作一种替代高级加密标准的候选方案提交给 NIST，要用来作为美国政府的对称数据加密算法标准替代 DES。两年之后，它成为从 15 个最初提案中脱颖而出的五个 AES 的候选方案之一。和 blowfish 类似，它也没有申请专利，免费供所有用户使用，Counterpane 已经提供了无版权的参考实现版本，这也是免费的。

twofish 允许使用 128 位、192 位或 256 位的密钥；SSH-2 指定使用 256 位的密钥。twofish 设计得比 blowfish 更灵活，可以在各种计算环境中（例如，慢处理器、小内存、嵌入硬件）很好地实现。其速度很快、设计保守（conservative）、相当健壮。可以在这里了解 towfish 的更多内容：

<http://www.counterpane.com/twofish.html>

可以在这里了解有关 NIST AES 的更多信息：

<http://www.nist.gov/aes/>

3.9.2.7 CAST

CAST 是由 Carlisle Adams 和 Stafford Tavares 在 20 世纪 90 年代早期设计的。Tavares 在加拿大 Kingston 的 Queen 大学工作，而 Adams 在德克萨斯州的 Entrust 技术公司工作。CAST 已经申请了专利，专利权归 Entrust 所有，该公司已经设计了两个版本的算法免费供给所有用户使用。这两个版本分别是 CAST-128 和 CAST-256，分别在 RFC-2144 和 RFC-2612 中介绍。SSH-2 使用 CAST-128，之所以这样命名是因为其密钥长度就是 128 位。

3.9.2.8 速度比较

我们曾经运行了一些简单的实验来比较各种加密算法的速度。由于没有一个SSH包可以包括所有的加密算法，因此我们使用了两个实验来覆盖所有的加密算法。表3-7和表3-8给出了把一个5MB的文件通过10-base-T以太网从300MHz的Linux上传输到一个100MHz的Sparc-20上所需要的时间。

表3-7：使用scp2 (F-Secure SSH2 2.0.13) 传输文件

加密算法	传输时间 (seconds)	吞吐量 (KB/秒)
RC4	22.5	227.4
Blowfish	24.5	208.6
CAST-128	26.4	193.9
Twofish	28.2	181.3
3DES	51.8	98.8

表3-8：使用scp1 (SSH-1.2.27) 进行相同的测试

加密算法	传输时间 (seconds)	吞吐量 (KB/秒)
RC4	5	1024.0
Blowfish	6	853.3
CAST-128	7	731.4
Twofish	14	365.7
3DES	15	341.3

当然这是一个宏观的比较，我们给出这些数据只是为了给大家一个大概的印象。记住这些数字反映的是单个配置测试中特定实现的性能，而不是算法本身的性能。使用不同的配置进行测试，得到的结果也就不同。各种算法之间的差别实际上比表面上看起来的要小。

注意scp1的速度大概比scp2快4倍。这是由于实现上的差异：scp1使用scp1-t服务器，而scp2使用SFTP子系统。[\[7.5.9\]](#)然而，它们交叠部分的相应的加密算法速度是一致的。

我们必须强调，我们之所以在SSH1测试中包含RC4只是为了完整进行比较的需要；由于安全性的缺点，RC4通常都不应该在SSH-1协议中使用。

3.9.3 散列函数

3.9.3.1 CRC-32

32位循环冗余校验（CRC-32）在 ISO 3309（注 27）中定义，它是一个用于检测数据发生偶然变换的无加密的 hash 函数。SSH-1 协议使用 CRC-32 进行完整性检查，这是一个缺陷，它为后面讨论的“插入攻击”大开了方便之门。[3.10.5] SSH-2 协议使用强加密 hash 函数进行完整性检查，从而可以防止这种攻击。

3.9.3.2 MD5

MD5（Message Digest algorithm number 5，消息摘要算法 5）是密码学中一种强加密的 128 位的散列算法，由 Ron Rivest 在 1991 年设计，这是他为 RSADSI 设计的算法（MD2 和 MD5）之一。MD5 没有申请专利，RSADSI 将其用于公共领域，它在 RFC-1321 中详细介绍。它成为一种标准散列算法已经有几年的历史，在很多加密产品和标准中广泛使用。den Boer 和 Bosselaers 在 1993 年成功地对 MD5 的压缩函数进行了冲突攻击，虽然这次攻击并没有造成什么实际损害，但是可能以后就会造成这种损害了，人们已经开始使用新加密算法来代替 MD5。RSADSI 自己推荐在已有的要求防冲突的应用程序中使用 SHA-1 或 RIPEMD-160 来代替 MD5（注 28）。

3.9.3.3 SHA-1

SHA-1（Secure Hash Algorithm，安全散列算法）是由 NSA 和 NIST 为使用美国政府数字签名标准而设计的。和 MD5 类似，它的设计也是对 MD4 进行改进，但是采取的方法不同。它会生成一个 160 位的散列值。现在还没听说对 SHA-1 有成功的攻击；而且如果 SHA-1 是安全的，那么它肯定比 MD5 更健壮，仅凭其散列值的长度更长就可以证明这一点。在有些应用程序中已经开始使用 SHA-1 替换 MD5；例如，SSH-2 使用 SHA-1 作为 MAC 散列函数，而 SSH-1 使用 MD5。

注 27： 国际标准化组织《ISO Information Processing Systems —— Data Communication High-level Data Link Control Procedure —— Frame Structure》，IS 3309，1984 年 8 月，第 3 版。

注 28： RSA 实验报告 #4，1996 年 11 月 12 日，<ftp://ftp.rsasecurity.com/pub/pdfs/bulletn4.pdf>。

3.9.3.4 RIPEMD-160

还有另外一种 MD4 的变体，就是 RIPEMD-160，它是由 Hans Dobbertin、Antoon Bosselaers 和 Bart Preneel 作为欧洲共同体 RIPE 项目而开发出来的。RIPE 代表 RACE 完整性原语验证（RACE Integrity Primitives Evaluation）（注 29）；其中 RACE 是为研究和开发欧洲的高级通信技术而使用的程序，这是一个从 1987 年 6 月到 1995 年 12 月由 EC 资助的程序（<http://www.race.analysys.co.uk/race/>）。RIPE 是 RACE 的努力方向之一，目标是研究并开发数据完整性检测技术。因此，RIPEMD-160 应该读作“the RIPE Message Digest (160 bits)”（RIPE 消息摘要，160 位）。”特别强调的是，它和 RIPEM 没什么关系，后者是 Mark Riordan 所实现的一种早期的增强隐私邮件（Privacy-Enhanced Mail，PEM）。

虽然 RIPEMD-160 并不是在 SSH 协议中定义的，但是在 OpenSSH 中它被用来以 `hmac-ripemd160@openssh.com` 实现了一种专用的 MAC 算法。RIPEMD-160 并没有申请专利，可以供用户免费使用。可以在这儿了解更多的相关内容：

<http://www.esat.kuleuven.ac.be/~bosselaer/ripemd160.html>

3.9.4 压缩算法：zlib

zlib 是目前惟一一种为 SSH 定义的压缩算法。在 SSH 协议文档中，术语“zlib”是指在通用的 `gzip` 压缩工具中首先实现的“deflate”无损压缩算法，后来文档记录成 RFC-1951。现在它已经成为一个软件库，名为 ZLIB，主页位于：

<http://www.info-zip.org/pub/infozip/zlib/>

3.10 SSH 可以防止的攻击

和所有的安全工具一样，SSH 可以有效地防止一些攻击，但是另外有一些攻击它不能防止。我们首先讨论前一种类型。

注 29：不要和另外一个“RIPE”混淆了，它是“Réseaux IP Européens”（欧洲 IP 网络），一种在欧洲和其他地方操作大地区 IP 网络工程的技术和实体的等同结合。（<http://www.ripe.net>）。

3.10.1 窃听

窃听者就是网络监听者，他可以丝毫不被发觉地读取网络信息。SSH的加密防止了窃听的危害。窃听者即使截获了SSH会话的内容，也不能将其解密。

3.10.2 名字服务和IP伪装

如果攻击者搞乱了你的名字服务（DNS、NIS等），那么和网络有关的程序就可能被强制连接到错误的机器上。同理，攻击者可以通过盗用IP地址来冒充一台机器。不管是哪一种情况，你都会碰到问题：你的客户端程序可能连接到了一台错误的机器上，在提供密码时，你的密码就可能会被窃取。SSH通过加密验证服务器主机的身份避免了这个风险。在设置会话时，SSH客户端会根据和密钥关联的本地服务器名列表和地址列表对服务器主机密钥进行验证。如果所提供的主机密钥不能和该列表中的任意一项匹配，那么SSH就会报错。如果这些警告错误不停地骚扰你，那么在一些安全性设置较低的情况下你可以禁用这种特性。[\[7.4.3.1\]](#)

SSH-2协议允许一同使用PKI证书和密钥。将来，我们希望在SSH产品中把这种特性和通用的PKI一起实现，这样可以降低密钥管理的负担并减少这种特殊安全性折衷的需要。

3.10.3 连接劫持（Connection Hijacking）

“主动攻击者”（他不但可以监听网络信息，而且在其中可以插入自己的内容）可以劫持一个TCP连接，字面上的意思就是使这个TCP连接偏离开正确的终点。这样所造成危害是很显然的：不管认证方法是如何好，攻击者都可以简单地一直等待到你登陆，然后窃取你的连接并在你的会话中插入自己的恶意的命令。SSH不能防止连接劫持，因为这是TCP本身的缺陷，而SSH是在TCP之上运行的。然而，SSH以一种低效的方式对这个问题进行了解决（有些类似于服务拒绝攻击）。SSH的完整性检测负责确定会话在传输过程中是否被修改过了，如果被修改了，就立即关闭连接，而不会使用任何被修改过的数据。

3.10.4 中间人攻击

中间人攻击是一种十分狡猾的主动攻击手段，如图3-8所示。攻击者位于你和你实

际的通信目标之间（也就是位于 SSH 客户端和服务器之间），他可以截获所有的通信并任意修改或删除消息。假设你试图连往一个 SSH 服务器，但是攻击者 Mary 截获了你的连接。它模拟一个 SSH 服务器的操作，这样你就发觉不了，结果是她和你共享了同一个会话密钥。同时，她又向你的目标服务器发起一条连接，获得服务器上一个单独的会话密钥。她可以以你的身份登录系统，因为你使用的是密码认证并把自己的密码交给了她。你和服务器都认为已经互相建立了连接，但是实际上你们都是和 Mary 建立的连接。然后她就可以坐在中间，在你和服务器之间转发数据（在一个方向上使用一个密钥解密，在另外一个方向上使用另外一个密钥重新加密）。当然，她可以读取所有转发的内容，如果她愿意，还可以任意修改这些内容。

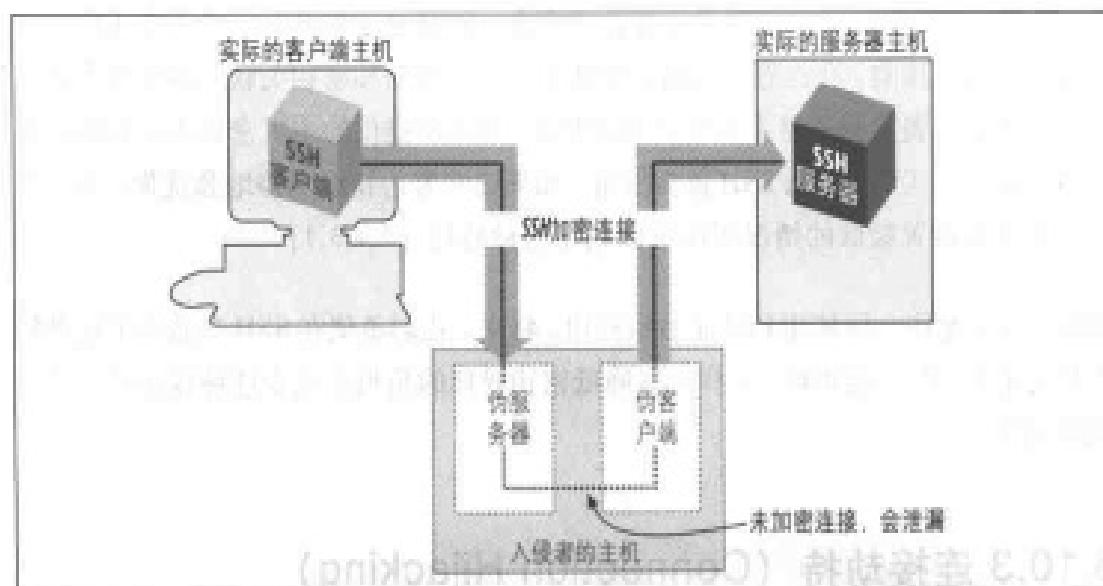


图 3-8：中间人攻击（通过中间人攻击，一旦你不知道自己正在和谁通信，你可能就会被欺骗。）
SSH 可以使用两种方法来防止这种攻击。第一种方法是服务器主机认证。除非 Mary 已经成功攻击了服务器主机，否则就不能扮演中间人的角色，因为她没有服务器的私有主机密钥。注意为了保证这种保护措施能正确运行，客户端必须根据自己的已知名主机列表对服务器提供的公共主机密钥进行检查；否则，就不能保证服务器就是真正的服务器。如果你第一次连接到一台新服务器上，并让 ssh 接受主机密钥，那么你实际上就为中间人攻击打开了一个后门。然而，假设你这一次没有被攻击者欺骗，只要这台服务器的主机密钥没被窃取，那么以后到这个服务器的连接也都是安全的。

SSH 防止中间人攻击的第二种方法是限制使用那些容易受到这种攻击的认证方法。

密码认证很容易受到中间人攻击，而公钥和基于主机的/RhostsRSA则对中间人攻击可以免疫。Mary 只通过监视密钥交换是不能得到会话密钥的；她必须执行一次主动攻击并和每一方都进行单独交换报文才能从服务器和客户端获取两个密钥。在SSH-1 和 SSH-2 中（注 30），密钥交换就是这样设计的；如果 Mary 真的这样做了，那么会话两端的会话标识符就不一样了。当客户端为每个公钥或可信主机认证都提供一个数字签名时，在已签名的数据中包括了会话标识符。这样，Mary 就不可能只伪造客户端提供给服务器的认证者，也不可能有什么法子强迫客户端对其他会话ID 进行签名。

如果不对服务器名、密钥的通信进行验证，那么 Mary 就可以继续执行中间人攻击，当然现在她不可能以你的身份登录到服务器端。但是她可以登录到自己的账号或其他已经成功攻击过的账号中。使用一些技巧，她仍然可以对你进行欺骗直至造成损害。

3.10.5 插入攻击

回想一下SSH-1使用的完整性检查机制是非常脆弱的。Ariel Futoransky 和 Emiliano Kargieman 在 1998 年 6 月成功地利用这种缺陷对 SSH-1 进行了攻击；纪实资料请参看 <http://www.core-sdi.com/advisories/ssh-advisory.htm>。这种“插入”（或称为补偿，compensation）攻击可以让那些执行主动网络攻击的攻击者在客户端和服务器之间发送的正文数据流之间插入任意数据。也就是说，它允许在连接中插入加密数据，最终会解密成攻击者所希望的数据，SSH 会毫无知觉地转发这些数据。发往服务器方向的数据是问题最严重的地方，因为这可以允许攻击者向用户的终端会话中插入任意的命令。虽然这并不是一个特别容易被攻击的点，但却是一个十分严重的缺陷。这种攻击是 CRC-32 和某些模式的加密算法的综合结果。我们可以使用 3DES 算法来防止这种攻击，3DES 对这种攻击是免疫的。

SSH1 1.2.25、F-Secure SSH1 1.3.5 及其后续版本，还有 OpenSSH 的所有版本（包括 crc32 插入攻击监测器）都专门进行了设计，来检测并防止这种攻击。这些检测程序增大了插入攻击的难度，但是并不能完全防止插入攻击发生的可能。SSH-2 使用强加密完整性检测手段来防止这种问题。

注 30： 至少可以使用 diffie-hellman-group1-sha1 方法交换密钥。我们假设如果以后增加的其他密钥交换方法也都具有这种特性。

3.11 SSH 不能防止的攻击

SSH并不是一个完整的安全方案。我们现在就给出几个SSH不能预防的攻击例子。

3.11.1 密码崩溃

SSH通过把密码在网络上传输时对其进行加密而极大地提高了密码的安全性。然而，密码认证仍然是一种脆弱的认证形式，在使用时必须小心。你必须选择良好的密码，密码必须对你来说容易记忆，但是对别人来说又不能太容易猜到。你还要确保自己的密码不会被窃取，因为一旦拥有了你的密码就足以访问你的账号。因此要当心：另外一个终端上说不定哪个家伙正在秘密地“肩窥(shoulder-surfing)”你（监视你从键盘上输入的内容）。你要使用的计算机可能已经被攻击，从而从键盘敲入的所有命令都会记录到攻击者的控制命令（Cracker Central Command）中。从IT协会打电话给你并问你密码要“修复你账号”的那个“好心”的家伙也未必就是他所说的人。

让我们反过来考虑一下公钥认证，因为它必须要有两个要素：如果没有私钥文件，口令即使被窃取了也没什么用处，因此攻击者需要得到口令和密钥文件才能进行攻击。当然，你正在使用的计算机上的SSH客户端可能受到他人的控制，当你毫无知觉地输入口令给私钥解密时，这个客户端就会像松鼠偷东西一样把你的密钥偷走了。如果你担心这一点，就不要使用不熟悉的计算机。我们相信将来加密智能卡和读卡机会无处不在，而且SSH也会对其提供支持，因此到时候你就可以方便携带自己的密钥，并可以在其他机器上使用而不用担心私钥泄漏的问题了。

如果你因为方便而必须使用密码认证，那么就可以考虑使用诸如S/Key之类的一次性密码机制来减少风险。[\[3.4.2.5\]](#)

3.11.2 IP 和 TCP 攻击

由于SSH是在TCP之上进行操作的，因此它也容易受到针对TCP和IP缺陷而发起的一些攻击。隐私性、完整性和认证可以确保SSH把服务拒绝攻击的危害限制在一定范围之内。

TCP/IP可以防止诸如网络拥塞和链路失效之类的网络问题。如果攻击者对路由器狂发信息，那么IP可以将其隔离在路由器之外。但是TCP/IP的设计并不能防止在网

络中插入伪造的报文。TCP或IP控制消息的源头并不需要认证。结果是TCP/IP也继承了很多缺陷，例如：

SYN 灾难 (SYN flood)

SYN 代表“同步 (synchronize)”，这是 TCP 报文的一个属性。在这种情况下，它指最初发送出来用来建立 TCP 连接的那个报文。该报文通常都会让接受者消耗大量的资源来准备接收要到达的连接。如果攻击者发出了大量这种报文，那么接收 TCP 的栈就可能溢出而不能接收合法的连接了。

TCP、RST 和伪 ICMP

另外一种 TCP 报文类型是 RST，表示“重置 (reset)”。TCP 连接的任何一方在任何时候都可以发送一个 RST 报文，该报文会导致立即挂断该连接。RST 报文可以很容易插入网络，结果是立即断开和任何目标的 TCP 连接。

同理，还有 ICMP，也就是 Internet 控制消息协议 (Internet Control Message Protocol)。ICMP 允许 IP 主机和路由器进行通信以了解网络状态和主机可达性。然而，这也不需要认证，因此在网络中插入伪 ICMP 报文会造成恶劣的影响。例如，有些 ICMP 消息说某个特定主机或 TCP 端口不可达了；伪造这种报文可以导致挂断连接。还有些 ICMP 消息负责交换路由信息（重定向和路由发现）；伪造这种报文可以导致敏感数据会被肆意路由转发，从而可能会对系统造成危害。

TCP 不同步和 TCP 劫持

通过巧妙地对 TCP 协议进行处理，攻击者可以使 TCP 连接的双方对数据字节流次序编号失去同步。在这种状态下，攻击者可以插入一些报文，这些报文会被当成是该连接的合法部分而接收，这样攻击者就可以在 TCP 数据流中插入任意数据。

SSH 没有为中断或禁止建立 TCP 连接的攻击提供任何防护措施。另一方面，SSH 的加密和主机认证都可以有效地防止路由不适当的攻击，这种情况中要么允许读取敏感数据，要么会把连接重定向到已经被攻击的服务器上。同理，劫持或修改 TCP 数据的攻击也会失效，因为 SSH 可以检测到这些变化，但是它们仍然可以中断连接，因为 SSH 本身也会对终端上产生的这种问题作出同样的响应。

因为这些问题都是由于 TCP/IP 的缺陷而产生的，所以只能通过更低级的网络层技术才能很好地解决，例如硬件链路加密或 IPSEC。[1.6.4]IPSEC 是 IP 安全协议，它是

下一代IP协议IPV6的一个部分，可以用作现在的IP标准IPV4的一个增强。IPSEC在IP报文层提供了加密、完整性和数据源认证服务。

3.11.3 流量分析

即使攻击者不能理解你的网络通信的具体内容，他只通过简单地监视网络通信（监视网络通信的数据量、源地址和目的地址以及时间）就可以获得很多有用信息。你和另外一个公司的网络通信突然增加就可能会提示他，你们可能正要发生一笔商业交易。流量的模式也可以显示出备份调度或每天的哪些时间最容易受到服务拒绝的攻击。来自系统管理员工作台上的SSH连接长时间没有动作就说明她可能离开了，现在正是闯入的好机会，不管是电子手段入侵还是物理闯入都是如此。

SSH不能解决流量分析的攻击。当我们使用SSH连往一个知名端口时，很容易就可以识别出SSH连接，SSH协议并没有增加流量分析的难度。SSH实现可以在空闲时发送一些随机的、非操作性的通信来干扰活动状态的分析，但是就我们所知现在还没有哪种SSH实现中具有这种特性。

3.11.4 隐秘通道

隐秘通道(*covert channel*)的意思是说使用难以预料或不会引起注意的方式来发送信息。假设有一天系统管理员Sally觉得自己的那些用户花太多时间来玩了，因此她决定关闭email和即时消息，这样他们就不能聊天了。要解决这个问题，你和你的朋友都同意使用主目录中的一个你们都可以读取的文件来交流信息，你每隔一会儿就得检查一下该文件来查看新消息。这种超出乎预料的通信方式就是一个隐秘通道。

隐秘通道很难取消。如果系统管理员Sally发现了你们这种基于文件的通信技术，她可以修改目录的权限，使得只有目录的属主才具有读取和搜索的权限，并限定目录属主也不能修改这种权限。在这样做的同时，她还可以确认你不能在其他地方（例如，*/tmp*）创建文件。（你的大部分程序都不能运行了，但是不要因此而责怪Sally。）即使是这样，你和你的朋友仍然可以相互显示其他用户主目录中的内容，这可以显示目录的修改日期和文件个数，这样你就可以根据这些可见的参数设计一种密码，并通过修改这些参数而进行通信。这是一种更复杂的隐秘通道，如果Sally对你们进行了更严格的限制，你们可以想出更古怪的法子来。

SSH不能防止隐秘通道。对隐秘通道的分析和控制通常都是安全性很高的计算机系统的一部分，例如设计用来在同一个系统中的不同安全层次上安全地处理信息的系统。顺便说一下，SSH数据流本身就可以很好地用作一种隐秘通道：SSH会话的内容可能是一个巧克力馅饼的配方，而隐含的内容则可能是用Morse码表示这两个公司即将合并，只需要使用报文长度是奇数还是偶数来分别表示Morse码的短横线和点号即可。

3.11.5 粗心大意

Mit der Dummheit köpfen Götter selbst vergebens.

(即使是上帝，对那些愚蠢的人也无能为力。)

— Friedrich von Schiller

安全工具本身并不能增加任何东西的安全性；这些工具只是能帮助人们来增加其他内容的安全性而已。这种说法有些迂腐，但却如此重要，根本就不需要重复。如果用户使用非常差的密码，或把口令写在纸条上贴在键盘下边，那么即使是世界上最好的密码或最安全的协议也无能为力；如果系统管理员总是忽略主机安全性的其他方面，从而导致主机密钥丢失或搭线窃听，那么任何密码或协议都解决不了问题。

正如Bruce Schneier所说的一样，“安全是一个过程，而不是一个产品”。SSH是一种很好的工具，但是它只能是一套完整的安全过程的一个部分；我们还应该保证主机完整性的其他方面；考虑安全顾问对相关软件和操作系统的建议，正确使用适当的补丁并合理配置应用环境，教育用户并保持用户对安全责任的警觉性。不要只把SSH安装上就认为安全了；还差得远呢。

3.12 小结

SSH协议使用公开发行的强加密工具来确保网络连接的隐私性、完整性和相互认证。SSH-1协议（也就是SSH-1.5）虽然稍微有些特别，但却应用十分广泛：它实际上是SSH1程序行为的文档记录。它还有很多缺陷，其中脆弱的完整性检测及由此而导致的Futoransky/Kargieman插入攻击可能是最严重的问题。现在的SSH协议版本SSH-2在实现上更加灵活，已经修正了SSH-1的一些问题，但是不幸的是由于许可

证的限制而使得其实施采用非常有限，很多商业用途只好继续使用免费的 SSH1 软件。

SSH 解决了很多和网络有关的安全漏洞，但并没有全部解决，尤其是它仍然容易受到针对其底层 TCP/IP 缺陷而发起的服务器拒绝攻击；它也不能解决一些考虑环境因素而产生的攻击方法，例如流量分析和隐秘通道。

本章内容：

- SSH1 和 SSH2
- F-Secure SSH 服务器
- OpenSSH
- 软件清单
- 使用 SSH 代替 Telnet 命令
- 小结

第四章

SSH 的安装 和编译时配置

现在我们已经知道 SSH 是什么以及 SSH 是如何工作的了，但是从哪儿能获得 SSH 呢？又如何安装 SSH 呢？本章介绍了 SSH 的几种通用而且健壮稳定的 Unix 实现，并介绍了如何获得、编译并安装这些软件，包括：

SSH1 和 SSH 安全 Shell (SSH2)

SSH Communications Security, Ltd. 的产品，分别实现了 SSH-1 和 SSH-2 协议。

F-Secure SSH 服务器

F-Secure Corporation 的 SSH1 和 SSH2 产品。

OpenSSH

一种免费的 SSH1 产品，对 SSH-2 协议的支持有单独的产品；OpenBSD 的一部分。

SSH 的非 Unix 实现将在第十三章到第十七章中介绍。

4.1 SSH1 和 SSH2

SSH1 和 SSH2（也就是 SSH 安全 Shell）最初都是为 Unix 编写的，现在已经移植到

其他操作系统上了。这两个产品是以源码形式发布的，在使用之前必须进行编译，当然各种平台上也有预编译好的可执行代码。

SSH1 和 SSH2 对于非商业用途可以免费使用。如果希望将其用于商业目的，那么根据许可证，必须购买该软件。其商业版本由 SSH Communication Security, Ltd. 和 F-Secure Corporation 两个公司负责销售，后面我们很快就会介绍相关内容。这两个软件的拷贝和使用的详细条款在 *COPYING* 文件（对于 SSH1）和 *LICENSING* 文件（对于 SSH2）中给出。在使用之前，要确信已经研读并正确理解了相应的条款。而且，由于这些产品都会涉及加密的问题，用户的本地法律可能会对用户是否可以使用或发布该软件有所规定。

4.1.1 特点

SSH1 和 SSH2 为 SSH 的特点定义了事实上的标准，具有很大的灵活性和很强的功能。这两种产品都有以下特点：

- 客户端程序（远程登录、远程命令执行以及通过网络安全拷贝文件）有很多运行时选项。
- SSH 服务器高度可配置。
- 所有程序都有命令行接口，可以方便地使用标准的 Unix 工具（shell、Perl 等）编写脚本。
- 具有很多可选的加密算法和认证机制。
- 有 SSH 代理，可以对密钥进行缓存以便使用。
- 支持 SOCKS 代理。
- 支持 TCP 端口转发和 X11 转发。
- 具有历史记录和日志记录的特点，以便于调试。

4.1.2 获得 SSH 的发行版本

SSH1 和 SSH2 都可以从匿名 FTP: [ftp.ssh.com](ftp://ftp.ssh.com) 上的 */pub/ssh* 目录中获得，也就是下面的 URL:

ftp://ftp.ssh.com/pub/ssh/

也可以从 SSH Communications Security, Ltd. 的 Web 站点上找到：

http://www.ssh.com/

4.1.2.1 解压文件

发行版都以使用 *tar* 格式的文件进行打包，并使用 *gzip* 进行压缩。要解开这些文件，可以使用 *tar* 和 *gunzip* 命令。例如，要从 *ssh-1.2.27.tar.gz* 中解压 SSH1 V1.2.27 文件，需要输入：

```
$ gunzip ssh-1.2.27.tar.gz  
$ tar xvf ssh-1.2.27.tar
```

也可以利用管道使用一个命令：

```
$ gunzip < ssh-1.2.27.tar.gz | tar xvf -
```

还可以使用 GNU Tar（在某些系统中称为 *gtar* 或 *tar*），简单地输入：

```
$ gtar xzvf ssh-1.2.27.tar.gz
```

执行的结果会生成一个新的子目录，其中包含了发行版本中的所有文件。

4.1.2.2 使用 PGP 进行验证

与每个 SSH1 和 SSH2 发行版一起发行的还有一个 PGP 签名文件，该文件用来保证该发行版的确是真实的，并且未曾被修改过。^[1.6.2]例如，与 *ssh-1.2.27.tar.gz* 文件一起的有一个 *ssh-1.2.27.tar.gz.sig* 文件，其中就包含了 PGP 签名。要验证该文件是真实无误的，需要安装 PGP。然后：

1. 如果是第一次使用 PGP，就需要先获得发行版的 PGP 公钥。下面的公钥是用来验证 SSH1 和 SSH2 的：

ftp://ftp.ssh.com/pub/ssh/SSH1-DISTRIBUTION-KEY-RSA.asc

ftp://ftp.ssh.com/pub/ssh/SSH2-DISTRIBUTION-KEY-RSA.asc

ftp://ftp.ssh.com/pub/ssh/SSH2-DISTRIBUTION-KEY-DSA.asc

然后分别把这些公钥保存到临时文件中，并输入下面的命令把这些公钥加入 PGP 密钥环中：

```
$ pgp -ka temporary_file_name
```

2. 下载这两个发行版本的文件（例如，*ssh-1.2.27.tar.gz*）和签名文件（例如，*ssh-1.2.27.tar.gz.sig*）。
3. 使用下面的命令对签名进行验证：

```
$ pgp ssh-1.2.27.tar.gz
```

如果没有出现警告信息，那么下载的发行版文件就是真实无误的。

一定要记得验证PGP签名。否则，就可能会使用一个不可信的第三方修改过的版本，那就被骗了。如果不验证PGP签名就盲目地安装程序，就可能会危害系统的安全。

4.1.3 编译并安装 SSH1

通常，SSH1都可以按照以下步骤进行编译和安装。用户应该仔细阅读 *README*、*INSTALL*等文件以及发行版本中的文档，以便了解在自己的环境中安装时是否有一些已知的问题，以及是否需要其他步骤。

1. 运行所提供的配置文件。[4.1.5]如果想全部使用缺省选项，就切换到 SSH1 发行版的根目录下并输入：

```
$ ./configure
```

2. 编译所有的程序：

```
$ make
```

3. 安装所有的程序。如果想在系统目录下安装这些文件，就需要 root 权限：

```
$ su root  
Password: *****  
# 进行安装
```

所安装的文件如下：

- 服务器程序 *sshd1*，以及一个到 *sshd1* 的名为 *sshd* 的链接
- 客户端程序 *ssh1* 和 *scp1*，以及到这两个程序的名为 *ssh* 和 *scp* 的两个链接
- 符号链接 *slogin1* 指向 *ssh1*，还有一个类似的链接 *slogin* 指向 *slogin1*

- 辅助程序 *ssh-add1*、*ssh-agent1*、*ssh-askpass1*、*ssh-keygen1* 以及分别链接到这些程序上的链接 *ssh-add*、*ssh-agent*、*ssh-askpass* 和 *ssh-keygen*
 - 辅助程序 *make-ssh-known-hosts*
 - 新生成的主机密钥对，由 *ssh-keygen* 创建，缺省放入 */etc/ssh_host_key*（私钥）和 */etc/ssh_host_key.pub*（公钥）中
 - 服务器配置文件，缺省是 */etc/sshd_config* [5.3.1]
 - 客户端配置文件，缺省是 */etc/ssh_config* [7.1.3]
 - 各个程序的手册页
4. 创建已知名主机文件。[4.1.6]

4.1.4 编译和安装 SSH2

SSH 的编译和安装与 SSH1 很相似，也是使用配置脚本和一对 *make* 命令：

1. 执行和 SSH1 类似的编译时配置文件。[4.1.5] 如果想全部使用缺省选项，就切换到 SSH2 发行版的根目录下并输入：

```
$ ./configure
```

2. 编译所有的程序：

```
$ make
```

3. 安装所有的程序。记住如果想在系统目录下安装文件，就需要先切换成 root 身份：

```
$ su root  
Password: *****  
# 进行安装
```

所安装的文件如下：

- 服务器程序 *sshd2*，还有一个到 *sshd2* 的名为 *sshd* 的链接
- 安全的 FTP 服务器程序 *sftp-server*
- 客户端程序 *ssh2*、*scp2* 和 *sftp2*，以及到这三个程序的名为 *ssh*、*scp* 和 *sftp* 的链接

- 辅助程序 `ssh-add2`、`ssh-agent2`、`ssh-askpass2`、`ssh-keygen2`、`ssh-probe2` 和 `ssh-signer2`，以及分别到这些程序上的名为 `ssh-add`、`ssh-agent`、`ssh-askpass`、`ssh-keygen`、`ssh-probe` 和 `ssh-signer` 的链接
- 辅助程序 `ssh-dummy-shell` 和 `ssh-pubkeymgr`
- 新生成的主机密钥对，由 `ssh-keygen2` 创建，缺省放入 `/etc/ssh2/hostkey`（私钥）和 `/etc/ssh2/hostkey.pub`（公钥）文件中
- 服务器配置文件，缺省是 `/etc/ssh2/sshd2_config` [5.3.1]
- 客户端配置文件，缺省是 `/etc/ssh2/ssh2_config` [7.1.3]
- 各个程序的手册页

4.1.4.1 在同一台机器上安装 SSH1 和 SSH2

注意 SSH1 和 SSH2 在安装时创建了一些名字相同的文件，例如链接 `sshd`。如果在同一台机器上同时安装 SSH1 和 SSH2 结果会是如何呢？幸运的是，假设安装了最新版本，那么即使把它们安装到同一个 `bin` 和 `etc` 目录下，它们也都可以正常运行。它们的 `Makefile` 都要检测另外一个程序是否存在，并适当地进行调节。

具体来说，SSH1 和 SSH2 都创建了符号链接 `sshd`、`ssh`、`scp`、`ssh-add`、`ssh-agent`、`ssh-askpass` 和 `ssh-keygen`。如果先安装了 SSH1 然后又安装了 SSH2，那么 SSH2 的 `Makefile` 就会给这些文件（译注 1）加上一个后缀 `.old`，然后再创建新的符号链接指向自己的 SSH2 程序。例如，`ssh` 原来指向 `ssh1`；在安装完 SSH2 之后，`ssh` 指向 `ssh2`，而 `ssh.old` 指向 `ssh1`。这是很恰当的，因为系统认为 SSH2 比 SSH1 的版本新。

另一方面，如果先安装了 SSH2 又安装 SSH1，那么 SSH1 的 `Makefile` 就不会修改 SSH2 的链接。结果是 `ssh` 仍然指向 `ssh2`，而没有链接指向 `ssh1`。这和安装 SSH1 之后也要允许 SSH2 向后兼容 SSH1 的实际情况是一致的。

4.1.5 编译时配置

SSH1 和 SSH2 的编译似乎是相同的，不是吗？只需要输入 `configure` 和几个 `make` 命令就可以了。噢，别这么快下结论。在编译并安装一个新安全产品时，不应该盲目

译注 1：指那些指向 SSH1 的程序的符号链接。

地接受缺省选项。这些 SSH 产品有很多选项，用户可以在编译时进行设置，对每个选项都应该仔细考虑。我们称这个过程为编译时配置 (compile-time configuration)。

编译时配置是通过在编译发行版本之前运行一个名为 *configure* 的脚本来执行的 (注 1)。概括地说，*configure* 要完成两个任务：

- 检查本地计算机，设置各种计算机特定选项和操作系统特定的选项。例如，*configure* 要了解系统中可以使用哪些头文件和库文件，以及用户的 C 编译器是否是 ANSI 的。
- 包含或排除 SSH 源代码中的一些特性。例如，*configure* 可以保留或删除对 Kerberos 认证的支持。

我们只讨论第二个任务，因为它是 SSH 特有的；而且我们只介绍那些和 SSH 或安全性直接相关的配置选项。例如，我们不会介绍和编译器（例如，是否应该打印警告信息）或操作系统（例如，是否应该使用特定的 Unix 库函数）有关的那些标志。要查看所有的配置项标记，请输入：

```
$ configure -help
```

也可以阅读发行版本根目录下的 *README* 和 *INSTALL* 文件。

顺便说一下，SSH1 和 SSH2 的行为都可以在三个层次上进行控制。第一个层次是本章中介绍的编译时配置。另外，服务器范围的配置（第五章）可以控制正在运行的 SSH 服务器的全局设置，每账号配置（第八章）可以控制每个用户账号接受 SSH 链接所使用的设置。图 4-1 给出了编译时配置在整个配置中所处的地位。我们每次介绍一种新类型的配置时都会提醒读者回想一下这幅图。

4.1.5.1 配置标准

configure 脚本接收命令行标志来控制自己的操作，每个命令行标记前面都有一个双横线 (--)。这些标志有两类：

注 1： *configure* 脚本是使用一个免费的软件包 *autoconf* 创建的。在编译 SSH1 和 SSH2 时无需了解这一点，但是如果你想了解 *autoconf* 的支持，可以访问 GNU 的 Web 站点 <http://www.gnu.org/>。

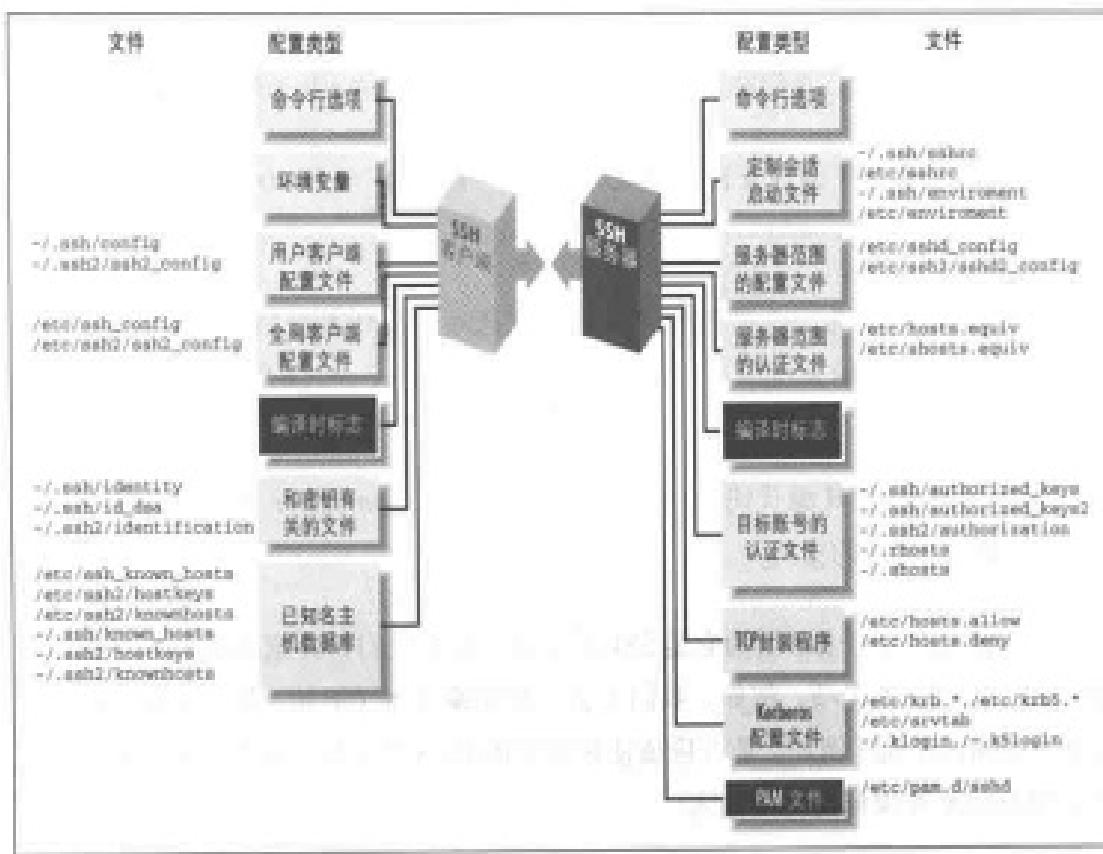


图 4-1: SSH 编译时配置 (高亮部分)

with/without 标志

编译时包含一个包。这些标记都以 `--with` 或 `--without` 开头。例如，要支持 X Window 系统可以使用 `--with-x` 标志，否则就使用 `--without-x` 标志。

enable/disable 标志

设置 SSH1 的缺省操作。这些标志以 `--enable` 或 `--disable` 开头。例如，SSH2 中的 X 转发特性可以使用 `--enable-X11-forwarding` 标志启用，或者使用 `--disable-X11-forwarding` 禁用。有些缺省操作可以在以后使用服务器范围配置或每账号配置覆盖。

以 `--with` 或 `--enable` 开头的标志也可以在后面跟上一个等号和一个字符串，例如：

```
--with-etcdir=/usr/local/etc
--enable-X11-forwarding=no
```

这里可以使用各种字符串，但通常都是 yes 或 no。对于一个给定的包 P ， $--with-P$ 和 $--with-P=yes$ 标志是等效的。下表给出了这两种表示法的关系：

如果输入：	则等效于：
$--with-P=yes$	$--with-P$
$--with-P=no$	$--without-P$

下面这个表给出了对于一个给定特性 F 的两种表示法的关系：

如果输入：	则等效于：
$--enable-F=yes$	$--enable-F$
$--enable-F=no$	$--disable-F$

在后文中，我们会给出使用不同命令行标志的 *configure* 的很多例子。大部分例子一次都只解释一个标志，因此我们使用下面的形式来表示命令行中还有其他标志：

```
$ configure ... --enable-fancy-feature ...
```

正确运行 *configure* 只有一次，就是在编译之前，并且要在同一个命令行中给出所有想要的标志。

警告： 在选择 *configure* 标志时一定要慎重，否则就可能浪费很多时间。*configure* 脚本并不很聪明，对用户的输入只执行很少的合法性检查，甚至根本就不执行这种检查。如果用户提供了一个无效的标志，*configure* 就可能会毫无知觉地运行几分钟，处理了其他很多配置选项，最后处理到错误的标志时就死掉了。这时就只能再重新运行这个脚本了。

还有，不要依赖于标志的缺省值，因为这些标志在不同的 SSH 实现上可能是不同的。为了最大化安全性控制，在运行 *configure* 时最好直接指定所有的标志。

4.1.5.2 安装、文件和目录

现在让我们开始讨论 *configure* 的和 SSH 有关的标志。首先，我们讨论有关文件的标志，这些标志可以用来选择安装目录、开启或关闭 setuid 位以及给文件和目录设置可写权限。

SSH 可执行文件被安装到您选定的目录（缺省是 `/usr/local`）下的 `/bin` 子目录中。这是通过 `configure` 标志 `--prefix` 指定的。例如，要把 `bin` 目录放在 `/usr/local/ssh` 中，并把可执行文件安装在 `/usr/local/ssh/bin` 目录中：

```
# SSH1, SSH2, OpenSSH
$ configure ... --prefix=/usr/local/ssh ...
```

有些和 SSH 有关的系统文件被安装到 `/etc` 目录中。这个缺省的位置可以使用 `configure` 标志 `--with-etcdir` 来修改，用户可以另外提供一个目录名（对于 OpenSSH 来说，这个标志是 `--sysconfdir`）：

```
# SSH1, SSH2
$ configure ... --with-etcdir=/usr/local/etc ...
```

`--with-etcdir` 标志在众多标志中极为特殊，因为没有相应的 `--without-etcdir` 标记。SSH1 和 SSH2 必须要指定安装目录，否则 `Makefile` 就不会编译。

然后，缺省情况下有些可执行文件都被安装并 `setuid` 成 `root`: `ssh1` (对于 SSH1) 和 `ssh-signer2` (对于 SSH2)。`ssh1` 需要被 `setuid` 成 `root` 才能执行可信主机认证（也就是各种基于主机的认证方法），原因如下：

- 要访问本地主机密钥，只有 `root` 用户才可以读取该文件。
- 要分配一个特权端口，这种操作也只有 `root` 才可以执行。

在 SSH2 中不再使用特权端口，第一种功能也已经被移植到了一个单独的程序 `ssh-signer2`，该程序对可信主机认证所使用的认证报文进行签名。如果没有把该程序 `setuid` 成 `root`，那么基于主机的认证都不能成功执行。[3.5.2.3]

SSH1 的 `ssh` 客户端可以使用 `configure` 标志 `--enable-suid-ssh` 和 `--disable-suid-ssh` 来控制是否 `setuid` 成 `root`:

```
# 仅对 SSH1
$ configure ... --disable-suid-ssh ...
```

同理，SSH2 的 `ssh-signer2` 的 `setuid` 位也可以使用 `--enable-suid-ssh-signer` 和 `--disable-suid-ssh-signer` 标志控制，例如：

```
# 仅对 SSH2
```

```
$ configure ... --disable-suid-ssh-signer ...
```

最后，SSH 服务器需要对用户账号中的特定的文件和目录（例如，*.rhosts* 文件和 *authorized_keys* 文件）设置特定的权限（注 2）。具体地说就是要去掉组可写权限和其他用户的可写权限。组可写权限在共享账号中非常有用（这样组成员就可以方便地修改账号的 SSH 文件）。这种限制可以使用 *configure* 的 *--enable-group-writeability* 标志来放宽（注 3）：

```
# SSH1, SSH2
$ configure ... --enable-group-writeability ...
```

现在服务器就允许 SSH 可以连接到那些对 SSH 文件具有组可写权限的账号上了。

4.1.5.3 TCP/IP 支持

SSH1 和 SSH2 的大部分 TCP/IP 特性都是通过服务器范围的配置进行控制的，[5.4.3] 但是有些也可以在编译时配置中使用。这些特性包括 TCP NODELAY 特性、TCP-wrapper、SO_LINGER socket 选项以及最大连接数的限制。

如果想在广域网而不是在快速以太网连接上运行 SSH，那么就可以考虑为 SSH 连接禁用 TCP/IP 的 NODELAY 特性，也就是 Nagle 算法。Nagle 算法减少了发送少量数据的 TCP 段（例如，会话末端的小字节流）的个数。用户可以使用 *--disable-tcp-nodelay* 标志在编译时禁用这个特性：

```
# SSH1, SSH2
$ configure ... --disable-tcp-nodelay ...
```

另外，也可以使用 NoDelay 配置关键字在服务器范围的配置中启用或禁用这种特性。[5.4.3.8]

TCP-wrapper 是根据到达的 TCP 连接的源地址进行控制访问的一种安全特性。[9.4] 例如，TCP-wrapper 可以通过查询 DNS 来验证发起连接的主机的身份，也可以拒绝来自给定地址、地址范围或 DNS 域的连接。虽然 SSH 已经包含了一些具有 AllowHosts、DenyHosts 等特性的控制方法，但是 TCP-wrapper 所提供的功能更

注 2： 只有在服务器上启用了 StrictModes 时才是如此。

注 3： 是的，就是“writeability”，虽然拼写不对。

完整。它允许使用目前在任何一种 SSH 都尚未实现的一些控制方法，例如，限制 X 连接的转发源。

如果在编译时我们指定了 `--with-libwrap` 标志并提供 wrapper 库 `libwrap.a` 的路径，那么 SSH1 和 SSH2 都可以支持 TCP-wrapper。

```
# SSH1, SSH2  
$ configure ... --with-libwrap=/usr/local/lib ...
```

如果安装的 Unix 中没有 TCP-wrapper 库，那么可以从这里自行下载并编译：

<ftp://ftp.porcupine.org/pub/security/index.html>

要了解有关 TCP-wrapper 的更多信息，请阅读 `tcpd` 和 `hosts_access` 的手册页。

SSH1 使用一个更低级的选项来控制 `SO_LINGER` socket 标志，该标志在编译时可以启用，也可以禁用。假设 SSH1 正在使用一个打开的套接字进行通信，当数据仍在排队时这个套接字就关闭了。此时数据会怎样呢？`SO_LINGER` 标志的设置确定了要发生的操作。当该标志被启用时，socket 就“延迟”关闭，直到数据被转发出去或经过指定的超时时间之后才关闭。这个标志的使用需要清楚了解 TCP/IP 和 socket 操作的详细知识，因此如果用户清楚自己在干嘛，就使用 `--enable-so-linger` 标志好了：

```
# 仅对 SSH1  
$ configure ... --enable-so-linger ...
```

最后，用户可以通知 `sshd2` 限制可以支持的最大并发连接数。缺省情况下它可以接收的并发连接数没有限制，但是如果想对服务器上的资源进行限制保护的话，也可以设置一个上限。正确的标志是 `--with-ssh-connection-limit` 后面跟上一个非负整数；例如：

```
# 仅对 SSH2  
$ configure ... --with-ssh-connection-limit=50 ...
```

在运行时可以使用服务器范围的配置关键字 `MaxConnections` 来覆盖该值。[5.4.3.6]

4.1.5.4 X Window 的支持

如果想使用 SSH 在两个运行 X Window 的主机之间进行通信，必须确保在编译时包含 X 的支持。（缺省情况下，是包含对 X 的支持。）反之，如果永远都不会使用 X，那么就可以不用这种支持，从而节省磁盘空间。用户可以根据需要使用 `--with-x` 或 `--without-x` 标志：

```
# SSH1, SSH2
$ configure ... --without-x ...
```

对于那些明确要求彻底删除 X 用户来说，*configure* 还有几个其他更有用的有关 X 的标志。具体地说，用户可以启用或禁用对 X 转发的支持，它可以允许 SSH 服务器上打开的 X 应用程序显示在 SSH 客户端机器上。[9.3]

对于 SSH1 来说，X 转发支持可以由 SSH 客户端和服务器分别加以控制：

```
# 仅对 SSH1
$ configure ... --disable-server-x11-forwarding ...
$ configure ... --disable-client-x11-forwarding ...
```

对于 SSH2 来说，X 转发支持是由编译时标志 `--enable-x11-forwarding`（或 `--disable-x11-forwarding`）作为一个整体进行控制的：

```
# 仅对 SSH2
$ configure ... --disable-x11-forwarding ...
```

要记住，这些 enable/disable 标志只能设置 SSH 的缺省操作。以后 X 转发还可以通过服务器范围的配置使用 `X11Forwarding`（SSH1、OpenSSH）或 `ForwardX11`（SSH2）配置关键字启用或禁用。[9.3.3]

4.1.5.5 TCP 端口转发

端口转发使 SSH 能够对使用基于 TCP/IP 的程序传输的数据进行加密。[9.2] 如果需要，可以在编译时禁用这种特性。X 窗口转发不会受到这些通用端口转发标志的影响。

在 SSH1 中，端口转发可以在服务器上禁用，也可以在客户端上禁用，还可以两者都禁用。要在 SSH1 服务器上禁用端口转发，就使用 *configure* 标志 `--disable-server-port-forwarding`。同理，要在 SSH1 客户端上禁用端口转发，就使用

configure 标志 `--disable-client-port-forwarding`。缺省情况下，端口转发在编译时是启用的。

在 SSH2 中，对端口转发的支持并不区分客户端和服务器而单独进行控制。*configure* 标志 `--enable-tcp-port-forwarding` 和 `--disable-tcp-port-forwarding` 分别启用和禁用这种特性。

4.1.5.6 加密和密码

SSH1 在编译时可以支持特定的加密算法，例如，IDEA、Blowfish、DES 和 ARCFour；也可以不支持任何加密算法。（在 SSH2 中，这种支持是在服务器范围的配置中使用 `Ciphers` 关键字进行控制的。[5.4.5]）包含这种支持的标志有：

`--with-idea`

包含 IDEA 算法

`--with-blowfish`

包含 Blowfish 算法

`--with-des`

包含 DES 算法

`--with-arcfour`

包含 ARCFour 算法

`--with-none`

允许不加密传输

要排除这些支持，就使用 `--without` 形式的标记：

```
# 仅对 SSH1
$ configure ... --without-blowfish ...
```

警告： 我们推荐使用 `--without-none` 标志来禁止不加密传输。否则，闯入服务器的入侵者就可以在配置文件中加入一行（“Ciphers None”）来关闭客户端使用 SSH 加密。这样做还可能引起其他安全漏洞。[5.4.5] 如果需要使用不加密传输进行测试，就使用 `--with-none` 标志另外构建一个 SSH1 服务器，并将其设置为只有系统管理员可以执行。还要结合 SSH-1 协议考虑，关闭加密并不仅仅消除了数据的隐私性，而且也导致服务器认证和数据完整性都无效了。

有些 SSH 实现包含了 RSA 加密算法用于公钥认证。[3.9.1.1]在本书印刷时，有很多实现就不采用 RSA 加密算法，因为直到 2000 年 9 月它还一直受到专利的保护；现在专利已经过期，RSA 可以用于公共用途了。在专利期限之内，该公司提供了 RSA 的一种“参考实现”，称为 RSAREF，它可以免费用于教育目的和非商业目的，而不会侵犯专利权。我们猜想 RSAREF 最终会被人们舍弃不用，因为现在已经有更通用的 RSA 实现可以免费使用了。而且，我们并不鼓励使用 RSAREF，因为它中间有安全漏洞，公司也不再对其提供支持了。然而，用户仍然可以使用 *configure* 标志 *--with-rsaref* 通知 SSH1 使用 RSAREF 而不使用它自己的 RSA 实现：

```
# 仅对 SSH1  
$ configure ... --with-rsaref ...
```

然后把 RSAREF 解包到 SSH1 实现顶层的一个名为 *rsaref2* 的目录中。缺省情况下使用了 RSA 加密，用户也可以在 *configure* 中指定 *--without-rsaref* 标志（这里没有 *--with-rsaref* 标志）。要了解有关 RSAREF 的更多信息，请访问 <http://www.rsa.com/>。

4.1.5.7 认证

SSH1 和 SSH2 在编译时都可以支持几种认证方法。对于 SSH1 来说，可用的认证方法有 Kerberos、SecurID 以及 TIS (Trusted Information Systems，可靠信息系统) 的 Gauntlet 工具包。SSH2 可以支持使用 OpenPGP 密钥的认证（注 4）。在 SSH 2.3.0 中还包含了支持 Kerberos-5 的实验性代码，但是现在还不支持 Kerberos-5，其定义也尚未加入 SSH-2 草案标准中。

Kerberos[11.4]是一种比较特殊的认证机制，它不使用用户口令进行认证，而是传递证书（ticket）进行认证；证书是一个具有一定生命周期的小字节序列。配置标志 *--with-kerberos5* 和 *--without-kerberos5* 可以控制在编译 SSH 时是否包含 Kerberos 的支持（注 5）。*--with-kerberos5* 标志在使用时可以在后面跟上一个字符串来指定包含 Kerberos 文件的目录。

注 4： SSH2 的 *configure* 脚本可以识别所有与 SecurID 和 Gauntlet 有关的标志，但是在本书付印时的源代码中都尚不支持这种特性。

注 5： 编译时不要为 SSH1 1.2.27 或更早的版本包含 Kerberos 的支持，因为这些版本中有一个十分严重的 bug。[11.4.4.5]如果想使用 Kerberos，应该使用 1.2.28 或更新的版本。OpenSSH 则没有这个 bug。

```
# 仅对 SSH1
$ configure ... --with-kerberos5=/usr/kerberos ...
```

如果 `--with-kerberos5` 标志中没有给出目录名，那么就使用缺省位置 `/usr/local`。另外，Kerberos 可更新证书（ticket-granting ticket）的特性可以使用 `--enable-kerberos-tgt-passing` 标志启用，这也是缺省选项：

```
# 仅对 SSH1
$ configure ... --enable-kerberos-tgt-passing ...
```

`SecurID` 也是比较特殊的一种认证机制：用户需要携带一些电子卡片，大小和信用卡差不多，这些卡片上面显示一些整数数字，而且会随机变化。在认证时，用户除了要输入用户名和密码之外，还要输入卡片上现在显示的数字。

要在编译时增加 `SecurID` 的支持，就要使用 `--with-securid` 标志，并在后面提供包含 `SecurID` 头文件和库文件的目录：

```
# 仅对 SSH1
$ configure ... --with-securid=/usr/ace ...
```

`Gauntlet` 是一个防火墙工具包，其中包含了一个认证服务器程序 `authserv`。如果正在运行 `Gauntlet` 并想让 `SSH1` 和它的认证服务器通信，就可以使用 `--with-tis` 标志并提供本地 `Gauntlet` 目录的路径来启用这种特性：

```
# SSH1, SSH2
$ configure ... --with-tis=/usr/local/gauntlet ...
```

`PGP` 是在很多计算平台上都可以使用的一种通用的加密认证程序。[1.6.2]SSH2 也可以根据 `PGP` 密钥对用户进行认证，条件是这些密钥必须符合 `OpenPGP` 标准（`RFC2440`, “`OpenPGP 消息格式`”；有些 `PGP` 版本，特别是一些老版本，可能不兼容 `OpenPGP` 格式）。要包含这种支持，就得使用 `--with-pgp` 标记进行编译：

```
# 仅对 SSH2
$ configure ... --with-pgp ...
```

4.1.5.8 SOCKS 代理支持

`SOCKS` 是代理使用的一种网络协议。代理是一个软件组件，它可以使用户伪装成另外一种身份，从而达到隐藏或保护的目的。例如，假设某个公司允许其员工在网上冲浪，但是却不想把内部机器名暴露到公司之外。那么他们就可以在内部网络和

Internet 之间加入一个代理服务器，这样所有的 Web 请求看起来都是从这个代理服务器发出的。另外，代理可以防止那些不想要的信息进入内部网络，其功能类似于防火墙。

SSH1 和 SSH2 都可以支持 SOCKS，也就是说它们可以创建经过 SOCKS 代理服务器的连接。对于 SSH1 来说，这种支持可以在编译时启用，它可以处理 SOCKS4 或 SOCKS5 协议。SSH2 只能支持 SOCKS4，而且这种支持是内建的，任何时候都可以使用（它没有外部的库文件，也不需要指定编译选项来启用）。

SSH1 使用外部库来支持 SOCKS，因此要想在 SSH1 中使用 SOCKS，那么在编译 SSH1 之前必须安装这样的库。我们在实验时使用的是从 NEC 网络系统实验室 (<http://www.socks.nec.com/>) 下载的 SOCKS5 包（注 6）。

SSH1 有关 SOCKS 的 *configure* 选项有三个：

--with-socks4

 使用 SOCKS4

--with-socks5

 使用 SOCKS5

--with-socks

 使用 SOCKS5 或 SOCKS4，如果两个都可以使用就使用 SOCKS5

SSH2 的 SOCKS 功能是由 SocksServer 客户端配置选项控制的。[7.4.6]除了在配置文件或在命令行中指定 *-o* 选项这些常用方法之外，还可以使用 SSH_SOCKS_SERVER 环境变量进行设置。

SocksServer 缺省值为 0，也就是 SSH2 假设没有 SOCKS 服务器。配置标志：

--with-socks-server=*string*

给该参数设置一个非 0 的缺省值，这可以让用户在安装 SSH2 时假定有一个 SOCKS 服务器。注意这里和在客户端的全局配置文件中直接使用 SocksServer 的用法不

注 6：NEC 的 SOCKS5 参考实现的许可证仅仅对“非商业目的使用免费，例如，用于学术界、研究领域和内部商业”。许可证的全文可以在它们的 Web 站点上找到。

同，因为配置参数总会覆盖环境变量的值。如果你使用编译选项，那么用户就可以使用SSH SOCKS SERVER另外指定一个SOCKS服务器；如果你使用了全局文件，那么用户就无能为力了（尽管他们还可以直接使用自己的 SocksServer 来覆盖该值）。

有关 SOCKS 的更多信息请参看 SSH SOCKS 支持的工作原理的详细讨论以及 <http://www.socks.nec.com/>。

4.1.5.9 用户登录和 shell

用户登录和shell的一些内容也可以在编译时配置中进行控制。可以使用自行定制的登录程序来代替 /bin/login，并修改缺省的用户搜索路径。

当用户使用 ssh 或 slogin 登录到一台远程机器上时，远程 SSH 服务器就执行一个进程来完成这种登录。缺省情况下，SSH1 服务器运行一个登录 shell。实际上服务器可以运行选定的登录程序，可以是 /bin/login（缺省），也可以是其他用户选定的程序，例如 Kerberos 登录程序，或者对 /bin/login 进行修改增加了其他特性的程序。

其他登录程序的选择是在编译时使用 *configure* 标志 --with-login 并提供登录程序的路径来确定的：

```
# 仅对 SSH1
$ configure ... --with-login=/usr/local/bin/my-login ...
```

用户选用的登录程序必须要支持和 /bin/login 相同的命令行标志，包括 -h（指定主机名）、-p（向登录 shell 传递环境变量）和 -f（强制登录而不检查密码）。这是因为 sshd1 就是在这样命令行的登录程序基础上繁衍而来的：

```
name_of_login_program -h hostname -p -f --username
```

如果用户指定了 --with-login 标志并想使用自己选择的登录程序，那么还必须在服务器范围的配置中开启 UseLogin 关键字：[5.5.3]

```
# SSH1 服务器端配置文件中的关键字
UseLogin yes
```

登录程序要执行一些有用的操作，例如设置用户的缺省搜索路径。如果 sshd1 不调

用登录程序（例如，在编译时使用了`--without-login`标志），那么你可以告诉它为 SSH 会话设置缺省搜索路径。这可以使用编译标志`--with-path`实现：

```
# 仅对 SSH1
$ configure ... --with-path="/usr/bin:/usr/local/bin:/usr/mine/bin" ...
```

如果不指定`--with-path`，而且 Unix 环境也没有提供缺省路径，那么`sshd1`缺省将其设置为：

```
PATH="/bin:/usr/bin:/usr/ucb:/usr/bin/X11:/usr/local/bin"
```

4.1.5.10 禁止登录

`/etc/nologin`文件对很多 Unix 版本都有特殊的意义。如果该文件存在，那么所有的用户都被禁止登录。`sshd`也遵循这一文件的规定。但是，用户可以告诉`sshd1`绕开`/etc/nologin`文件，从而允许指定的用户登录。这可以通过创建另外一个文件来存放这些用户实现，例如，`/etc/nologin.allow`：该文件中存放的是即使`/etc/nologin`文件存在也允许登录的用户名。例如，把系统管理员的用户名放入`/etc/nologin.allow`是个好主意，这样就可以防止系统管理员不能登录到机器上的情况发生。还必须使用`configure`选项`--with-nologin-allow`来启用这种特性，该标志后面应该给出这个特例文件的路径：

```
# 仅对 SSH1
$ configure ... --with-nologin-allow=/etc/nologin.allow ...
```

4.1.5.11 scp 的行为

安全拷贝的客户端可以显示自己进度的统计信息。在文件通过网络拷贝的过程中，`scp`可以显示出文件已经传输的百分比。SSH1 发行版本有几个与统计信息有关的`configure`标志。有一对标志可以控制是否把统计信息代码编译到`scp`中，还有一对可以控制`scp`显示统计信息的缺省行为。

`--with-scp-stats`和`--without-scp-stats`标志控制`scp`是否包含统计代码。缺省情况下是包含这些代码的。要禁止`scp`包含这些代码：

```
# 仅对 SSH1
$ configure ... --without-scp-stats ...
```

如果包含了统计代码，还有其他配置标志可以控制 *scp* 显示统计信息的缺省方式。*--enable-scp-stats* 和 *--disable-scp-stats* 标志设置单个文件传输的缺省行为。如果这两个标志都没有使用，那么缺省情况下也是启用统计信息的。要禁用这种特性，可以这样：

```
# 仅对 SSH1
$ configure ... --disable-scp-stats ...
```

不管是对单个文件传输进行配置还是对多个文件传输进行配置，统计信息必须使用 *scp* 的命令行选项（*-Q* 和 *-a*）以及用户环境变量（*SSH SCP STATS*、*SSH NO SCP STATS*、*SSH ALL SCP STATS* 和 *SSH NO ALL SCP STATS*）来开启或关闭。^[7.5.7]当然，统计信息的代码必须在运行时已经配置为可用了（*--with-scp-stats*）。

4.1.5.12 R- 命令 (*rsh*) 的兼容性

在 SSH1 和 OpenSSH 中，如果 *ssh* 不能获得一个安全连接连往远程主机，那么它也可以使用 r- 命令 (*rsh*、*rcp*、*rlogin*) 建立一条不安全的连接。这种特性对于向后兼容是很有用的，但是不符合安全设置的要求。SSH2 明确规定不包含这种不安全特性。

SSH1 的 *configure* 标志 *--with-rsh* 和 *--without-rsh* 确定 *ssh* 是否可以使用 *rsh* 建立连接。要允许使用 *rsh*，就要在该标志后面提供 r- 命令可执行文件的路径：

```
# SSH1, OpenSSH
$ configure ... --with-rsh=/usr/ucb/rsh ...
```

如果你包含了 *rsh* 的支持，每个用户都可以使用关键字 *FallbackToRsh* 和 *UserRsh* 在自己的账号中自行控制。^[7.4.5.8]或者，你可以完全禁止在 *ssh* 中使用 *rsh*，那就要这样编译：

```
# SSH1, OpenSSH
$ configure ... --without-rsh ...
```

4.1.5.13 SSH-1/SSH-2 代理兼容性

使用 SSH-1 和 SSH-2 的代理^[2.5]通常都是不兼容的。也就是说，这两种代理都不

能存储另一个版本代理的密钥，也不能转发来自另一个版本的连接。[6.3.2.4]但是，SSH2 代理有一种可选特性，在三种特殊情况下可以处理 SSH-1 协议的应用程序：

- SSH2 实现中必须包含 RSA 的支持，因为 SSH1 使用 RSA 对密钥进行加密。在本书付印时，F-Secure SSH2 服务器提供了 RSA 的支持，但是 SSH2 没有。
- SSH2 的 *configure* 脚本必须是使用 *--with-ssh-agent1-compat* 标志运行的：

```
# 仅对 SSH2
$ configure ... --with-ssh-agent1-compat ...
```

- SSH2 代理 *ssh-agent2* 必须是使用命令行标志 *-I*（是数字 1，而不是小写的 L）运行的：

```
# 仅对 SSH2
$ ssh-agent2 -1
```

4.1.5.14 调试输出

SSH 服务器会根据需要产生详细的调试输出。[5.8]在编译时，用户可以启用不同级别的调试，也可以包含对 Electric Fence 内存分配调试器的支持。

如果需要，SSH2 服务器可以使用两级调试输出，也可以不使用。如果不使用调试代码，程序的性能可能会稍微有些提高；但是使用了调试代码，程序更容易维护。我们推荐至少包含一些调试代码，因为用户不可能预见什么时候会需要诊断问题。

“轻”调试和“重”调试是在源代码中可以指定的两个调试级别。轻调试输出是由 *configure* 标志 *--enable-debug* 和 *--disable-debug*（缺省）控制的。而重调试输出是由 *configure* 标志 *--enable-debug-heavy* 和 *--disable-debug-heavy*（缺省）控制的。例如：

```
# 仅对 SSH2
$ configure ... --enable-debug --disable-debug-heavy ...
```

这两级调试并不是互斥的：可以选用轻调试或重调试，也可以二者都选，当然也可以一个都不选。我们建议打开重调试；否则产生的消息的内容会太少而不够用。

最后，SSH2 内存分配可以使用 Electric Fence 进行跟踪，它是 Pixar 的 Bruce Perens 编写的一个免费的内存分配调试器。要使用它，必须在服务器上安装 Electric Fence。

configure 标志 `--enable-efence` 和 `--disable-efence` (缺省) 控制是否使用 Electric Fence:

```
# 仅对 SSH2  
$ configure ... --enable-efence ...
```

该标志让 SSH2 的程序连接到 Electric Fence 库 *libefence.a* 上, 该库提供了修改过的 `malloc()`、`free()` 以及其他一些与内存有关的函数。Electric Fence 可以在如下网址找到:

<http://sources.isc.org-devel/memleak/>

4.1.6 创建服务器范围的已知名主机文件

在一台主机上配置并安装好 SSH1 之后, 就应该创建服务器范围的已知名主机文件 [2.3.1]了。这个文件通常都是 */etc/ssh_known_hosts*, 它包含了本地域中所有机器以及该域中的用户使用 SSH1 经常连往的所有远程主机的公共主机密钥。例如, *myhost.example.com* 上的已知名主机文件可能包含了 *example.com* 域中的所有机器的主机密钥, 还可能含有其他主机密钥。

如果把 SSH 客户端配置成把新主机密钥加入用户自己的已知名主机文件中, 那么你可以不维护这个文件。[7.4.3.1]但是, 最好尽可能地把通用的主机放入这个集中控制的文件中, 这是由于以下的原因:

- 方便用户, 避免老是提示增加密钥。
- 更安全。当接收一个新的 SSH 服务器密钥时, 就为中间人攻击打开了大门。[3.10.4]如果我们能预先知道远程主机的密钥, 当入侵者伪装成一个远程主机时, SSH 客户端就可以判断出这是一个假密钥。

已知名主机文件主要用于可信主机认证。[3.4.2]只有那些从该文件中包含的主机上连过来的用户才能使用这种方法进行认证。

用户可以手工搜集主机密钥, 这可以在机器上安装 SSH 时进行, 也可以在安装好 SSH 之后进行。但是如果有很多主机, SSH1 另外给出了一个工具来帮助搜集主机密钥: *make-ssh-known-hosts*。这个 Perl 脚本可以查询域名服务器 (DNS) 从而找

到本地域中的所有主机名，并使用 SSH 连接到这些主机上来获得这些主机的主机密钥。然后把这些密钥写入标准输出，以供包含进已知名文件中。

在这种最简单的形式中，该程序可以使用一个参数调用，也就是本地域名：

```
# 仅对 SSH1  
$ make-ssh-known-hosts example.com > /etc/ssh_known_hosts
```

make-ssh-known-hosts 有很多命令行标志可以用来定制自己的行为。[\[4.1.6.1\]](#) 另外，可以在域名之后使用 Perl 风格的规则表达式作为参数来限制所查询的机器。例如，要显示 *example.com* 域中所有主机名以 z 开头的主机密钥，可以这样使用：

```
$ make-ssh-known-hosts example.com '^z'
```

第二个规则表达式参数执行相反的任务：它不包含匹配该规则表达式的主机密钥。用户可以对前面的例子进行扩充，使其不包含以 x 结尾的主机：

```
$ make-ssh-known-hosts example.com '^z' 'x$'
```

开个玩笑，下面这个命令不会产生任何主机密钥：

```
$ make-ssh-known-hosts example.com mymachine mymachine
```

因为它包含的内容和排斥的内容完全相同。

4.1.6.1 *make-ssh-known-hosts* 命令行标志

make-ssh-known-hosts 的命令行标志有两种形式，现在我们分别加以讨论：

- 双短横线后面跟上一个完整的单词，例如，`--passwordtimeout`。
- 单短横线后面跟上一个单词的缩写形式，例如，`-pa`。

下面这些标志都是和程序位置有关的：

```
--nslookup (-n) path
```

通知脚本要执行的 *nslookup* 的完整路径，*nslookup* 是执行 DNS 查询的程序。缺省值是在 shell 的当前搜索路径中定位 *nslookup*。

--ssh (-ss) path

通知脚本 SSH 客户端的完整路径。用户也可以在这儿给 *ssh* 提供一个命令行选项。缺省值是在 shell 的当前搜索路径中定位 *ssh*。

以下的标志都是和超时有关的：

--passwordtimeout (-pa) timeout

要等待用户输入密码多长时间，单位是秒。缺省值是不提示输入密码。该值为 0 表示提示用户输入密码，而禁用超时特性。

--pingtimeout (-pi) timeout

使用 ping 命令时可以等待主机 SSH 端口的响应多长时间，单位是秒。缺省值是 3 秒。

--timeout (-ti) timeout

可以等待 SSH 命令执行完多长时间，单位是秒。缺省值是 60 秒。

以下这些标记和域有关：

--initialdns (-i) nameserver

要查询的主名字服务器；否则就使用解析列表。第一次查询对 *make-ssh-known-hosts* 的域名参数的区域 SOA 记录进行。然后从 SOA 记录中给出的主名字服务器中执行区域转换。

--server (-se) nameserver

如果给出了这个标志，就跳过 SOA 记录查询，立即从名字服务器中执行区域转换。

--subdomains (-su) domain1, domain2, ...

通常 *make-ssh-known-hosts* 都使用域名缩写为每个主机给出一些别名，域名缩写的形式为从最左边到最右边遍历域名，但是不包括倒数第二个。例如，主机 *foo.bar.baz.geewhiz.edu* 就可以得出以下的别名：

foo

foo.bar

foo.bar.baz

foo.bar.baz.geewhiz.edu

该选项可以让用户选择只包括这些域的一个子集，而不是包括所有的域。

--domainnamesplit (-do)

通过把域名分割成前缀的形式来为输出中的每个主机密钥创建别名。例如，域名 *a.b.c* 可以分割成前缀 *a*、*a.b* 和 *a.b.c*，每个前缀都附加在主机名后面来创建别名。

--norecursive (-nor)

只获得所给出的域的密钥，而且不会递归其子域。缺省情况下要检查子域。

以下的标志和输出及调试有关：

--debug (-de) level

指定一个非负整数的调试级别。级别越高，生成的调试输出越多。缺省值是 5。到本书截稿为止，*make-ssh-known-hosts* 中使用的最高调试级别是 80。

--silent (-si)

结束时扬声器不会响。缺省情况下会响。

--keyscan (-k)

以 *ssh-keyscan* 使用的一种格式显示结果，*ssh-keyscan* 是用来搜集 SSH 公钥的程序。*ssh-keyscan* 是单独一个程序，而不是 SSH1 的一部分。[13.4]

最后，还有一个和故障恢复有关的标记：

--notrustdaemon (-notr)

make-ssh-known-hosts 调用 *ssh host cat /etc/ssh_host_key.pub* 来获得一个主机的公钥。如果该命令由于某些原因（例如，密钥文件存在别的地方）没能成功执行，那么 SSH 仍可以使用 SSH 协议获得密钥，并将其保存在用户的 *~/.ssh/known_hosts* 文件中。通常 *make-ssh-known-hosts* 都是使用这个密钥；使用 --notrustdaemon 标志，可以把这个密钥包含进来，但是要将其注释掉。

4.2 F-Secure SSH 服务器

F-Secure Corporation，前身是 DataFellows, Ltd.，一家芬兰的软件公司，开发了 SSH 的一种商业实现版本，它源自于 SSH Communications Security 公司的实现版

本。F-Secure 服务器产品线——F-Secure 服务器可以在 Unix 上运行，而 SSH-1 和 SSH-2 服务器是另外可以单独使用的产品。这些产品都对 SSH1 和 SSH2 重新进行了打包，并加入了商业许可证和一些特性：

- F-Secure SSH 产品在所有平台（Unix、Windows、Macintosh）上的手册。
- 在 SSH-2 中加入了其他加密算法，例如，RSA 和 IDEA（请参看 F-Secure 现在的手册）。
- 另外一个 SSH 客户端 *edd*（加密数据转换），它是一个 Unix 过滤器，对标准输入进行加密或解密，并将结果写入标准输出。
- SSH1 中的其他选项（请参看附录二）。

4.2.1 F-Secure 的获取和安装

F-Secure 可以从 <http://www.f-secure.com/> 获得。该公司的商业 SSH 产品可以在这个站点上购买并下载，另外免费的“评估版”也可以在这儿下载。

除了上一节中已经介绍的一些其他特性之外，F-Secure Unix SSH 的安装、配置和操作和 SCS 版本几乎完全相同。SSH2 已经增加了 F-Secure 所没有的一些新特性，因此需要检查 F-Secure 文档，以便确认这些特性是否可用。

4.3 OpenSSH

OpenSSH 是 SSH-1 和 SSH-2 的一种免费实现，可以从 OpenSSH 的网站上下载：

<http://www.openssh.com>

由于 OpenSSH 是由 OpenBSD 项目开发的，因此 OpenSSH 的主要版本都是专门供 OpenBSD Unix 操作系统使用的，事实上在 OpenBSD 的基本安装中也都包含了 OpenSSH。同时还有一个团队在做一些独立的相关工作，它们维护一个“可移植的”版本，使其可以在各种 Unix 风格的操作系统上编译，并紧随主开发版本而不断更新。所支持的平台包括 Linux、Solaris、AIX、IRIX、HP/UX、FreeBSD 以及 NetBSD（FreeBSD 中也包含了 OpenSSH）。可移植版本后都跟上一个“p”后缀。例如，2.1.1p4 就是 OpenSSH 2.1.1 的可移植版本的第四个发行版。

4.3.1 前提条件

OpenSSH 要依赖于另外两个软件包： OpenSSL 和 zlib。 OpenSSL 是一个加密库，可以在 <http://www.openssl.com/> 找到； OpenSSH 中所有的加密都是通过 OpenSSL 实现的。 zlib 是一个数据压缩程序库，可以在 <http://www.info-zip.org/pub/infozip/zlib/> 找到。在编译 OpenSSH 之前，必须获得并安装这些包。

4.3.2 编译

OpenSSH 的编译和 SSH1 及 SSH2 类似，都是使用相同的 *configure;make;make install* 命令。但是在 2.2.0 之前的一些 OpenSSH 版本中，*make install* 不能自动创建并安装主机密钥。如果你的主机密钥不见了，你可以使用 *make host-key* 来安装。

4.3.3 PAM

缺省情况下，OpenSSH 使用 PAM 进行密码认证。PAM 的全称是可插入认证模块 (Pluggable Authentication Modules) 系统，它是认证、授权和账号管理 (AAA) 使用的一种通用框架。其思想是让程序通过动态加载库来调用 PAM 来执行 AAA 函数，让系统管理员自由配置每个程序使用不同种类的认证。要了解有关 PAM 的更多信息，请访问 <http://www.kernel.org/pub/linux/libs/pam/>。

通常，如果一个程序使用了 PAM，那么就需要在主机上进行一些配置，说明让 PAM 应该为这个程序执行些什么操作。PAM 的配置文件通常都在 */etc/pam.d* 目录中。

警告： 在很多使用 PAM 的操作系统中（包括 RedHat Linux），OpenSSH 缺省情况下的编译都包含了 PAM 的支持（可以使用 *configure* 标志 *--without-pam* 关闭这个支持）。但是，还必须对 PAM 进行配置，从而让其了解到 *sshd* 使用 PAM，否则密码认证就不能工作。缺省情况下，对于那些没有明确配置要使用 PAM 的程序，PAM 通常禁止为其进行认证。

为 SSH 而配置 PAM 通常都只需要把适当的 *sshd.pam* 文件从发行版本的 *contrib* 目录中拷贝到 */etc/pam.d/sshd* 中。各种风格的 Unix 中都包含有样例文件。

注意当修改完 PAM 配置时并不需要重新启动 *sshd*；每次使用 PAM 时都会检查配置文件。

4.3.4 随机性

OpenSSH的主要代码依靠主机操作系统来提供一些随机性,这是使用一个设备驱动程序访问`/dev/urandom`实现的。这是因为OpenBSD的操作系统有这种设备。如果在诸如Solaris之类的没有这种设备的平台上编译OpenSSH,那么就需要使用其他手段来提供一些随机性。此时有两种选择:

- 使用内建的“internal entropy-gathering”系统。
- 安装EGD(Entropy Gathering Daemon)包(<http://www.lothar.com/tech/crypto/>)。

除非用户将其配置成使用EGD,否则OpenSSH缺省地选用第一种方法,也就是内建系统。内建系统使用一些可以配置的命令,这些命令可以监视系统操作的各种变化,并将输出结果合并在一起。用户可以使用文件`/etc/ssh_prng_cmds`控制使用哪些命令,以及如何使用这些命令。

4.3.5 编译标志

和其他SSH实现一样,OpenSSH也有一些编译标志,有些是和SSH相同的,有些则不同。下面我们给出一些重要的标志:

--without-pam

禁用PAM支持

在OpenSSH中忽略对PAM的支持。这个标志通常都没有必要使用,因为`configure`进程会确定主机上是否有PAM;如果有,就很可能使用PAM。

--with-md5-passwords

启用MD5密码

--without-shadow

禁用shadow密码的支持

这些选项控制OpenSSH对Unix账号数据库的操作(`passwd`映射)。只有当OpenSSH没有使用PAM时,这两个标志才有关系,因为否则就是由Pam读取账号信息,而不是OpenSSH。

如果系统使用了MD5而不是传统的加密函数来对密码进行散列操作,而且没有使用PAM,那么就应该启用`--with-md5-passwords`。

“shadow passwords”是指我们习惯于把散列过的密码保存在受限的文件`/etc/`

shadow 中 (*/etc/passwd* 必须是所有用户都可以读取的)。使用 `--without-shadow` 来禁止读取 */etc/shadow* 文件，这应该是必须的。

`--with-ssl-dir=PATH` 设置 OpenSSL 的安装路径
如果 OpenSSL 不是安装在通常的 */usr/local/ssl* 目录中，就使用这个标志来指明所安装的位置。

`--with-xauth=PATH` 设置 *xauth* 程序的路径
在 OpenSSH 中，*xauth* 程序的缺省位置是一个编译时参数。

`--with-random=FILE` 从指定文件中读取随机数
指定提供随机源的字符设备文件，通常是 */dev/urandom*。

`--with-egd-pool=FILE` 从 EGD 缓冲池 FILE 中读取随机数（缺省为没有）
如果是像前面我们所介绍的那样安装的 EGD，就要使用这个标志让 OpenSSH 使用 EGD 作为自己的随机数源。

`--with-kerberos4=PATH` 启用 Kerberos-4 支持
`--with-afs=PATH` 启用 AFS 支持
这两个标志适用于 Kerberos-4 和 AFS。[3.4.2.4] 注意在 OpenSSH 中不支持 Kerberos-5。

`--with-skey` 启用 S/Key 支持
启用对密码认证使用的 S/Key 的一次密码系统的支持。[3.4.2.5]

`--with-tcp-wrappers` 启用 TCP-wrappers 支持
和 SSH1 的 *configure* 标志 `--with-libwrap` 相同。[4.1.5.3]

`--with-ipaddr-display` 在 \$DISPLAY 中使用 IP 地址，而不是使用主机名
在 X 转发中，使用 192.168.10.1:10.0 的形式来表示 DISPLAY 的值，而不是使用 hostname:10.0 的形式。该标志在一些 X 库中非常有用，这些库使用主机名时有些 bug，它们使用 IPC 机制和 X 服务器通信，而不是使用 TCP 机制。

`--with-default-path=PATH` 缺省的服务器路径
OpenSSH 在运行子程序时所使用的缺省路径。

`--with-ipv4-default` 使用 IPv4，除非给出了“-6”标志
`--with-4in6` 对 IPv4 进行检查并将其转换成 Ipv6 的映射地址
OpenSSH 支持 IPv6，Ipv6 是下一代协议族，目前仍处于开发阶段，在 Internet

上的使用也处于开始阶段(现在IP的版本是IPv4)。OpenSSH缺省的配置是尽可能使用IPv6,有时这样会有问题。如果碰到提示“af=10”或“address family 10”的错误,那就是由于IPv6所导致的,此时应该试一下-4运行时选项,或者--with-ipv4-default编译标志。

--with-pid-dir=PATH 指定 ssh.pid 文件的位置

OpenSSH 的 pid 文件的位置,此处存储了现在正在运行的守护进程的 pid。缺省值是 /var/run/sshd.pid。

4.4 软件清单

表 4-1 给出了 SSH 所安装的文件和程序清单。

表 4-1: 软件清单

内容	SSH1	OpenSSH	SSH2
服务器配置	/etc/sshd_config	/etc/sshd_config	/etc/ssh2/sshd2_config
客户端全局配置	/etc/ssh_config	/etc/ssh_config	/etc/ssh2/ssh2_config
主机私钥	/etc/ssh_host_key	/etc/ssh_host_dsa_key	/etc/ssh2/hostkey
主机公钥	/etc/ssh_host_key.pub	/etc/ssh_host_dsa_key.pub	/etc/ssh2/hostkey.pub
客户端主机密钥	/etc/ssh_known_hosts	/etc/ssh_known_hosts ~/.ssh/ssh_known_hosts ~/.ssh/ssh_known_hosts2	/etc/ssh2/hostkeys ~/.ssh/ssh_known_hosts ~/.ssh2/hostkeys/*
远程主机密钥	~/.ssh/ssh_known_hosts	~/.ssh/ssh_known_hosts ~/.ssh/ssh_known_hosts2	~/.ssh2/knownhosts/*
libwrap 控制文件	/etc/hosts.allow /etc/hosts.deny	/etc/hosts.allow /etc/hosts.deny	/etc/hosts.allow /etc/hosts.deny

表 4-1：软件清单（续）

内容	SSH1	OpenSSH	SSH2
授权可以登录的公钥	<code>~/.ssh/authorized_keys</code>	<code>~/.ssh/authorized_keys</code> <code>~/.ssh/authorized_keys2</code>	<code>~/.ssh2/authorization</code>
授权可以登录的可信主机	<code>/etc/hosts.equiv</code> <code>/etc/shosts.equiv</code> <code>~/.shosts</code> <code>~/.rhosts</code>	<code>/etc/hosts.equiv</code> <code>/etc/shosts.equiv</code> <code>~/.shosts</code> <code>~/.rhosts</code>	<code>/etc/hosts.equiv</code> <code>/etc/shosts.equiv</code> <code>~/.shosts</code> <code>~/.rhosts</code>
公钥认证使用的缺省公钥对	<code>~/.ssh/identity</code> <code>{.pub}</code>	<code>SSH-1/RSA:</code> <code>~/.ssh/identity</code> <code>{.pub}</code> <code>SSH-2/DSA:</code> <code>~/.ssh/id_dsa</code> <code>{.pub}a</code>	(无缺省值)
随机数种子	<code>~/.ssh/random_seed</code> <code>/etc/ssh_random_seed</code>	<code>~/.ssh/prng_seedb</code>	<code>~/.ssh2/random_seed</code> <code>/etc/ssh2/random_seed</code>
生成随机数所使用的命令	-	<code>/etc/ssh_prng_cmds</code>	-
Kerberos	<code>/etc/krb5.conf</code> <code>~/.k5login</code>	<code>/etc krb.conf</code> <code>~/.klogin</code>	-
终端上的客户端	<code>ssh1</code> 链接到 <code>ssh1</code> 上的 <code>slogin</code>	<code>ssh</code> 链接到 <code>ssh</code> 上的 <code>slogin</code>	<code>ssh2</code>
安全文件拷贝客户端	<code>scp1</code>	<code>scp</code>	<code>scp2</code>
签名程序	-	-	<code>ssh-signer2</code>
<code>sftp2/scp2</code> 服务器	-	-	<code>sftp-server2</code>
认证代理	<code>ssh-agent1</code>	<code>ssh-agent</code>	<code>ssh-agent2</code>
密钥生成程序	<code>ssh-keygen1</code>	<code>ssh-keygen</code>	<code>ssh-keygen2</code>
增加 / 删除密钥	<code>ssh-add1</code>	<code>ssh-add</code>	<code>ssh-add2</code>

表 4-1：软件清单（续）

内容	SSH1	OpenSSH	SSH2
搜索 SSH 服务器	-	-	<i>ssh-probe2</i>
通过终端或 X 获取口令	<i>ssh-askpass1</i>	-	<i>ssh-askpass2</i>
服务器程序	<i>sshd1</i>	<i>sshd</i>	<i>sshd2</i>

- a. 这里不能像 OpenSSH/1 一样改成使用 *-i* 选项；而应该使用 *-o Identity2=key_file*。
- b. 只有在使用 OpenSSH 内部的 entropy-gathering 机制时才有（也就是系统中没有 /dev/random 或 equivalent）。即使有 /dev/random 文件，SSH1 和 SSH2 也都使用随机数文件。

4.5 使用 SSH 代替 r- 命令

SSH 和 r- 命令 (*rsh*、*rcp*、*rlogin*) 可以在一台机器上和平共处。但是，由于 r- 命令是不安全的，因此很多系统管理员都倾向于使用 SSH 对应的程序 (*ssh*、*scp*、*slogin*) 来代替 r- 命令。这种替换工作可以分为两个步骤：

- 安装 SSH 并删除 *rsh*、*rcp* 和 *rlogin*；这需要用户具有一定的水平。
- 修改其他调用 r- 命令的程序或脚本。

r- 命令和对应的 SSH 命令十分类似，用户常常会想把 SSH 命令重新命名成 r- 命令（例如，把 *ssh* 重命名成 *rsh*，等等）。不管怎么说，这两个命令常用语法的确是相同的：

```
$ rsh -l jones remote.example.com
$ ssh -l jones remote.example.com

$ scp myfile remote.example.com:
$ scp myfile remote.example.com:
```

为什么不直接重命名呢？这是因为这两组程序实际上是有些不兼容的。例如，并不是所有版本的 *ssh* 都支持 *rsh* 的“主机名链接”特性，[2.7.3] 有些旧版本的 *rcp* 用来定义远程文件名的语法也不一样。

在以下章节中，我们会讨论一些通用的会调用 r- 命令的 Unix 程序，并介绍如何对这些程序进行修改，从而让这些程序使用 SSH。

4.5.1 /usr/hosts 目录

rsh 程序有一个有趣的特性，称为主机名链接。[2.7.3]如果把可执行文件 *rsh* 重命名成其他名字，那么 *rsh* 就会把这个新的名字当作是一个主机名，并缺省地连接到这个主机名上。例如，如果把 *rsh* 重命名成“*petunia*”，在调用时就执行 *rsh petunia*。这种重命名可以是真的重命名，也可以是建立指向 *rsh* 的硬链接或符号链接：

```
$ ls -l petunia
lrwxrwxrwx 1 root      12 Jan 31 1996 petunia -> /usr/ucb/rsh
$ petunia
Welcome to petunia!
Last login was Wed Oct 6 21:38:14 from rhododendron
You have mail.
```

有些 Unix 机器上有一个目录包含了指向 *rsh* 的符号链接，用来表示局域网内部（或外部）的主机，这个目录通常是 */usr/hosts*：

```
$ ls -l /usr/hosts
lrwxrwxrwx 1 root      12 Jan 31 1996 lily -> /usr/ucb/rsh
lrwxrwxrwx 1 root      12 Jan 31 1996 petunia -> /usr/ucb/rsh
lrwxrwxrwx 1 root      12 Jan 31 1996 rhododendron -> /usr/ucb/rsh
...
```

如果在这种机器上把 */usr/ucb/rsh* 删除了，显然这些链接都孤立了。此时可以删除这些链接并将其替换成指向 *ssh* 的链接，这可以使用这样的 shell 脚本来完成：

```
#!/bin/sh
SSH=/usr/local/bin/ssh
cd /usr/hosts
for file in *
do
    rm -f $file
    ln -s $SSH $file
    echo "Linked $file to $SSH"
done
```

4.5.2 CVS

CVS (Concurrent Version System, 并发版本系统) 是一个版本控制系统。它包含了对一组文件各个阶段进行修改的历史记录，有助于协调多个用户对相同的文件进行的修改工作。它可以使用 *rsh* 连接到远程主机的仓库中。例如，可以导入一个文件新的版本：

```
$ cvs commit myfile
```

如果仓库是在一台远程主机上，那么CVS就可以调用*rsh*来访问远程主机上的仓库。要使用更安全的方法，CVS可以调用*ssh*来代替*rsh*。当然，远程主机上必须正在运行SSH服务器，而且如果使用公钥认证，那么远程账号中必须在适当的地方包含了密钥（注7）。

要让CVS使用*ssh*，可以简单地把环境变量CVS_RSH设置成你的*ssh*客户端所在的目录：

```
# Bourne shell族
# 放在 ~/.profile 中保持
CVS_RSH=/usr/local/bin/ssh
export CVS_RSH

# C shell族
# 放在 ~/.login 中保持
setenv CVS_RSH /usr/local/bin/ssh
```

这种方法有一个问题：每次导入文件时，登录者的名字都要是远程账号名，而不能是自己的名字。这个问题通常可以通过在远程*authorized_keys*文件中使用“environment=”选项设置远程的LOGNAME变量来解决。[8.2.6.1]

4.5.3 GNU Emacs

Emacs的变量*remote-shell-program*包含了任何想要调用远程shell程序的路径。可以简单地将其重新定义成*ssh*可执行文件所在的完整路径。还有，*rlogin*包*rlogin.el*也定义了一个变量*relogin-program*，可以将其重新定义成使用*slogin*。

4.5.4 Pine

Pine邮件阅读程序使用*rsh*来调用远程机器上的邮件服务器软件。例如，它可以调用远程邮件服务器上的IMAP守护进程*imapd*。我们可以通过修改Pine配置文件的变量*rsh-path*的值来使用另外一个程序代替*rsh*。这个变量中存放了打开远程shell

注7： CVS还有一种远程访问方法会涉及到自己的服务器，称为*pserver*。这种机制可以使用SSH端口转发特性来加强安全性；请参看第九章。

连接所使用的程序名，通常是 `/usr/ucb/rsh`。用户可以在自己的 Pine 配置文件 `~/.pinerc` 中设置新值，或者在系统范围的 Pine 配置文件（通常是 `/usr/local/lib/pine.conf`）中设置。例如：

```
# 在 Pine 配置文件中设置  
rsh-path=/usr/local/bin/ssh
```

还有一个变量 `rsh-command` 用来构建要在远程邮件服务器上执行的实际命令字符串。该值的模式和 C 函数 `printf()` 的风格相同。用户很可能就不需要修改该值，因为 `rsh` 和 `ssh` 都符合缺省的模式，也就是：

```
"%s %s -l %s exec /etc/r%sd"
```

前 3 个 “`%s`” 模式分别表示 `rsh-path`、远程主机名和远程用户名。（第四个 “`%`” 表示远程邮件守护进程名，这和我们无关。）因此在缺省情况下，如果用户名是 `alice`，远程邮件服务器是 `mail.example.com`，那么 `rsh-command` 就应该是：

```
/usr/ucb/rsh mail.example.com -l alice ...
```

如果我们修改了 `rsh-path`，该命名就变成了：

```
/usr/local/bin/ssh mail.example.com -l alice ...
```

正如我们前面所说的一样，用户可能不需要对 `rsh-command` 进行任何处理，我们在这里介绍这些只是以防万一，仅供参考使用。我们会在后面给出一个集成了 Pine 和 SSH1 的详细实例。[11.3]

4.5.5 rsync、rdist

`rsync` 和 `rdist` 是一台机器上的不同目录或多个主机之间对文件进行同步的软件工具。这两个程序都会调用 `rsh` 连接到远程机器上，而且二者都可以方便地使用 SSH：用户只需要为 `rsync` 设置 `RSYNC_RSH`、在 `rdist` 中使用 `-P` 选项即可。在 `rsync` 中使用 SSH 是安全地维护远程主机上的完整目录树的一种相当简单而且有效的方法。

4.6 小结

SSH1、SSH2、F-Secure SSH 服务器和 OpenSSH 都可以使用配置脚本通过编译时

配置进行各种各样的定制。我们已经介绍了 SSH 特有的标志，但是要记住，其他操作系统特有的标志也可以在你安装的环境中起作用，因此用户一定要阅读软件本身所提供的安装说明。

SSH 软件在安装之后就可以替换 Unix 系统中不安全的 r- 命令，这不但可以直接运行 SSH 实现，还可以在调用 *rsh* 的其他程序中实现，例如，Emacs 和 Pine。

第五章

服务器范围 的配置

本章内容：

- 服务器名
- 运行服务器
- 服务器配置：概述
- 开始：原始设置
- 允许用户登录：认证
和访问控制
- 用户登录和账号
- 子系统
- 历史记录、日志记录
和调试
- SSH-1和SSH-2服务
器的兼容性
- 小结

在安装好 SSH 服务器 (*sshd*) 之后，现在就该决定服务器的各种操作了。应该允许使用哪种认证技术呢？服务器密钥应该有多少位呢？空闲连接在超时之后应该断开连接还是让其继续连接呢？这些问题以及其他一些问题都需要仔细考虑。*sshd* 的缺省值都很合理，但是不要盲目地全部接受这些缺省值。你的服务器应该遵守仔细制订的安全策略。幸运的是，*sshd* 的配置非常灵活，用户可以用其实现各种技巧。

sshd 可以在三个层次上进行配置，本章介绍的是第二个层次：服务器范围的配置，此时由系统管理员来控制服务器的全局运行方式。这包括大量、丰富的特性，例如 TCP/IP 设置、加密、认证、访问控制和错误日志。有些特性是通过修改服务器范围的配置文件进行控制的，有些则是在调用的时候向服务器传递命令行参数进行控制的。

另外两个配置层次是编译时配置（第四章）和每账号配置（第八章），前者在服务器编译时就指定了包含哪些特定的功能，不包含哪些功能，后者则是由终端用户来修改自己账号所使用的服务器的行为。我们在本章后文中会更详细地讨论这三个层次之间的区别。

本章只介绍SSH1/SSH2服务器及其派生产品OpenSSH和F-Secure SSH服务器。但是我们所提到的所有实现都是Unix上的SSH1和SSH2。我们一直试图说明各种风格的*sshd*具有哪些特性、没有哪些特性，但是这种情况随着版本的不断更新肯定会发生改变，因此要阅读每种产品的文档才能了解最新信息。

5.1 服务器名

SSH1的SSH服务器名为*sshd1*，SSH2的SSH服务器为*sshd2*，OpenSSH的SSH服务器名为*sshd*。但是，用户可以使用*sshd*来调用*sshd1*和*sshd2*，因为它们的Makefile创建了一个名为*sshd*的符号链接。^{[4.1.3] [4.1.4]}如果系统中安装了*sshd2*，那么这个链接就指向*sshd2*，否则就指向*sshd1*（SSH1的Makefile不会覆盖SSH2所安装的链接）。

本章中的有些特性仅能适用于*sshd1*、*sshd2*或OpenSSH的*sshd*，或者只适用于其中某两个。我们用下面的方法来说明这种情况：

- 如果一个命令行选项只适用于一个包，例如SSH1，我们就在例子中使用*sshd1*和一个注释来表示。例如，在SSH1的-d选项（调试模式）中就可能是这样：

```
* 仅对SSH1  
$ sshd1 -d
```

- 如果一个命令行选项只适用于SSH2，我们就在例子的注释中使用*sshd2*来表示。其-d选项需要一个参数：

```
* 仅对SSH2  
$ sshd2 -d 2
```

- 同理我们可以使用这样的注释来表示OpenSSH和F-Secure特有的特性：

```
* 仅对OpenSSH  
* 仅对F-Secure SSH
```

- 如果一个命令行选项可以用于几个包，那么我们就使用*sshd*来表示SSH服务器。例如，-b选项（设置服务器密钥中的位数）是SSH1和OpenSSH都有的一个选项，因此我们就这样表示：

```
# SSH1, OpenSSH  
$ sshd -b 1024
```

- 同理，当我们讨论配置关键字时，有些关键字也只适用于SSH1、SSH2、

OpenSSH 或其中特定的组合。我们在例子之前都给出一个注释以示区别。例如，`MaxConnections` 关键字可以限制可用的 TCP/IP 链接的个数，它只能在 SSH2 中使用，因此我们所给出的例子就是：

```
# 仅对 SSH2  
MaxConnections 32
```

5.2 运行服务器

通常，SSH 服务器都是在服务器主机启动时调用的，之后就留在后台作为一个守护进程运行。对于大部分用途这样都能正常运行。另外，你也可以手工调用服务器。这在你调试服务器、试验服务器的各种选项或作为非超级用户来运行服务器时十分有用。手工调用需要执行很多工作，而且需要仔细考虑，但是在有些情况下我们别无选择，只能这样处理。

最通常的情况是一台计算机上只运行了一个 SSH 服务器。它通过产生很多子进程来处理多个连接，每个子进程用于一个连接。如果你喜欢，也可以运行多个服务器（注 1）。例如，你可以同时运行 `sshd1` 和 `sshd2`，或者同时运行一个服务器的多个实例，让每个服务器都监听一个不同的 TCP 端口。

5.2.1 以超级用户身份运行服务器

我们可以简单地输入服务器名来调用 SSH 服务器：

```
# SSH1, SSH2, OpenSSH  
$ sshd
```

然后服务器就在后台自动运行；在该行的后边不再需要任何选项。

要在服务器主机启动时调用 SSH 服务器，就要在 `/etc/rc.local` 或你的系统的启动文件中适当地添加一些内容。例如：

```
# 对 sshd 指定路径  
SSHD=/usr/local/bin/sshd  
# 如果 sshd 存在，运行它并向系统控制台返回成功  
if [ -x "$SSHD" ]
```

注 1：如果可以使用 `inetd` 调用 `sshd`，就会为每个连接都创建一个 `sshd`。[\[5.4.3.2\]](#)

```
then
    $SSHDA && echo 'Starting sshd'
fi
```

SSH2 使用了一个 SysV 风格的初始化控制脚本，名为 *sshd2.startup*。

5.2.2 以普通用户身份运行 SSH 服务器

任何用户都可以运行 *sshd*，不过普通用户需要先完成几个步骤：

1. 得到系统管理员的许可。
2. 生成一个主机密钥。
3. 选择端口号。
4. 创建服务器配置文件（可选）。

在开始试验之前，用户应该先问一下系统管理员：能不能运行 SSH 服务器。虽然从技术的观点来说这个步骤并不是必须的，但却是一个明智的决定。系统管理员可能不喜欢你背着他另外开个登录的后门。而且，如果系统管理员已经禁用了 SSH 或一些 SSH 特性，那么这可能是由于安全原因，此时就不应该使用 SSH 了！

接下来，用户必须创建自己的主机密钥。超级用户可以读取现有的所有主机密钥。主机密钥是使用 *ssh-keygen* 程序生成的。^[6.2]现在，我们要创建一个 1024 位的主机密钥，并将其保存在 *~/myserver/hostkey* 文件中，对于 SSH1 或 OpenSSH，可以这样使用：

```
# SSH1, OpenSSH
$ ssh-keygen -N '' -b 1024 -f ~/myserver/hostkey
```

该命令在 *~/myserver* 目录中创建文件 *hostkey* 和 *hostkey.pub*（因此要确保该目录存在）。SSH2 使用的命令也类似：

```
# 仅对 SSH2
$ ssh-keygen2 -P -b 1024 ~/myserver/hostkey
```

-P 和 *-N* 选项可以把创建的密钥以正文保存，因为 *sshd* 希望不要提示用户输入口令就可以读取密钥。

再次，用户必须选择一个 SSH 服务器要监听的端口号。端口号是使用 *sshd* 的 *-p* 命令行选项或配置文件中的 *Port* 关键字设置的，稍后我们就会介绍。用户的服务器不能监听缺省的 22 端口，因为只有超级用户可以运行程序来监听该端口。端口号必须大于或等于 1024，因为较小的端口号都是保留给操作系统的特权程序使用的。*[3.4.2.3]* 端口号也不能和服务器上其他程序所使用的端口号冲突；如果端口号引起了冲突，那么在试图启动服务器时就会出现错误：

```
error: bind: Address already in use
```

如果接收到这个错误，就另外试一个可用端口（大于 1024）。不要使用计算机的服务映像中给出的端口号（通常是 */etc/services* 或网络信息服务（NIS）的“服务”映像，可以使用 Unix 的命令 *ypcat -k services* 来查看该文件的内容）。系统管理员已经把这些端口号保留给特定的程序或协议使用了，因此如果使用这些端口号就可能会出现问题。

最后，还必须创建自己的 SSH 服务器配置文件。否则，服务器就使用内建的特性或使用服务器范围的配置文件（如果存在），这样服务器的操作可能就不是用户想要的。

假设用户已经在 *~/myserver/hostkey* 中创建了一个主机密钥，并选择端口号 2345，又在 *~/myserver/config* 中创建了一个配置文件，那么就可以使用这个命令来调用 SSH 服务器：

```
# SSH1, SSH2, OpenSSH
$ sshd -h ~/myserver/hostkey -p 2345 -f ~/myserver/config
```

由普通用户运行的服务器有一些缺点：

- 它是在普通用户而不是 root 的 uid 下运行的，因此只能连接到用户自己的账号上。
- 它是手工调用的，而不是在计算机启动时自动调用的。因此，要运行服务器，用户必须有一次是不使用 SSH 连接到服务器上的。每次计算机重新启动时，SSH 服务器也就死掉了，需要重复以上步骤。还好用户可以创建一个 *cron* 任务让它自动运行。

- 在设置服务器时，如果碰到有些工作不正常的情况，仔细阅读服务器打印出的诊断信息是很有帮助的。不幸的是，服务器日志都是写到服务器的日志文件中的，该文件不归用户所有，可能用户也不能访问。由于`sshd`是通过`syslog`服务处理日志的，因此普通用户不能控制日志消息发往什么地方。要查看这些日志，需要知道系统日志在哪儿，通常是在`/var/adm/messages`、`/var/log/messages`或者其他和`syslogd`的设置有关的一些位置，用户需要有适当的权限才可以读取这些文件。要避免这件讨厌的事情，可以考虑以调试模式运行服务器，这样消息就可以显示在终端上了（同时也写入系统日志）。[5.8]这样，用户可以方便地研究错误消息并诊断故障，直到服务器正常为止。

然而，对于大部分用户来说，SSH的优点使得这些不便之处显得微不足道。假设系统管理员允许，即使不是超级用户，也可以使用`sshd`来增加登录的安全性。

5.3 服务器配置：概述

正如在本章开始所介绍的一样，服务器`sshd`的行为可以在三个层次上进行控制：

- 编译时配置（第四章）是在编译`sshd`时完成的。例如，服务器可以编译成支持`rhosts`认证，也可以编译成不支持`rhosts`认证。
- 服务器范围的配置（也就是本章的重点）是由系统管理员执行的，适用于服务器的运行实例。例如，系统管理员可以禁止一个给定域中的所有主机访问SSH，也可以让SSH服务器监听一个特定的端口。

服务器范围的配置并不是和编译时配置相独立的。例如，只有在服务器编译时包含了可信主机认证的支持，服务器才能支持可信主机认证选项。否则，这些选项就无效。这种依赖关系在本书中贯穿始终。图5-1重点突出了服务器配置任务。

- 每账号配置（第八章）比较特殊，它是由终端用户执行的，终端用户是SSH连接发往的目的账号的主人。例如，用户可以允许或禁止来自特定主机的连接访问自己的账号，这个设置可以覆盖服务器范围的配置。

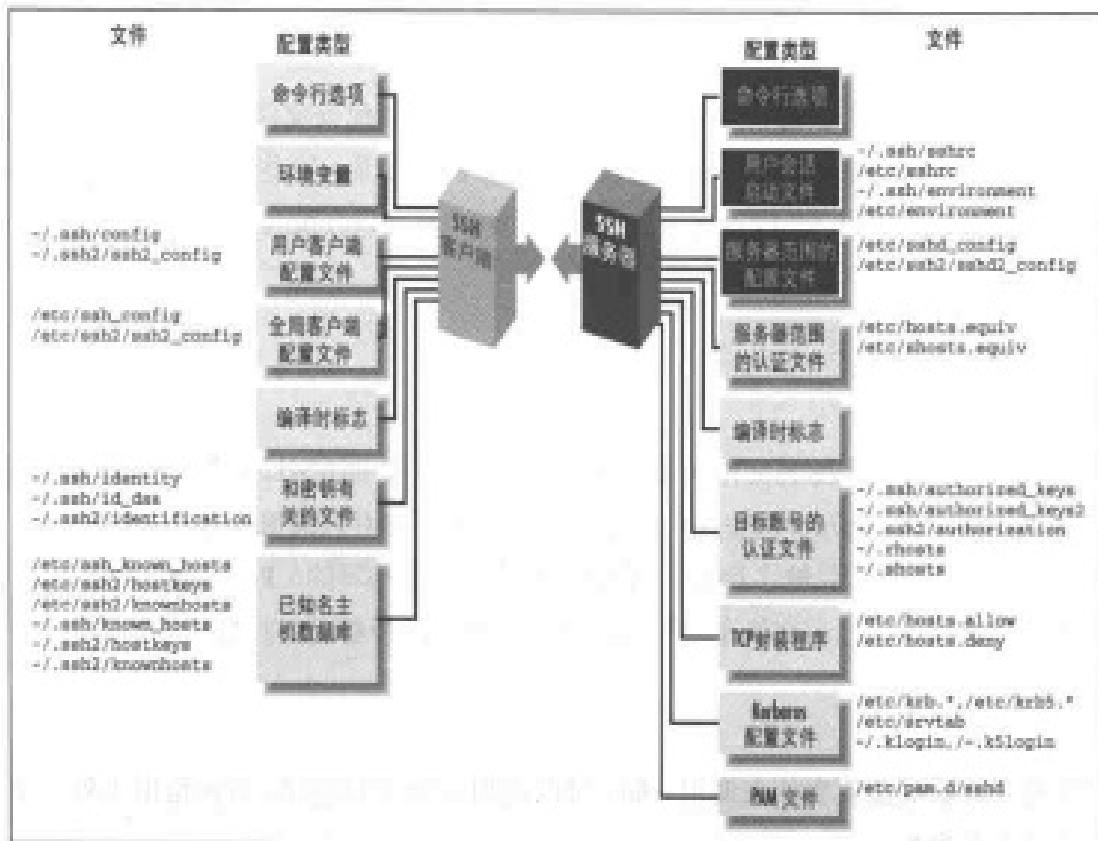


图 5-1：服务器范围的配置（高亮显示部分）

假设用户 deborah 在 `client.unc.edu` 上调用一个 SSH 客户端。客户端的行为是由编译时所选定的编译时选项、`client.unc.edu` 上整个机器范围内的客户端配置文件、deborah 自己的客户端配置文件以及 deborah 在调用 SSH 客户端时所使用的命令行选项共同决定的。`server.unc.edu` 上运行的一个 SSH 服务器负责接收 deborah 的连接，并将其转换到 charlie 的账号上。服务器的行为是由编译 `sshd` 时使用的编译时选项、`server.unc.edu` 上机器范围的服务器配置文件、运行 SSH 服务器时所使用的命令行选项以及 charlie 的个人服务器配置文件（例如，`authorized_keys` 文件）加上设置成功登录会话所使用的环境变量的几个文件共同确定的。

使用这三个层次的服务器配置，每个层次上都有很多地方可以修改服务器的行为，这样事情就变得相当复杂了。具体点说，就是不同的选项可以同时生效，也可以互相抵消。例如，用户 Charlie 可以配置 `server.unc.edu` 上自己的账号，使其接收来自 `client.unc.edu` 的连接，而 `server.unc.edu` 的系统管理员可以把 SSH 服务器配置成拒绝接收这些连接。（在这种情况下，输家是 Charlie。）系统管理员不但要知道如何配置服务器本身，还要知道自己的选择是如何和编译时选项及每账号配置交互影响的。

5.3.1 服务器配置文件

服务器范围的配置可以通过两种方法实现：通过服务器配置文件，或者通过命令行选项。在服务器配置文件中，我们可以设置很多配置变量，这些配置变量称为关键字（keyword）。例如，要设置服务器监听哪个端口，可以在配置文件中包含这样一行：

```
# SSH1, SSH2, OpenSSH  
Port 1022
```

SSH1 和 OpenSSH 的配置文件通常是 */etc/sshd_config*，而 SSH2 的配置文件通常是 */etc/ssh2/sshd2_config*。这些文件都包含一些关键字和对应的值，都和 Port 的例子一样，每行是一对（关键字和值）。关键字是不区分大小写的：Port、port 和 PoRt 都被认为是相同的。在文件中同时还包含有注释：注释都以 # 开始：

```
# 一条注释
```

要使用非缺省的配置文件来调用 *sshd*，可以使用 -f 命令行选项，后面给出另外一个文件名作为参数：

```
# SSH1, SSH2, OpenSSH  
$ sshd -f /usr/local/ssh/my_config
```

对于 SSH2 来说，配置文件的格式除了关键字之外还有以下的扩展：

段

自从随 SSH2 一起发布的样例配置文件中出现标号 *：之后，在配置文件的开头通常都会出现这个标号。它实际上根本没有什么用处，而且容易引起混淆；请参看下面的说明。

子系统

以 “subsystem-” 关键字开始，例如：

```
# 仅对 SSH2  
subsystem-sftp      sftp-server
```

就指明了 SSH2 客户端可以使用名字调用的一个子系统和一个预先定义的命令。子系统是一个抽象层，也是一种十分方便的特性。[5.7]

什么是 *: ?

SSH2 提供的 *sshd2_config* 文件在开始包含了下面这样一行内容，位置正好位于关键字设置之前：

```
# The "*" defines for all hosts  
*:;
```

这些内容根本就没有必要，甚至是一种误导。在客户端的配置文件中，冒号标志着配置文件中的一段内容的开始，[7.1.3.3] 此后的一段内容（直到下一段，或者文件末尾）只有在用户连接到主机名是这个标号的计算机上时才管用。

段标号语法在服务器配置文件中也可以被识别，但是没有用处。按照源代码的编写方法，能和服务器范围匹配的标号只有 *，这也是缺省的标号，因此它根本就没有必要。

段标号是一种误导，因为它建议用户可以这样在服务器配置文件中对段进行标记：

```
client.host.net:  
    AllowUsers smith
```

通过对客户端配置文件的分析，用户可能希望对来自 *client.host.net* 的连接进行限制，只允许它登录到“smith”账号中。这是不可能的。实际上，以 * 作为标号的语句都会被 *sshd* 忽略。这一点一定要小心！

5.3.2 命令行选项

另外，在调用服务器时，用户还可以提供命令行选项。例如，可以使用 *-p* 命令行选项指定端口号：

```
* SSH1, SSH2, OpenSSH  
$ sshd -p 1022
```

命令行选项会覆盖配置文件中的设置。因此，如果配置文件中设置了端口是 1022，而调用服务器时指定了 *-p 2468* 选项，那么所使用的端口就是 2468。

大部分命令行选项的特性都和配置文件中的特性相同，这样使用主要是为了方便起

见；也有一些选项提供了特殊的功能。例如，`-f`选项就通知`sshd`使用另外一个配置文件，这种特性放在配置文件中是没什么用处的。

另一方面，关键字也没有必要和命令行选项完全相同。大部分SSH1和OpenSSH关键字都是不同的。但是所有的SSH2关键字都可以使用`-o`服务器的命令行选项设置。例如，要使用这种方法来设置TCP端口号：

```
# 仅对 SSH2
$ sshd2 -o "Port 1022"
```

5.3.3 修改配置

`sshd`在启动时读取配置文件。因此，如果用户在服务器运行时修改了配置文件，那么修改不会立即影响到服务器。用户必须强制服务器重新读取配置文件以便使它接受所做的修改。这是通过向服务器进程发送一个SIGHUP信号来实现的（注2）。服务器进程的pid保存在一个文件中，对于SSH1这个文件通常是`/etc/sshd.pid`，对于SSH2通常是`/var/run/sshd2_22.pid`，对于OpenSSH通常是`/var/run/sshd.pid`。
[5.4.1.3]

假设PID文件是`/etc/sshd.pid`，也就是`sshd1`所使用的缺省文件。要向服务器进程发送SIGHUP信号，可以运行Unix的`kill`命令：

```
$ cat /etc/sshd.pid
119384
$ kill -HUP 119384
```

或者可以更方便地这样使用：

```
$ kill -HUP `cat /etc/sshd.pid`
```

SIGHUP信号重启`sshd`（新的`sshd`的pid不同了），但是不会中断现有的SSH连接，因此在有客户端连接到服务器上时，发送这个信号也是安全的。新的`sshd`进程读取新的配置文件，并按新的配置文件进行设置。

SIGHUP技术只能用于更新配置文件的设置，而不能用于更新命令行选项的设置。要修改命令行选项，必须删掉服务器进程并使用新的选项来重启`sshd`。例如：

注2：SSH2服务器支持使用SIGHUP重启2.0.12版本的`sshd`。

```
# SSH1, SSH2, OpenSSH
$ kill 119384
$ sshd new_options
```

5.3.4 一个巧妙的重新配置的例子

由于命令行选项会覆盖配置文件中对应的设置，因此有时会出现一些很有趣的事情。假设配置文件中定义了服务器密钥的位数应该是 1024：

```
# SSH1, OpenSSH
ServerKeyBits 1024
```

而服务器是使用 `-b` 命令行选项调用的，将该值修改成 512：

```
# SSH1, OpenSSH
$ sshd -b 512
```

那么服务器就使用 512 位的密钥。假设现在使用 SIGHUP 信号重新启动 `sshd`：

```
# 仅对 SSH1
$ kill -HUP `cat /etc/sshd.pid`  
  
# 仅对 OpenSSH
$ kill -HUP `cat /var/run/sshd.pid`
```

并强制 `sshd` 重新读取配置文件。这时你认为服务器密钥的长度应该是多少呢？服务器重新读取配置文件之后将密钥长度设置成 1024；还是命令行选项依然有效，密钥长度仍然是 512 呢？实际上，命令行的优先权要高些，密钥仍然是 512 位的。`sshd` 把命令行参数（`argv`）保存下来，重新启动时仍然使用这些参数。

5.4 开始：原始设置

现在我们开始详细讨论使用关键字和命令行选项的 SSH 服务器的配置。要记住 SSH2 和 OpenSSH 的产品仍然在不断更新，其特性也会不断变化。要了解最新的消息就要阅读它们的文档。SSH1 已经不再主动开发了，因此其特性就不会有太多变化了。

我们首先讨论原始设置，例如：重要的文件应该保存在什么地方？这些文件的权限应该如何设置？应该使用什么样的 TCP/IP 设置？服务器密钥的特性应该如何？应该支持哪些加密算法？

5.4.1 文件位置

sshd 要求一些文件必须存在，其中包括服务器主机密钥、随机数种子和其他一些数据。服务器会按照缺省位置来查找这些文件，或者用户也可以使用后文中介绍的关键字和命令行选项来覆盖这些设置。

虽然用户可以任意把这些文件放在任何地方，但是我们强烈建议把这些文件保存在服务器上的本地硬盘中，而不要放在远程装载的磁盘上（例如，通过 NFS）。这是出于安全性的考虑，因为 NFS 在网络上传输敏感数据时都未加密过。尤其是对于未加密的私有主机密钥来说，这样做的危害就尤其严重。

为了给出一个实际运行的例子，我们使用 */usr/local/ssh* 目录来代表我们推荐（而非缺省）的 SSH 服务器文件的存储位置。

5.4.1.1 主机密钥文件

sshd 的主机密钥向 SSH 客户端唯一地标识 SSH 服务器。主机密钥保存在一对文件中，一个文件包含私钥，另外一个文件包含公钥。对于 SSH1 和 OpenSSH 来说，私钥保存在 */etc/ssh_host_key* 中，只能由诸如 SSH 服务器和客户端之类的特权程序读取。这些文件的位置可以使用 *HostKey* 关键字修改：

```
# SSH1, OpenSSH
HostKey /usr/local/ssh/key
```

服务器的公钥保存在另外一个文件中，该文件的名字是私钥文件名后面加上 *.pub*。因此 SSH1 和 OpenSSH 缺省的公钥文件名是 */etc/ssh_host_key.pub*，前面的例子中 *HostKey* 关键字就暗示了我们使用的公钥文件是 */usr/local/ssh/key.pub*。

OpenSSH 服务器 *hai* 有一个 SSH-2 主机密钥，缺省情况下是在 */etc/ssh_host_dsa_key* 中，其位置可以使用 *HostDsaKey* 关键字进行修改：

```
# 仅对 OpenSSH
HostDsaKey /usr/local/openssh/key2
```

对于 SSH2 来说，如果服务器是由超级用户运行的，那么缺省的私钥文件是 */etc/ssh2/hostkey*；如果是由其他用户运行的，那么缺省的私钥文件是 *~/.ssh2/hostkey*。要指定其他的私钥文件，可以使用 *HostKeyFile* 关键字：

```
# 仅对 SSH2  
HostKeyFile /usr/local/ssh/key
```

超级用户运行的服务器的公钥文件通常是`/etc/ssh2/hostkey.pub`, 而其他用户运行的服务器的公钥文件通常是`~/.ssh2/hostkey.pub`, 用户也可以使用`PublicHostKeyFile`关键字单独进行修改:

```
# 仅对 SSH2  
PublicHostKeyFile /usr/local/ssh/pubkey
```

如果喜欢使用命令行选项, `sshd` 也可以支持`-h` 命令行选项来指定私钥文件:

```
# SSH1, SSH2, OpenSSH  
$ sshd -h /usr/local/ssh/key
```

同样, 公钥文件名是在私钥文件名后面加上`.pub`, 在我们这个例子中, 公钥文件就是`/usr/local/ssh/key.pub`。

5.4.1.2 随机数种子文件

SSH 服务器要为加密操作生成伪随机数。^[3.7] 它为此目的维护了一个随机数据池, 如果操作系统提供了随机数, 那么随机数据池中的数据就来自于操作系统(例如, 很多 Unix 风格的操作系统上的`/dev/random`), 否则就来自于机器状态位的变化(例如, 时钟时间、进程使用的资源总量等等)。这个数据池就称为随机数种子(`seed`)。SSH1 将其存储在`/etc/ssh_random_seed`中, 也可以使用`RandomSeed`关键字来修改这个缺省位置。

```
# 仅对 SSH1  
RandomSeed /usr/local/ssh/seed
```

同理, SSH2 的随机数种子保存在`/etc/ssh2/random_seed`文件中, 可以使用`RandomSeedFile`关键字进行修改:

```
# 仅对 SSH2  
RandomSeedFile /usr/local/ssh/seed2
```

如果用户使用的系统中有一个随机位源, 例如`/dev/urandom`, 那么 OpenSSH 就不能创建随机数种子文件。

5.4.1.3 进程 ID 文件

我们前面已经说过 SSH1 服务器的 pid 是保存在 `/etc/ssh.pid` 中的，这个位置可以使用 `PidFile` 关键字覆盖：

```
# SSH1, OpenSSH
PidFile /usr/local/ssh/pid
```

SSH2 没有对应的关键字。其 pid 文件通常是 `/var/run/sshd2_N.pid`，其中 N 是服务器的端口号。由于 SSH2 使用的缺省端口是 22，因此缺省的 pid 文件就是 `/var/run/sshd2_22.pid`。如果在一台机器上的多个端口上同时运行了多个 `sshd2` 进程，那么它们的 pid 文件可以通过这种命名约定加以区分。

5.4.1.4 服务器配置文件

SSH1 和 OpenSSH 的服务器配置文件通常都是 `/etc/sshd_config`，而 SSH2 服务器的配置文件通常是 `/etc/ssh2/sshd2_config`。我们也可以使用 `-f` 命令行选项重新指定其他配置文件：

```
# SSH1, SSH2, OpenSSH
$ sshd -f /usr/local/ssh/config
```

这在测试一个新服务器的配置时特别有用：用户可以创建一个新文件并通知 `sshd` 来读取该文件。如果在同一台机器上运行了多个 `sshd`，并希望它们使用不同的配置文件，也必须使用这个命令行选项。

5.4.1.5 用户的 SSH 目录

`sshd1` 希望用户所有和 SSH 有关的文件都在 `~/.ssh` 目录中。这个位置不能使用服务器范围的配置进行修改。（如果要这样做，必须得修改源代码。）

`sshd2` 希望用户的所有和 SSH 有关的文件缺省都在 `~/.ssh` 目录中，这个位置可以使用 `UserConfigDirectory` 关键字进行修改。该目录可以以纯文字的形式给出，例如：

```
* 仅对 SSH2
UserConfigDirectory /usr/local/ssh/my_dir
```

也可以使用类似于 `printf` 的模式，例如：

```
# 仅对 SSH2
UserConfigDirectory %D/.my-ssh
```

%D 模式代表用户主目录。因此上面的例子实际上代表 `~/.my-ssh`。下表给出了所有可用的模式：

模式	意义
%D	用户主目录
%U	用户登录名
%IU	用户 uid (Unix 的用户 ID)
%IG	用户 gid (Unix 的组 ID)

对于系统管理员来说，UserConfigDirectory关键字提供了一种快速覆盖所有用户的SSH2设置的方法。具体地说，可以让 `sshd2` 忽略每个用户的 `~/.ssh2` 目录，而使用用户所提供的目录。例如，下面这一行：

```
# 仅对 SSH2
UserConfigDirectory /usr/sneaky/ssh/%IU/
```

就告诉 `sshd2` 对所有的用户都使用 `/usr/sneaky/ssh/<username>` 中的设置，而不再使用每个用户 `~/.ssh` 中的设置。如果用户的机器被攻击了，这个强大的工具就可能会被滥用。如果入侵者在 `sshd2_config` 中插入了这样一行：

```
# 仅对 SSH2
UserConfigDirectory /tmp/hack
```

并把自己的公钥文件放到 `/tmp/hack` 中，那么他就可以访问所有用户的账号了。

5.4.1.6 每账号认证文件

SSH1 和 OpenSSH 服务器都希望能在 `~/.ssh/authorized_keys` (OpenSSH/2 的缺省公钥认证文件位于 `~/.ssh/authorized_keys2` 中) 目录中找到用户的公钥认证文件。这些文件的位置不能使用服务器范围的配置来修改。

SSH2 服务器使用的密钥文件的格式不同。[6.1.2] 认证文件通常是 `~/.ssh2/authorization`，其中包含了各个公钥文件的文件名，而不是公钥本身。我们可以使用关键字 `AuthorizationFile` 来通知 `sshd2` 使用其他认证文件。

```
# 仅对 SSH2
AuthorizationFile my_public_keys
```

其中的文件名可以是绝对路径，也可以是位于用户的SSH2目录下的相对路径。上面这个例子指定认证文件是`~/.ssh2/my_public_keys`。

5.4.2 文件权限

作为一种安全产品，SSH1、SSH2和OpenSSH都需要保护服务器主机上特定的文件和目录不能被其他用户访问。假设所有用户都对用户的*authorized_keys*和*.rhosts*文件具有写入权限，那么该主机上的所有用户都可以修改这些文件并方便地访问该用户的账号。*sshd*有几个配置关键字可以降低这种风险。

5.4.2.1 用户文件可以接受的权限

用户通常都不会总是很小心地保护自己目录中的重要文件和目录，例如*.rhosts*文件和自己的SSH目录。这种疏漏可以导致一些安全漏洞并危及账号的安全。要避免这种情况的出现，可以对*sshd*进行配置，让它拒绝接受权限不对的用户的连接。

*StrictModes*关键字加上一个值`yes`（缺省值）可以让*sshd*对重要文件和目录的权限进行检查。这些文件和目录的属主必须是账号属主或root，而且必须禁用同组和其他用户的可写权限。对于SSH1来说，*StrictModes*要检查以下内容：

- 用户主目录。
- 用户的`~/.rhosts`和`~/.shosts`文件。
- 用户的SSH配置目录`~/.ssh`。
- 用户的SSH`~/.ssh/authorized_keys`文件。

对于OpenSSH来说，*strictModes*除了要检查和SSH1相同的文件以外，还要检查SSH-2连接使用的用户认证文件`~/.ssh/authorized_keys2`。

对于SSH2来说，*StrictModes*所要检查的内容就少多了，只需要检查可信主机认证所使用的文件即可：[3.4.2.3]（注3）

注3：SSH2 2.2.0的*sshd2_config*的手册页中说没有实现*StrictModes*，但是这并不是绝对的。

- 用户主目录。
- 用户的 `~/.rhosts` 和 `~/.shosts` 文件。

如果检查失败，服务器就拒绝对该用户的 SSH 连接。如果 `StrictModes` 后面跟上的值是 `no`，就不会执行这些检测。

```
# SSH1, SSH2, OpenSSH
StrictModes no
```

但是，我们强烈建议启用这种检测。

然而，即使启用了 `StrictModes`，在两种情况下也可能会失效。首先，`sshd` 可能是使用 `--enable-group-writeability` 标志编译的[4.1.5.2]，这样 `StrictModes` 就认为同组的用户对这些文件也具有可写权限。其次，用户可以使用 POSIX ACL（POSIX ACL 可以在 Solaris 和其他一些风格的 Unix 上使用）更精确地设置文件的权限，这样 `StrictModes` 就不能完整地完成测试了。

5.4.2.2 新创建的文件的权限

Unix 进程的 `umask` 确定了该进程创建的所有文件和目录的缺省权限。`sshd` 的 `umask` 可以使用 `Umask` 关键字来指定，这样该进程所创建的所有权限都具有我们想要的权限。该值是一个通用的 Unix `umask` 值，通常都以八进制形式给出：

```
# 仅对 SSH1
# Create files r-w-r--r-- and directories rwx-r-xr-x:
Umask 022
```

记住必须在该值前面加上一个 0，这样 `sshd` 才会把它当作八进制来处理。有关 `umask` 的更多信息，请参看 Unix 上有关 `umask` 和 shell 的手册页。

`sshd` 会创建一个 `pid` 文件 (`/etc/sshd.pid`，或者是 `PidFile` 所设定的文件) 和一个随机数种子文件 (`/etc/ssh_random_seed`，或者是 `RandomSeed` 所设定的文件)。服务器的 `umask` 设置只对 `pid` 文件管用。随机数种子文件创建时就指定了权限是 0600，只有文件属主才具有读写权限。更确切地说，`umask` 只会影响 `sshd` 所派生出来的进程（也就是用户 shell），但是该值通常都会被用户 shell 的设置覆盖。

5.4.3 TCP/IP 设置

由于SSH协议都是在TCP/IP协议之上进行操作，因此`sshd`允许在各种和TCP/IP有关的参数之上进行控制。

5.4.3.1 端口号和网络接口

缺省情况下，`sshd`监听TCP端口22。用户可以使用`Port`关键字来修改`sshd`监听的端口号：

```
# SSH1, SSH2, OpenSSH
Port 9876
```

也可以使用`-p`命令行选项：

```
# SSH1, SSH2, OpenSSH
$ sshd -p 9876
```

SSH1和OpenSSH服务器可以接受十进制、八进制和十六进制的整数作为端口号，而SSH2服务器只能接受十进制的端口号。请参看“配置文件中的数字值”。

用户也可以把`sshd`配置成把自己的监听端口绑定到一个特定的网络接口上。缺省情况下，该端口被绑定到该主机上所有活动网络接口上。`ListenAddress`关键字可以限制`sshd`只监听一个网络接口，缺省值是0.0.0.0。

例如，假设一台计算机有两个以太网卡并分属于两个不同的网络。一个网卡的地址是192.168.10.23，另外一个网卡的地址是192.168.11.17。缺省情况下，`sshd`要监听这两个端口；因此，用户可以连接到任意一个地址的22端口上。然而，用户可能并不希望一直都这样使用；有时可能希望只让一个网卡提供SSH服务，而另外一个网卡不提供：

```
# SSH1, SSH2, OpenSSH
ListenAddress 192.168.10.23
```

当然，只有在这两个网络没有连接在一起时（也就是使用路由器），这才表示一种真实的限制，这样192.168.11/24网络就不能访问192.168.10.23的22端口。

OpenSSH允许在配置文件中有多个`ListenAddress`行，从而允许监听多个选定的网络接口：

```
# 仅对OpenSSH  
ListenAddress 192.168.10.23  
ListenAddress 192.168.11.17
```

配置文件中的数字值

SSH1和OpenSSH可以接受使用标准的C语言格式的十进制、八进制和十六进制数值。如果所给出的数值以0x开始，那它就被当作是十六进制。如果数值以0开始，就被当作八进制。其他的数值被当作十进制。

SSH2则不同，它要求所有的数值都以十进制的形式给出。

5.4.3.2 通过inetd调用sshd

*sshd*通常都是作为后台进程运行的，它创建一些子进程来处理连接。另外，它也可以像其他网络后台进程一样通过*inetd*调用。在这种情况下，*inetd*会为每个连接都调用这个后台进程的一个新实例。

如果想使用*inetd*来调用*sshd*，用户必须得在服务器主机的TCP/IP服务映射文件（可能是`/etc/services`或`/etc/inet/services`）中放置一项供SSH使用，例如：

```
ssh      tcp/22
```

并在*inetd*配置文件`/etc/inetd.conf`中为SSH服务加入一行。该行必须使用*-i*命令行选项调用*sshd*，这样就可以让*inetd*正确调用*sshd*：

```
ssh stream  tcp      nowait  root    /usr/local/sbin/sshd    sshd -i
```

这一行确切的意思是简单地启动*sshd*，并希望使用标准输入输出对一个TCP socket之上的连接进行处理。这和不使用*-i*选项的操作是不同的，后者会让*sshd*成为一个主服务器，监听TCP连接并创建子进程来处理所有的连接。

使用*inetd*来调用*sshd*的方法有好处也有坏处。从底层来说，如果会话使用服务器密钥，那么基于*inetd*的SSH连接的启动就会很慢，因为*sshd*每次都要创建一个新密钥。这种情况适用于使用SSH-1协议的连接，也就是SSH1和OpenSSH/1服务器。
[3.5.1.2]当然，这是否会成为一个瓶颈问题，要取决于使用的服务器的速度。从高层来讲，*inetd*方法允许使用一个封装的程序来调用*sshd*，这是很有必要的。而且，

*inetd*提供了一个针对所有类型的网络连接进行集中控制点，例如，用户可以简单地禁用*inetd*而不用再删掉其他进程。

5.4.3.3 空闲连接

假设在一台服务器和一台客户端之间建立起了一个SSH连接，但是很长时间都没有使用这个连接传递过数据。那么服务器应该如何处理这种情况呢？是依然保持该连接，还是断开？

SSH1提供了IdleTimeout关键字，它可以告诉如果连接空闲（也就是用户在一定的周期内都没有传递数据）服务器应该怎样操作。如果IdleTimeout是0（缺省值），那么服务器就什么也不做，依然保持这个空闲连接：

```
# 仅对SSH1
IdleTimeout 0
```

否则，在超过指定的空闲时间间隔之后，服务器就会断开这个连接。在这种情况下，IdleTimeout的值是一个正整数，后面可以跟上一个字符：s代表秒，m代表分，h代表小时，d代表天，w代表周。如果后面没有给出字符，该数字就表示秒。

下面几种方法都可以把IdleTimeout设置成刚好一天：

```
# 仅对SSH1
IdleTimeout 1d
IdleTimeout 24h
IdleTimeout 1440m
IdleTimeout 86400s
IdleTimeout 86400
```

我们也可以使用idle-timeout选项对用户的*authorized_keys*文件中给定的密钥设置空闲超时时间。[8.2.7]特别提醒一下，这个选项会覆盖服务器的IdleTimeout值，但是只会影响到这个密钥。每账号选项覆盖服务器范围的选项是极罕见的情况。

5.4.3.4 KeepAlive

KeepAlive和IdleTimeout既相互关联，又互不相同。IdleTimeout负责检测并中断状态良好但未使用的连接，而KeepAlive则负责处理连接失效的情况。假设一个客户端建立了一条SSH连接，经过一段时间之后，客户端突然崩溃了。如果SSH

服务器没有被要求主动向客户端发送数据，那么它就永远不会觉察到对方的这种“半死 (half-dead)”状态的 TCP 连接，*sshd* 依然会维持这个连接，并使用系统内存、进程槽 (process slot) 之类的资源 (而且能让系统管理员的 *ps* 命令显示该进程)。

KeepAlive 通知 *sshd* 如果一个连接发生了问题应该如何处理，这种问题可能是网络延时太长或客户端崩溃了：

```
# SSH1, SSH2, OpenSSH
KeepAlive yes
```

该值为 yes (缺省值) 就让服务器保持客户端的连接。这样使得 TCP 要周期性地发送 *keepalive* 消息，并希望获得响应。如果经过一段时间之后，系统还没有收到响应消息，就向 *sshd* 返回一个错误，然后 *sshd* 就中断连接。该值为 no 就意味着不使用 *keepalive* 消息。

TCP 的 *keepalive* 特性，以及 SSH 的 *KeepAlive* 特性，都是用来防止半死的连接再耗费时间。*keepalive* 消息的间隔和超时周期通常都是这样设置的：这两个值都设置得很大，通常都是几个小时。这样就可以最小化由 *keepalive* 消息所引起的网络负载，并防止连接由于暂时的问题（例如网络暂时超时，或产生路由回环）而不必要地中断掉。这些计时器并不是在 SSH 中设置的；这都是主机的 TCP 栈的特性。这两个值不要轻易修改，因为它们会影响到该主机上使用 *keepalive* 消息的所有 TCP 连接。

KeepAlive 和连接超时。 我们要注意到，*KeepAlive* 并不能处理由于防火墙、代理、NAT 或 IP 伪装超时所引起的连接丢失的问题，这一点非常重要。当 SSH 连接经过使用这些机制的网络时就会出现这个问题，在连接空闲一段时间之后，这些网络机制就可以确定关闭这些连接。由于这些机制是用来保证共享资源（例如有限的外部、可路由的 IP 地址）的可用性的，因此这些超时时间通常都很短，可能大约是几分钟到一个小时。顾名思义，“*KeepAlive*”本身就暗示了可以这样合理地使用，因为用户把自己的连接一直保持在活动状态就是想这样做的。但是实际上，*KeepAlive* 的名字有些名不符实；更好的名字应该是 “*DetectDead*”（不过这个名字听起来倒像是一个二流的牧师在宣扬“不要被僵尸吞噬”时所使用的一个词）。为了让 *KeepAlive* 来解决这个问题，用户只好在 SSH 主机上缩短 TCP *keepalive* 的间隔。但是这样使用就和 *KeepAlive* 的目的正好背道而驰了，而且也并不明智，因为这不仅会影响 SSH 连接，而且还会影响到所有使用 *keepalive* 的连接，即使那些根本不需要的这样

处理的连接也是如此。在服务器端把它设置成一条基本规则的影响尤其糟糕，因为繁忙的服务器可能会使用很多TCP连接，而且它会认为这样做的代价并不惨重，会对很多连接都启用KeepAlive特性。这样会造成很多不想要且颇具破坏力的网络负载，尤其是它广泛应用之后更是如此。

最好记住，之所以要这样做是因为超时的问题会不断骚扰你。用户可能希望即使一个SSH连接没有使用也要保持它一段时间，但是如果它占用了公司可以使用的有限外发Internet TCP连接，那么最好还是这样使用这种机制，这样对大部分程序都有好处。隔一段时间再输入一次ssh实际上并没有那么困难；如果你觉得要敲击的键盘次数太多，可以使用shell的别名特性来简化工作。如果认为超时特性设置得不合适或根本就不必要，就和系统管理员讨论一下，以便进行修改。

对于那些的确需要超时的情况，实现这种keepalive行为的正确方法是，使用在SSH中实现的应用层机制——让其通过这个连接周期性地发送SSH协议消息，从而确保该连接不是空闲的。

这种特性在我们知道的SSH实现中都还没有提供，但是我们期待它们以后会对此进行改进。在NAT等环境中，超时是一个非常常见的问题，我们不提倡滥用TCP keepalives方法作为一种解决方案。同时，还有更好的一种低级方案，就是使用一个程序每隔一段时间通过这个连接发送一些字符。运行Emacs并让它在模式行中显示时间；在后台运行一个程序，如果连接空闲了20分钟就让该程序向终端输出“Boo！”；诸如此类，可以灵活运用。

5.4.3.5 失败的登录

假设一个用户试图通过SSH进行登录，但是认证失败了。此时服务器应该如何处理呢？关键字LoginGraceTime和PasswordGuesses可以控制服务器的响应。

用户被限定必须在一段有限的时间内成功进行认证，缺省是10分钟。超时是由LoginGraceTime关键字来控制的，该值的单位是秒：

```
# SSH1, SSH2, OpenSSH
LoginGraceTime 60
```

也可以使用-g命令行选项完成同样的功能：

```
# SSH1, SSH2, OpenSSH
$ sshd -g 60
```

要禁用这种特性，就要把 `LoginGraceTime` 设置成 0：

```
# SSH1, SSH2, OpenSSH
LoginGraceTime 0
```

或者使用命令行选项：

```
# SSH1, SSH2, OpenSSH
$ sshd -g 0
```

如果连接请求使用密码认证，那么 `sshd2` 就只能允许客户端尝试三次，如果认证不成功，就断开该连接。这种限制可以使用 `PasswordGuesses` 关键字进行修改：

```
# 仅对 SSH2
PasswordGuesses 5
```

使用公钥认证的情况就更复杂了。这种情况下客户端可以发起两种请求：查询一个特定的公钥是否可以用来进行认证，以登录到目的账号中，以及使用对相应的私钥的签名进行实际的认证。最好是不限制客户端可以发起的查询次数，否则就制约了一个用户在代理中可以存放的密钥的数目。但是对于失败尝试的次数进行限制是合理的。现在还没有哪个 SSH 服务器是按照我们认为是正确的这种思路来处理的。`SSH1` 和 `SSH2` 只是简单地允许可以查询公钥，不限制查询次数。另一方面，`OpenSSH` 对每一种认证尝试的次数和任何密钥查询的次数都进行了限制，而且它使用程序内部设置的 5 次进行限制，用户不能配置这个参数（源代码中说该值是 6，但是它编码的方法限定该值是 5）。因此如果用户在自己的代理中有 5 个密钥，那么就不能使用密码认证来登录 `OpenSSH` 服务器了，因为在 `OpenSSH` 判断用户没有使用它要求的密钥之后就拒绝再进行连接。否则，如果有 6 个密钥，而第 6 个密钥刚好就是要使用的密钥，那么你就够倒霉的了；你只好从代理中删除一些密钥（或者不使用代理）才能登录 `OpenSSH` 服务器（顺便说一下，这个数字对于 `OpenSSH/2` 来说更小）。

当然，这又带来一个安全问题需要讨论。最好是不允许客户端进行查询，而且总是禁止客户端一次次进行登录尝试。这样，如果登录失败，那么客户端就不知道到底是因为签名有误还是密钥没有通过认证。这样就增加了攻击者判断要窃取哪个密钥的难度。但是通常这样使用会使得合法的客户端也要耗费很大的力气才能登录，因此协议还是允许进行查询。

5.4.3.6 限制并发连接

缺省情况下`sshd`可以处理任意数目的并发连接。SSH2提供了一个`MaxConnections`关键字对这个数字进行限制，也就是说，如果想保留服务器机器上的一些资源就可以这样使用：

```
# 仅对 SSH2  
MaxConnections 32
```

要将其修改成不限制连接数目，就把`MaxConnections`设置成0：

```
# 仅对 SSH2  
MaxConnections 0
```

当然，连接数目也会受到可用内存或其他操作系统资源的限制。`MaxConnections`对于这些因素就无能为力了（十分抱歉，并不能使用一个关键字来提高CPU的速度！）。

5.4.3.7 逆向IP映射

SSH2服务器也可以根据客户端地IP地址进行逆向DNS查询。也就是说，它先查找地址所关联的主机名（或域名），然后再查找该名的地址，从而确保客户端的地址就是这个地址。如果这种检查失败了，那么服务器就拒绝进行连接。

`sshd2`使用`gethostbyname()`和`gethostbyaddr()`系统服务来执行这种映射，因此所要查询的数据库和主机操作系统的配置有关。它可以使用DNS、网络信息服务（NIS或YP）、服务器上的静态文件，也可以结合使用这几种方法。

要启用这种检测，就要使用`RequireReverseMapping`关键字，并将其设置成`yes`或`no`（缺省值）：

```
# 仅对 SSH2  
RequireReverseMapping yes
```

这种特性是面向安全的一致性检测的一部分。SSH使用加密签名来确定对方的身份，但是对方公钥的列表（已知名数据库）通常都是使用主机名进行索引的，因此SSH必须把地址转换成主机名才能检测对方的身份。逆向映射的目的是试图确保对方没有玩弄修改命名服务器的把戏。但是这只不过是一种折衷的方法，因为现在Internet的DNS逆向地址映射并不能随时保持最新。SSH服务器可能会拒绝合法的连接，因为

用户并不能控制这个过时的逆向地址映射。总之，我们建议关闭这个特性；这一点勿庸置疑。

5.4.3.8 控制 TCP_NODELAY

TCP/IP有一种称为Nagle算法的特性，它被设计用来减小只会发送少量数据（例如，一个字节）的TCP段的个数，通常是将其作为交互终端会话的一部分进行发送。在诸如以太网之类的快速链路上面，通常并不需要Nagle算法。但是在广域网络上，对X客户端或字符终端的显示进行响应就可能产生大量的延时，因为使用该算法来传输多字节的终端控制序列非常不方便。在这种情况下，就应该使用NoDelay关键字关闭Nagle算法：

```
# 仅对 SSH2  
NoDelay yes
```

NoDelay通过在向 Unix 内核请求一个 TCP 连接时设置 TCP_NODELAY 位来禁用 Nagle 算法。该值的合法设置可以是 yes（禁用）和 no（启用，缺省值）。

为了能够正常工作，用户必须在编译时使用`--enable-tcp-nodelay`标记启用这个特性。[\[4.1.5.3\]](#)还要注意，除了可以使用客户端的配置关键字 NoDelay 在服务器范围的配置中启用或禁用 TCP_NODELAY，还可以使用 SSH2 客户端来实现相同的功能。

5.4.3.9 发现其他服务器

SSH2 2.1.0 增加了一种特性，可以用来自动查找并发现 SSH2 服务器。如果我们把 MaxBroadcastsPerSecond 关键字设置成大于 0 的一个整数值，就会让 SSH2 服务器监听发送到 22 端口的 UDP 广播：

```
# 仅对 SSH2  
MaxBroadcastsPerSecond 10
```

SSH2 中新提供的一个程序 `ssh-probe2` 会发送广播请求，并显示所找到的 SSH2 服务器的位置和版本号。服务器只会对这种查询请求每秒钟响应几次；对速度进行限制可以防止服务拒绝攻击，这种攻击手段可以让查询在服务器中泛滥，导致服务器浪费所有的 CPU 时间来响应这些请求，从而降低系统的可用性甚至引发系统崩溃。

`MaxBroadcastsPerSecond` 和 `ssh-probe2` 是用来定位 SSH2 服务器的一种非常特别的方法。随着动态 DNS 和 SRV 记录应用的日益广泛，以后可能就不再需要这种技术了。

5.4.3.10 代理转发

代理转发允许一系列 SSH 连接（从一台机器到另一台机器，再到另外一台机器，……）使用一个代理无缝地进行操作。[\[6.3.5\]](#) 在 SSH2 服务器中我们可以把关键字 `ForwardAgent` 或 `AllowAgentForwarding` 设置成 `yes`（缺省值）或 `no` 来启用或禁用代理转发：

```
# 仅对 SSH2  
ForwardAgent no
```

也可以使用客户端来启用或禁用代理转发。[\[6.3.5.3\]](#)

显然代理转发非常方便，但在安全性要求很高的环境中，最好禁用这种特性。因为转发的代理是使用 Unix 域套接字来实现的，攻击者可以很容易地访问这些代理。这些套接字实际上就是文件系统中的一些节点，只能使用文件权限进行保护。

例如，假设你维护了一个网路，其中有很多不可信的主机，你要从一个更安全的网络中使用 SSH 连接到这些主机上。你可以考虑在不可信的主机上禁用代理转发。否则，攻击者可能成功地侵入一台不可信主机；并控制了对来自合法的 SSH 连接的代理转发；他使用已经加载到代理中的密钥就可以通过 SSH 访问网络。（但是攻击者并不能从中提取出密钥。）

5.4.3.11 转发

SSH 的转发或隧道特性通过加密连接，对其他基于 TCP/IP 的应用程序进行保护。我们会在第九章中详细介绍转发，但是在本章中我们会介绍一下几个配置关键字，它们可以用来启用和禁用转发。

TCP 端口转发可以通过把关键字 `AllowTcp-Forwarding` 设置成 `yes`（缺省值）或 `no` 来启用或禁用：

```
# SSH1, SSH2, OpenSSH
```

```
AllowTcpForwarding no
```

或者用户可以指定特定的用户或组来使用转发特性：

```
# 仅对 SSH2
AllowTcpForwardingForUsers smith jones roberts
AllowTcpForwardingForGroups students faculty
DenyTcpForwardingForUsers badguys
DenyTcpForwardingForGroups bad*
```

X 转发可以使用关键字 X11Forwarding (SSH1、SSH2、OpenSSH)、ForwardX11 或 AllowX11Forwarding (SSH2 中 x11Forwarding 的同义词) 单独启用或禁用。它们的缺省值都是 yes，也就是启用转发特性：

```
# SSH1, SSH2, OpenSSH
X11Forwarding no

# 仅对 SSH2：均可生效
ForwardX11 no
AllowX11Forwarding no
```

5.4.4 服务器密钥的生成

所有的 SSH 服务器都要维护一个主机密钥，这个主机密钥是由系统管理员在安装 SSH 时生成的，以后便一直在认证中用来标识服务器主机的身份。[5.4.1.1]

SSH-1 服务器有些特殊，它在运行时要维护另外一个密钥，称为服务器密钥，该密钥用来对客户端 / 服务器之间的通信进行保护。这个密钥是临时的，永远不会保存到磁盘上。服务器在启动时生成这个密钥（位数），并以固定的周期重新生成这个密钥。SSH1 和 OpenSSH 都可以指定服务器密钥的长度。密钥的缺省长度是 768 位，最小为 512 位，用户可以使用 ServerKeyBits 关键字来指定服务器密钥的长度：

```
# SSH1, OpenSSH
ServerKeyBits 1024
```

也可以使用 -b 命令行选项：

```
# SSH1, OpenSSH
$ sshd -b 1024
```

用户还可以指定服务器密钥的生命期，或称为重新生成服务器密钥的间隔。当生命期结束时，系统会生成另外一个服务器密钥，如此周而复始，例如，每 10 分钟循环

一次。这是一种安全特性：如果入侵者截获了一个服务器密钥，他也只能在一段有限的时间内（在我们的例子中是10分钟）使用这个密钥来解密所传输的数据。同理，如果加密过的传输数据被探测器捕获了，那么10分钟之后，用来解密该会话的服务器密钥也将被销毁。

密钥重生的周期是以秒为单位指定的，缺省值为3600秒钟（1小时）。这个周期可以使用 `KeyRegeneration-Interval` 关键字来指定：

```
# SSH1, OpenSSH  
KeyRegenerationInterval 1200
```

也可以使用 `-k` 命令行选项：

```
# SSH1, OpenSSH  
$ sshd -k 1200
```

该值为0就关闭了密钥重生的特性：

```
# SSH1, OpenSSH  
KeyRegenerationInterval 0
```

或者：

```
# SSH1, OpenSSH  
$ sshd -k 0
```

`RekeyIntervalSeconds`关键字规定了`sshd2`每隔多长时间和客户端执行一次密钥交换，从而替换会话数据加密和完整性检测所使用的密钥。缺省值为3600秒（1小时），该值为0表示禁用生成密钥的特性（注4）：

```
# 仅对 SSH2  
RekeyIntervalSeconds 7200
```

5.4.5 加密算法

SSH服务器可以在安全连接中使用很多数据加密算法；客户端可以从服务器支持的

注4： 注意，如果你想让很多其他SSH客户端都可以使用这种设置，那么现在就必须在SSH2服务器中禁止重新生成会话密钥。这是因为SSH不支持会话重新生成密钥，一旦重新生成的时间间隔超时，连接就会因错误而中断。

算法中选择一种。SSH2有一个服务器配置选项可以设置所有可用的算法，这些算法是从服务器软件所支持的算法中挑选出来的。Ciphers关键字就是用于这个目的，其值可以有两种形式：

- 使用逗号分隔开的算法名列表（字符串），指明可以使用哪些算法。下表给出了可以使用的值。

值	含义
3des-cbc	3DES (Triple DES) 算法
blowfish-cbc	Blowfish 算法
twofish-cbc	TwoFish 算法
arcfour	ARCFOUR 算法
none	不用加密

只有 SSH 是使用 --with-none 标志编译的，才可以不用加密算法。-cbc 后缀说明算法使用块链 (block chaining)。这些算法都属于块算法 (block cipher)，这种算法可以使用很多模式；CBC 只是其中之一。

- 一个字符串，说明一组算法。下表给出了所支持的值：

值	含义
none	不加密传输
any	服务器中实现的任何算法，包括 none
anycipher	和 any 相同，但是不包括 none
anystd	IETF SecSH 草案中给出的任何标准算法（假设在服务器中已经实现了这种算法），包括 none
anystdcipher	和 anystd 相同，但是不包括 none

下面是几个例子：

```
# SSH2, OpenSSH/2
Ciphers 3des-cbc
Ciphers 3des-cbc,blowfish-cbc,arcfour
Ciphers any
```

单个算法和一组算法不能混合使用：

```
# 此为非法
Ciphers 3des,anystd
```

`Ciphers` 关键字非常有用，可以快速禁用单个加密算法，也就是说，如果我们发现了某个加密算法有安全漏洞就可以这样使用。此时只需要把这个算法从 `Ciphers` 列表中删除并重新启动服务器即可。

用户可以在编译时就指定 SSH1 服务器禁用某些加密算法。[\[4.1.5.6\]](#) 特别是缺省编译不应该包含 `none` 算法类型。这是一种安全手段，可以让不安全的 SSH 会话更难建立。否则，如果攻击者短时间内获得了对用户账号的访问权限，那么他可以在用户的 SSH 客户端配置文件中加入 “`Ciphers none`”。用户可能永远都不会觉察到这点微小的改变，但是以后所有的 SSH 连接都不安全了（注 5）。

用户最好只在测试时使用 `none` 算法。使用 SSH-1 协议而不用加密算法会出现严重的缺陷：不但丧失了数据的私密性，而且同时也丧失了服务器认证和完整性保护的功能。SSH-2 则不会受到这些问题的影响。但是不管是哪种情况，都不能再使用密码认证了，因为此时密码是以明文形式传递的。

5.4.5.1 MAC 算法

`MAC` 关键字可以让用户选择 `sshd2` 进行完整性检测所使用的算法，称为消息认证代码（Message Authentication Code），用于 `sshd2`。[\[3.2.3\]](#) 可以使用的算法有：[\[3.9.3\]](#)

```
hmac-sha1
hmac-md5
hmac-md5-96
```

下表给出了该关键字可以使用的几个特殊用法：

值	含义
any	所有支持的算法
anymac	所有支持的算法，不包括 <code>none</code>
anystd	所有的标准算法；也就是说 SSH-2 协议当前工作草案中所定义的算法

注 5：如果用户真的不使用任何加密算法连接，那么 `ssh` 就会打印这样一条警告信息，“`WARNING: Encryption is disabled!`” 即便如此，攻击者也可以在客户端中设置 `QuietMode` 关键字来禁止显示这条消息。[\[5.8.1.3\]](#)

值	含义
anystdmac	和 anystd 相同，但是不包括 none
none	不使用 MAC；这是不安全的

5.4.6 SSH 协议的选择

在 OpenSSH 中，用户可以使用 `Protocol` 关键字选择支持 SSH-1 或 SSH-2，还是对二者都提供支持。该值可以为 1（用于 SSH-1，缺省值）、2（用于 SSH-2），或者是“1, 2”：

```
# 仅对OpenSSH  
Protocol 1,2
```

5.5 允许用户登录：认证和访问控制

SSH 服务器相当大的一部分工作就是对来自客户端的连接请求进行授权或拒绝其连接。这是在两个层次上实现的：认证和访问控制（也就是授权）。

正如我们已经看到的一样，认证负责对发起连接请求的用户的身份进行验证。访问控制负责允许或禁止来自特定用户、机器或 Internet 域的 SSH 连接到服务器上。

5.5.1 认证

`sshd` 支持使用几种不同的技术进行认证，这些技术都可以启用或禁用。[\[3.1.3\]](#)[\[3.4.2\]](#) 例如，如果不信任密码认证，那么可以在服务器范围内将其关闭，但同时仍然可以使用公钥认证。

随着 SSH 的不断发展，对认证进行配置的语法也已经多次发生了变化。我们不仅要介绍现有的关键字，而且还会介绍一些现在已经舍弃不用的关键字，这样即使运行的 `sshd` 版本较老也能了解相应的信息。

在 SSH1 和 OpenSSH 中，各种认证技术都是使用以下形式的关键字来启用或禁用的：

`Name_Of_Technique Authentication`

例如，密码认证是由 Password-Authentication 关键字控制的，RSA 公钥认证是由 RSA-Authentication 控制的，依此类推，每种技术使用一个关键字。该值可以是 yes 或 no：

```
# SSH1, OpenSSH; SSH2 中不推荐  
RSAAuthentication yes
```

SSH2 早期的版本还为每种认证技术都使用一个关键字，但是使用关键字代表一组算法的用法更为通用。它不使用 RSAAuthentication（指 RSA 算法），而是使用 PublicKeyAuthentication，这样并没有指定一种特定的算法。

```
# 仅对 SSH2，但不推荐  
PubkeyAuthentication yes
```

这样就为支持其他公钥算法留下了扩展的余地。原来诸如 RSAAuthentication 之类的关键字仍然可以使用，它们被用作是这些更通用的关键字的同义词。

现在 SSH2 的语法已经完全不同了。它并不会为每种技术都创建一个新关键字，而是只使用两个关键字：AllowedAuthentications 和 RequiredAuthentications，每个关键字后面都可以跟上一种或多种认证技术的名字，例如：

```
# 仅对 SSH2；推荐技术  
AllowedAuthentications password,hostbased,publickey
```

AllowedAuthentications 说明可以使用哪种技术连接到这个 SSH 服务器上（注 6）；反之，RequiredAuthentications 规定了必须使用的认证技术（注 7）。下面的配置说明服务器在允许建立连接之前要求使用公钥认证和密码认证：

```
# 仅对 SSH2；推荐技术  
AllowedAuthentications publickey,password  
RequiredAuthentications publickey,password
```

它是 AllowedAuthentications 的一个子集：必需的技术一定也是可以使用的技术。缺省情况下，sshd2 只允许使用密码认证和公钥认证。

注 6：此处的顺序并不重要，因为客户端可以控制认证过程。

注 7：在 SSH2 2.0.13 中不再使用 RequiredAuthentications 关键字，这样就会导致认证总是失败。这个问题在 2.1.0 中已经修正了。

如果仔细考虑一下这个问题，就会发现这些关键字很容易混淆，至少是不好选择。实际上，如果一直使用 RequiredAuthentications，其值几乎总是会和 AllowedAuthentications 相同：没有什么地方会说一种方法是允许使用的，但却不是必需的；否则这种方法就不能进行认证并成功获得一个连接。让用户可以指定多个可用方法子集会使这种特性更有用，这些方法子集可以结合使用，共同用于客户端的认证。

表 5-1 给出了和认证有关的关键字。

表 5-1：与认证有关的关键字

类型	SSH1	OpenSSH	新 SSH2	老 SSH2
AllowedAuthentications	无	无	有	无
DSSAuthentication	无	有 ^a	无	无
KerberosAuthentication	有	有	无	无
PasswordAuthentication	有	有	不建议使用	有
PubKeyAuthentication	无	无	不建议使用	有
RequiredAuthentications	无	无	有	无
RhostsAuthentication	有	有	无	有
RhostsPubKeyAuthentication	无	无	无	有
RhostsRSAAuthentication	有	有	无	有
RSAAuthentication	有	有	不建议使用	有
SKeyAuthentication	无	有	无	无
TISAAuthentication	有	有 ^b	无	无

a. 仅对于 SSH-2 协议。

b. 实际上是 S/Key 认证，而不是 TIS。

现在我们介绍一下如何启用和禁用各种认证方法。

5.5.1.1 密码认证

密码认证使用登录密码作为一种身份证明。[3.4.2.1]在 SSH1 和 OpenSSH 中，密码认证可以把 PasswordAuthentication 关键字设置成 yes（缺省值）或 no 来启用或禁用：

```
# SSH1, OpenSSH; SSH2 中不推荐  
PasswordAuthentication yes
```

SSH2 使用 PasswordAuthentication，但是我们并不提倡使用这个关键字；而是应该使用 AllowedAuthentications 关键字，并在后面跟上一个 password：

```
# 仅对 SSH2  
AllowedAuthentications password
```

通常密码认证使用普通登录密码。但是这也也可以使用编译时配置进行修改。对于 SSH1 来说，如果 Kerberos 或 SecurID 的支持已经被编译到了服务器中，那么密码认证就会变成 Kerberos [5.5.1.7] 或 SecurID。[5.5.1.9]

5.5.1.2 公钥认证

公钥认证使用加密密钥对用户的身份进行验证。[3.4.2.2] 在 SSH1 和 OpenSSH/1 中，公钥认证使用 RSA 加密，我们可以使用 RSAAuthentication 关键字来启用或禁用这种特性。该值可以为 yes（缺省值）或 no：

```
# SSH1, OpenSSH; SSH2 中不推荐  
RSAAuthentication yes
```

关键字 RSAAuthentication 和听起来更通用的 PublicKeyAuthentication 关键字一样，都用于 SSH2，二者功能相同，但是这两个都不建议使用。我们建议使用 AllowedAuthentications 关键字，并在后面设置一个值 publickey：

```
# 仅对 SSH2  
AllowedAuthentications publickey
```

OpenSSH 使用 DSAAuthentication 关键字为 SSH-2 连接提供公钥认证：

```
# 仅对 OpenSSH/2  
DSAAuthentication yes
```

公钥认证对于大部分 Unix 的 SSH 实现都可以进行详细的配置。有关为每个账号定制认证的详细内容请参看第八章。

5.5.1.3 Rhosts 认证

可信主机认证通过检查远程主机名和相关用户名而实现对 SSH 客户端身份的验证。

[3.4.2.3]在SSH1和OpenSSH中，可以支持两种可信主机认证。安全性差一点的可信主机认证模拟了Berkeley的r-命令(*rsh*、*rcp*、*rlogin*)的行为，它们检查服务器文件*/etc/hosts.equiv*和*~/.rhosts*用于认证权限的限制，并使用网络名字服务(例如，DNS、NIS)和特权TCP源端口来验证客户端的身份。SSH2不支持这种不安全的技术。

Rhosts认证是通过把*RhostsAuthentication*关键字设置成yes(缺省值)或no来启用或禁用的：

```
# SSH1, OpenSSH
RhostsAuthentication yes
```

虽然Rhosts认证可能很有用，但不幸的是它允许连接使用不安全的r-命令发起连接，因为它们使用的限制文件的权限相同。要避免这种潜在的安全风险，可以使用SSH专用的文件*/etc/shosts.equiv*和*~/.shosts*文件，并删除*/etc/hosts.equiv*和*~/.rhosts*文件。还可以使用关键字*IgnoreRhosts*通知SSH服务器忽略用户的*.rhosts*和*.shosts*文件；该值可以是yes(忽略这两个文件)或no(缺省值)：

```
# SSH1, SSH2, OpenSSH
IgnoreRhosts yes
```

提示：虽然该关键字名字中包含“Rhosts”，但是要记住它也可以对*.shosts*文件起作用。而且，虽然可以使用*Ignore-Rhosts*来忽略用户文件，但是*/etc/hosts.equiv*和*/etc/shosts.equiv*文件依然有效。

SSH1和SSH2也可以对root用户的Rhosts认证单独进行控制。关键字*IgnoreRootRhosts*允许或禁止用户使用超级用户的*.rhosts*和*.shosts*文件，该值可以覆盖*IgnoreRhosts*的设置：

```
# SSH1, SSH2
IgnoreRootRhosts yes
```

该值可以是yes(忽略文件)或no(不忽略文件)。如果没有明确指明，*IgnoreRootRhosts*默认和*IgnoreRhosts*值相同。例如，用户可以这样允许使用其他用户的*.rhosts*文件，但是不能使用root的*.rhosts*文件：

```
# 仅对SSH1
IgnoreRhosts no
```

```
IgnoreRootRhosts yes
```

用户也可以忽略其他用户的*.rhosts*文件，而只使用root用户的*.rhosts*文件：

```
# 仅对SSH1
IgnoreRhosts yes
IgnoreRootRhosts no
```

同样，*IgnoreRootRhosts*也不会考虑*/etc/hosts.equiv*和*/etc/shosts.equiv*的内容而停止服务器。为了更加安全，最好是彻底禁止访问*.rhosts*文件。

服务器机器的环境（例如，DNS、NIS以及静态主机文件中各项的次序）的其他方面可能会使*Rhosts*认证更加复杂。这也可能会导致新的系统漏洞。[\[3.4.2.3\]](#)

5.5.1.4 强可信主机认证

SSH1、SSH2和OpenSSH还都支持第二种安全性更高的可信主机认证。对于SSH1和OpenSSH/1来说，这种认证称为*RhostsRSA*认证；对于SSH2来说，称为基于主机的认证（注8）。在这两种认证中，都通过对主机密钥进行加密测试，从而对*rhosts*认证中安全性较差的部分进行了增强。[\[3.4.2.3\]](#) */etc/hosts.equiv*和*~/.rhosts*（以及SSH专用的*/etc/shosts.equiv*和*~/.shosts*）文件依然有效，但是不能通过这种测试。

SSH1和OpenSSH使用关键字*RhostsRSAAuthentication*（可真奇怪！）来启用或禁用这种认证：

```
# SSH1, OpenSSH; SSH2中不推荐
RhostsRSAAuthentication yes
```

*sshd2*可以使用*RhostsRSAAuthentication*关键字，也可以使用听起来更通用的*RhostsPubKeyAuthentication*关键字，二者功能相同；但是，这两个关键字都已经要考虑舍弃不用了。我们现在使用*AllowedAuthentications*关键字来实现相同的功能，可以将其设置成*hostbased*：

```
# 仅对SSH2
AllowedAuthentications hostbased
```

注8：OpenSSH 2.3.0也不支持对SSH-2连接使用基于主机的认证。

5.5.1.5 提取已知名主机的公钥

sshd2 需要自己使用基于主机的认证所接受的连接的所有主机的公钥。这些公钥保存在 */etc/ssh2/knownhosts* 目录中的各个文件中。在主机请求建立一个连接时，就从这个目录中提取出这个主机的公钥。服务器还可以搜索目标用户账号的 *~/.ssh2/knownhosts* 目录。这种可选特性是使用 *UserKnownHosts* 关键字来启用的，该值可以是 *yes*（缺省值）或 *no*：

```
# 仅对 SSH2
UserKnownHosts no
```

OpenSSH 支持相同的功能，但是用法正好相反，它使用 *IgnoreUserKnownHosts* 关键字来设置。该值为 *yes* 表明要忽略用户的已知名主机数据库；该值缺省为 *no*：

```
# 仅对 OpenSSH
IgnoreUserKnownHosts yes
```

在安全性要求非常高的环境中，让 *sshd* 查阅用户的已知名数据库是不能接受的一件事。因为基于主机的认证要依赖于客户端主机的完整性和正确的管理，所以系统管理员通常只会把可以使用基于主机认证的特权授权给有限的可信主机。但是，如果我们使用了用户的文件，那么这些用户就会把这种信任关系扩充，其中就可能包括不安全的远程主机。然后攻击者可以：

1. 攻击这个不安全的远程主机。
2. 模仿远程主机上的用户。
3. 通过 SSH 访问用户的本地账号，而不需要密钥口令或本地账号密码。

5.5.1.6 PGP 认证

PGP（Pretty Good Privacy）是另外一种使用公钥认证的安全产品。^[1.6.2] PGP 密钥和 SSH 密钥的实现不同也不能互换。但是，SSH2 最新的版本现在已经可以支持使用 PGP 密钥认证，它遵守 OpenPGP 标准。是的，用户可以使用自己喜欢的 PGP 密钥来保护 SSH2 服务器（只要密钥文件是 OpenPGP 兼容的就可以；有些 PGP 密钥，特别是老版本的软件所创建的 PGP 密钥，都是不兼容的）。在发布时，这种特性还处于文档标准化阶段。下面我们介绍一下它的工作原理。

首先，需要在服务器和客户端都安装 SSH2 2.0.13 或更高的版本，或者是 F-Secure 的相应版本。而且，它们必须都是使用 `--with-pgp` 编译时标志进行编译的，即包含了 PGP 的支持。[\[4.1.5.7\]](#)

还需要在客户端机器上构建 PGP 密钥环，以及 SSH2 客户端可以使用的认证密钥。以下是具体步骤：

1. 把 PGP 密钥环拷贝到用户账号中的 SSH2 目录 `~/.ssh2` 中。假设将其命名为 `secring.pgp`。

2. 在标识文件 (`~/.ssh2/identification` 或用户选定的其他文件) 中使用 `PgpSecretKeyFile` 关键字指明密钥环：

```
# 仅对 SSH2  
PgpSecretKeyFile secring.pgp
```

3. 标识要用于认证的 PGP 密钥。这可以使用以下三个关键字之一来实现：

- 要使用名字来标识密钥，可以使用 `IdPgpKeyName`:

```
# 仅对 SSH2  
IdPgpKeyName mykey
```

- 要使用 PGP 指纹 (fingerprint) 来标识密钥，可以使用 `IdPgpKeyFingerprint`:

```
# 仅对 SSH2  
IdPgpKeyFingerprint 48 B5 EA 28 80 5E 29 4D 03 33 7D 17 5E 2E CD 20
```

- 要使用密钥 ID 来标识密钥，可以使用 `IdPgpKeyId`:

```
# 仅对 SSH2  
IdPgpKeyId 0xD914738D
```

对于 `IdPgpKeyId` 来说，前面的 `0x` 是必须的，它说明该值是十六进制。用户可以不使用前缀 `0x`，而是以十进制形式给出该值，但是由于 PGP 就是以十六进制形式显示数据的，大家可能并不希望这样做。

用户需要在服务器机器上（比如说 `server.example.com`）构建自己的公钥环，以及用于 SSH2 服务器认证所需要的公钥：

1. 把公钥环从客户端拷贝到服务器上。（注意这是一个密钥环，而不是一个公钥。）把这个密钥环放到服务器上用户的 `~/.ssh2` 目录中。假设将其命名为 `pubring.pgp`。

2. 在认证文件`~/.ssh2/authorization`中，使用关键字`PgpPublicKeyFile`来标识这个公钥环。

```
# 仅对 SSH2  
PgpPublicKeyFile pubring.pgp
```

- 3.. 和客户端的标识文件一样，使用名字、指纹或密钥 ID 来标识公钥。相应的关键字则有些不同：分别是`PgpKeyName`、`PgpKeyFingerprint`和`PgpKeyId`。（标识文件的关键字都以“Id”开头。）

```
# 仅对 SSH2 : 使用其中一个  
PgpKeyName mykey  
PgpKeyFingerprint 48 B5 EA 28 80 5E 29 4D 03 33 7D 17 5E 2E CD 20  
PgpKeyId 0xD91473BD
```

现在成功了！从客户端上发起了一个 SSH2 会话。假设创建了另外一个标识文件来使用 PGP 认证，称为`~/.ssh2/idpgp`，其中包含了`PgpSecretKeyFile`和其他一些内容。使用`-i`标志来指明这个文件并初始化一个连接：

```
$ ssh2 -i idpgp server.example.com
```

如果各种设置都很正确，就会提示输入 PGP 口令：

```
Passphrase for pgp key "mykey":
```

输入 PGP 口令后，认证将会成功。

5.5.1.7 Kerberos 认证

SSH1 和 OpenSSH 都可以使用 Kerberos 作为一种认证机制（注 9）。我们在这里总结一下和 Kerberos 有关的配置关键字，并在以后对这个主题进行更详细的介绍。[11.4]在本书出版时，SSH2 2.3.0 版本刚好发行，其中包含了对 Kerberos-5 的“试验性”支持，我们在此就不再详细讨论了。

首先要注意只有在编译时启用了这种支持，才能使用 Kerberos 认证。除非使用了`--with-kerberos5` (SSH1) 或者`--with-kerberos4` (OpenSSH) 配置选项，否则`sshd` 就不能支持 Kerberos。

注 9： SSH1 和 OpenSSH 使用的 Kerberos 版本不同：SSH1 使用 Kerberos-5，而 OpenSSH 使用 Kerberos-4。

假设服务器支持 Kerberos，那么我们可以把关键字 KerberosAuthentication 设置成 yes 或 no 来启用或禁用 Kerberos 认证：

```
# SSH1, OpenSSH
KerberosAuthentication yes
```

如果 Kerberos 支持已经被编译进了服务器，那么该值缺省值就是 yes；否则，缺省值就是 no。

如果服务器中还启用了密码认证，那么连接可以使用 Kerberos 认证，也可以使用密码认证（由 Kerberos 服务器进行认证）：

```
# SSH1, OpenSSH
KerberosAuthentication yes
PasswordAuthentication yes
```

sshd 并不对本地登录密码进行检查，而是为用户请求 Kerberos TGT，如果证书和密码匹配，就允许用户登录（注 10）。它把 TGT 保存在用户的证书缓存中，这样就不需要另外执行一次 *kinit*。

如果 Kerberos 验证密码失败，那么服务器就可以使用普通的密码认证再对这个密码进行验证。这在使用 Kerberos 的环境中是很有用的，但并不是每个用户都会使用这种特性。要启用这个选项，可以把 KerberosOrLocalPasswd 关键字设置成 yes；该值缺省为 no：

```
# SSH1, OpenSSH
KerberosOrLocalPasswd yes
```

最后，关键字 KerberosTgtPassing 控制 SSH 服务器是否执行 Kerberos TGT 转发：

```
# SSH1, OpenSSH
KerberosTgtPassing yes
```

其缺省值的规则和 KerberosAuthentication 相同：如果 Kerberos 支持被编译进了服务器，那么缺省值是 yes，否则就是 no。

OpenSSH 增加了一个关键字 KerberosTicketCleanup，它负责在退出时检测用户的 Kerberos 证书缓存。该值可以是 yes 或 no，缺省值是 yes，意思是执行这种检测：

注 10：作为一种防止欺骗的措施，这需要把一个主机的证书成功授权给本地主机。

```
# 仅对OpenSSH  
KerberosTicketCleanup yes
```

5.5.1.8 TIS 认证

SSH1服务器可以通过可信信息系统（TIS）的Gauntlet防火墙工具包对用户进行认证。当SSH客户端试图通过Gauntlet进行认证时，SSH服务器就和Gauntlet的认证服务器*authsrv*进行通信，并把*authsrv*的请求转发到客户端，然后把客户端的响应转发给*authsrv*。

TIS认证是一个编译时选项，由`--with-tis`配置标志进行控制。[\[4.1.5.7\]](#)假设这种支持已经编译进了*sshd*，那么TIS认证就可以通过把TISAuthentication设置成yes或no（缺省值）来启用或禁用：

```
# 仅对SSH1  
TISAuthentication yes
```

有关TIS认证的更详细内容请参看SSH1发行版本中的*README.TIS*文件。有关可信信息系统和*authsrv*的其他信息可以在这些地方找到：

<http://www.tis.com/>
<http://www.msg.net/utility/FWTK/>
<http://www.fwtk.org/>

5.5.1.9 SecurID 认证

Security Dynamics的SecurID是一种基于硬件的认证技术。用户需要一个物理卡片用于认证，这个卡片称为SecurID卡。该卡中包含了一个微芯片，它（在一个小LCD上）显示一个整数，这个整数会周期性地变化。要进行认证，用户必须提供这个整数，同时提供一个密码。有些版本的SecurID卡还有一个小键盘，可以让用户输入密码，这样就实现了两层加密。

如果SSH1服务器在编译时使用`--with-securid`标志加入了对SecureID的支持，那么密码认证都会转换成SecurID认证。[\[4.1.5.7\]](#) 用户必须提供它们的卡片中现在显示的整数才能进行认证。

5.5.1.10 S/Key 认证

S/Key 是 Bellcore 发明的一次性密码系统，只有 OpenSSH 可以将其作为一种 SSH 认证方法。“一次”是说每次你认证时，用户都需要输入一个不同的密码；这样有助于防止攻击，因为即使密码被截获也没什么用处。它是这样工作的：

1. 当用户连接到一个远程服务上时，它会向用户提供一个整数和一个字符串，分别称为序列号和密钥。
2. 用户在本地机器中把这个序列号和密钥输入 S/Key 计算程序中。
3. 用户还要向这个计算程序输入一个口令，只有用户自己才知道这个口令。它不会通过网络传输，只会进入你本地机器上的计算程序，因此安全性是有保证的。
4. 计算程序根据你输入的这三个数据，产生一次性密码。
5. 输入密码并通过远程服务的认证。

如果设置了 `SkeyAuthentication` 关键字，那么 OpenSSH 服务器也可以支持 S/Key 认证。该值缺省为 `yes`，就是要支持 S/Key 认证。要关闭这种特性，可以把该值修改成 `no`。

```
# 仅对 OpenSSH  
SkeyAuthentication no
```

有关一次性密码的更多信息可以在这儿找到：

<http://www.ietf.cnri.reston.va.us/html.charters/otp-charter.html>

5.5.1.11 PAM 认证

Sun Microsystems 的可插入认证模块 (PAM) 系统是用于支持多种认证方法的一种框架方案。通常在出现新的认证机制时，程序都需要重新编写才能使用这些新机制。PAM 就消除了这种烦恼。程序在编写时都支持 PAM，这样新的认证机制都可以在运行时就插入，而不用重新修改源代码。PAM 更多的信息可以在这儿找到：

<http://www.sun.com/solaris/pam/>

OpenSSH也包括了对PAM的支持。SSH1 1.2.27已经由第三方集成了PAM，但是这种用法要修改SSH1的源代码。详细信息请参看：

http://diamond.rug.ac.be/sshd_PAM/

5.5.1.12 AFS 令牌传递

Andrew文件系统（AFS）是一种分布式文件系统，其目的和NFS类似，但是更可靠、更可扩充。它使用一个Kerberos 4协议的修改版本进行认证。OpenSSH可以使用`--with-afs`和`--with-kerberos4`编译时包含AFS的支持。关键字`AFSTokenPassing`就负责控制这种特性，该值可以是yes（接受转发的令牌，缺省值）或no：

```
# 仅对OpenSSH
KerberosAuthentication yes
KerberosTGTPassing     yes
AFSTokenPassing        yes
```

`AFSTokenPassing`让OpenSSH在远程主机上根据用户在客户端上现有的证书（也就是前面使用`klog`或`kinit`所获得的证书）建立Kerberos/AFS证书。这对于在AFS环境中使用OpenSSH是一个必要的条件，而不仅仅是一种便利的用法：例如，用户的远程主目录在AFS上，那么`sshd`就需要AFS证书才能访问远程的`~/.ssh`目录，从而执行公钥认证。在这种情况下，用户也可能还需要使用AFS工具来调整远程`~/.ssh`目录中的权限，从而允许`sshd`来读取所需要的内容。一定要确保其他用户不能读取自己的敏感文件（`~/.ssh/identity`，其他私钥文件，以及`~/.ssh/random_seed`）。有关AFS的更多信息，请访问：

http://www.atw.nih.gov/Docs/AFS/AFS_toc.html

<http://www.faqs.org/faqs/afs-faq/>

5.5.2 访问控制

服务器范围的访问控制可以允许或拒绝来自特定主机、Internet域或服务器主机上的特定用户账号的连接。其用法和认证无关。例如，即使一个用户的身份是合法的，你也可能希望拒绝来自该用户的连接。同理，如果特定的计算机或Internet域的安全策略很差，那么你也可能希望拒绝接收来自该域的所有SSH连接请求。

SSH 访问控制的文档很少，大多都非常精简。配置关键字的意义看起来似乎显而易见的，但实际上并非如此。在本节中，我们主要的目的是说明一些问题所在，这样就可以正确而有效地配置访问控制。

记住只有在服务器和账号都配置允许访问时，才允许 SSH 访问。如果服务器可以接受自己所有用户的 SSH 连接，那么每个用户仍能拒绝 SSH 连接连到自己的账号。[\[8.2\]](#)同理，如果一个账号配置成允许 SSH 访问，那么该主机上的 SSH 服务器可以禁止这种访问权限。这种两层的系统可以适用于所有的 SSH 访问控制，因此我们就不再重复了。[图 5-2](#) 对两层访问控制系统进行了总结（[注 11](#)）。

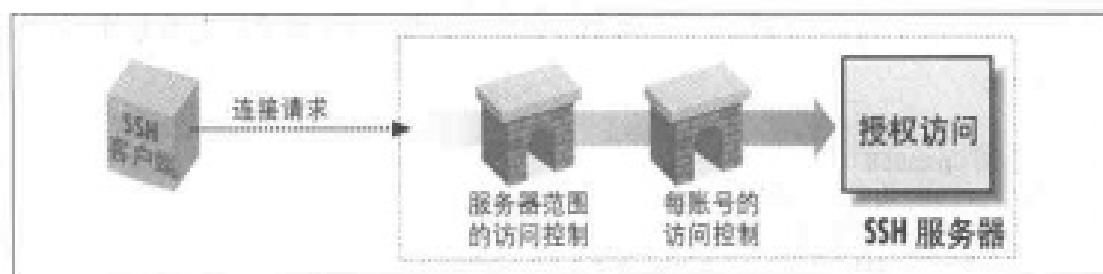


图 5-2：访问控制层次

5.5.2.1 账号访问控制

通常，只要设置正确，任何账号都可以接收 SSH 连接。这种访问权限可以使用服务器关键字 `AllowUsers` 和 `DenyUsers` 覆盖。`AllowUsers` 说明只有一部分本地账号可以接收连接。例如，下面这行：

```
# SSH1, SSH2, OpenSSH
AllowUsers smith
```

就允许本地的 `smith` 账号接收 SSH 连接，而且只允许 `smith` 接收连接。配置文件可以有多个 `AllowUsers` 行：

```
# SSH1, SSH2, OpenSSH
AllowUsers smith
AllowUsers jones
AllowUsers oreillystat
```

注 11：这个概念可以适用于我们在本节中讨论的配置关键字，但是不能适用于可信主机控制文件，即 `~/.rhosts` 和 `/etc/hosts.equiv`。这两个文件的设置会彼此覆盖。[\[3.4.2.3\]](#)

这样结果就会一直累积：本地账号smith、jones和oreilly可以接收SSH连接，而且只有这些账号可以接收连接。SSH服务器维护了一个所有的AllowUsers值的列表，在连接请求到达时，它会对该列表进行字符串比较操作（我们很快就会看到，这实际上是一个模式匹配）。如果匹配成功，那么就允许这个连接；否则，就拒绝这个连接。

警告：配置文件中如果单独出现一个AllowUsers关键字，后面没有任何内容，就表示禁止所有没提到的用户接收SSH连接。如果配置文件中没有AllowUsers关键字，那么服务器的AllowUsers列表就为空，任何用户都可以接收SSH连接。

DenyUsers正好和AllowUsers相反：它禁止特定的账号接收连接。例如：

```
# SSH1, SSH2, OpenSSH
DenyUsers smith
```

说明smith账号不能接收SSH连接。DenyUsers关键字也可以出现多次，这和AllowUsers一样，其影响也可以累积。和AllowUsers相同，服务器也为所有的DenyUsers值维护了一个列表，在有连接请求到达时，也要进行字符串比较。

AllowUsers和DenyUsers都可以使用更复杂的值，而不仅仅是单个账号名。sshd1和sshd2都支持一种有趣但可能会引起混淆的语法，用来指定一个账号名和一个主机名（或者IP地址），二者之间使用@符号分隔开：

```
# SSH1, SSH2
AllowUsers jones@example.com
```

不管这个字符串看起来像什么，它既不是一个email地址，也不表示example.com机器上的jones用户，而是说明了本地账号jones和远程客户端机器example.com的对应关系。其确切的意义是：“客户端example.com可以连接到服务器的jones账号中”。这种意思比较奇怪，如果jones是一个远程账号，那就更奇怪了，因为SSH服务器无法对账号名和远程客户端主机进行验证（除非它使用的是基于主机的认证）。

对于SSH1和OpenSSH来说，在AllowUsers和DenyUsers参数中的用户名和主机部分都可以使用通配符。“?”表示除“@”之外的任意一个符号；“*”表示多个字符，其中也不包括“@”。对于SSH2来说，用户可以使用所有的规则表达式，但是语法稍微有些不同；请参看附录一。

警告：SSH2规则表达式语言包括一些其中包含冒号(:)的关键字，例如[:digit:]。在SSH2访问控制模式中使用冒号会产生一个可怕而又难以跟踪的问题：它会被忽略，同时用户的配置文件中其余的部分也会被忽略！这个问题是由于解析器的一个缺陷，它会把冒号当作是要开始一个文件中有标号的段。标号不能和任何内容匹配，这样该段中没有包含的文件的其他部分就这样被忽略了。用户可以简单地在模式中使用引号来解决这个问题：

```
AllowHosts "10.1.1.[:digit:]##"
```

虽然没有文档说明，但是这种引号的语法的确可以正常工作。

下面我们给出一些例子。以下设置只允许账号名是5个字符而且以“mith”结尾的用户接收SSH连接：

```
# SSH1, SSH2, OpenSSH
AllowUsers ?mith
```

以下设置只允许账号名以“s”开始的用户接收来自主机名以“.edu”结尾的SSH连接：

```
# SSH1, SSH2, OpenSSH
AllowUsers s*@*.edu
```

以下设置只允许账号形式为“testN”（其中N是一个数字，例如“test123”）的用户接收SSH连接：

```
# 仅对SSH2
AllowUsers test[0-9]##
```

有一件非常不幸的事情，就是用户不能再使用传统的“地址/掩码长度”语法来指定IP网络了，例如，原来可以使用10.1.1.0/28来表示10.1.1.0到10.1.1.15。现在要使用AllowHosts[5.5.2.3]限制这个范围的IP地址就非常繁琐：

```
# SSH1
AllowHosts *@10.1.1.2 *@10.1.1.10 *@10.1.1.11 *@10.1.1.12 *@10.1.1.13
AllowHosts *@10.1.1.14 *@10.1.1.15
```

或者更加晦涩：

```
# SSH2
AllowHosts *@10.1.1.(?|(1[0-5]))
```

当然，如果一个网络的边界刚好是 8 的整数倍，那么对其进行限制就容易多了：

```
# SSH1, SSH2  
# 只允许连接来自 10.1.1.0/24  
AllowHosts *@10.1.1.*
```

但是要注意，这种限制很容易就可以绕开；攻击者只需要控制一个域名服务器并从一个名为 10.1.1.evil.org 的机器上登录上来就可以了。更有效的语法是：

```
# 仅对 SSH2  
AllowUsers *@10.1.1.[:isdigit:]##"
```

即使这样仍然还不够可靠。地址和基于主机名的限制最多也只提供了很脆弱的限制而已；这些只能用作强认证方法的一种辅助而已。

在一个 AllowUsers 行中也可以出现多个字符串。SSH1 和 OpenSSH 使用空格分隔字符串；但是 SSH1/OpenSSH 和 SSH2 的语法并不相同：

```
# SSH1, OpenSSH  
AllowUsers smith jones cs*
```

SSH2 使用逗号分隔字符串，其中不允许使用空格：

```
# 仅对 SSH2  
AllowUsers smith,jones,cs*
```

AllowUsers 和 DenyUsers 可以有效地结合使用。假设你正在讲授一门课程，希望只有自己的学生才能使用 SSH 访问你的服务器。可能学生的用户名都是以“stu”开始的，因此可以这样设置：

```
# SSH1, SSH2, OpenSSH  
AllowUsers stu*
```

后来，有一个学生 stu563 不再选修这门课程了，因此你想禁止她的 SSH 访问权限。你可以简单地把配置修改成：

```
# SSH1, SSH2, OpenSSH  
AllowUsers stu*  
DenyUsers stu563
```

嗯，这似乎有些奇怪。这两行看起来是相互冲突的，因为第一行允许 stu563 访问，而第二行又拒绝该用户访问。服务器是这样处理这个问题的：如果任何一行禁止某

个账号访问，那么这个账号就不能访问。因此在前面这个例子中，stu563 就被第二行取消了访问权限。

考虑另外一个例子：

```
# SSH1, SSH2, OpenSSH
AllowUsers smith
DenyUsers s*
```

表面上它允许接收连往 smith 账号的连接，但是拒绝连往所有以“s”开头的用户的连接。服务器又如何处理这种明显的冲突呢？实际上服务器会拒绝连往 smith 的连接，所使用的也是相同的规则：如果任何一条限制禁止访问（例如上面的 DenyUsers 行），那么就不能访问。只有所有的条件都允许用户访问，该用户才可以访问。

sshd 可以为 AllowUsers 和 DenyUsers 最多保存 256 个用户字符串。这种静态限制并没有文档明确说明，但是在任何一个关键字（例如一个 AllowUsers 后面跟上 256 个字符串）或多个关键字（例如 16 个 AllowUsers 后面各跟上 16 个字符串）后面的字符串必须遵守这个限制。也就是说，这种限制是服务器内在的，和配置文件中每行的长度无关。

最后，我们再使用 SSH1 的语法给出一个有用的配置例子：

```
AllowUsers walrus@* carpenter@* *@*,beach.net
```

这样可以限制大部分账号只能以 *beach.net* 域连接，但“walrus”和“carpenter”除外，这两个账号从什么地方都可以访问。walrus 和 carpenter 后面加上 @* 并没有什么必要，但是这有助于更清楚地说明这一行内容的意义。

有一点需要注意：这些访问控制语句中的主机名都要依赖于 DNS 的完整性，这很容易就可以欺骗。如果这是整个问题的关键，就考虑使用 IP 地址，但是维护工作就变得更麻烦了。

5.5.2.2 组访问控制

sshd 可以允许或禁止对服务器机器上一个 Unix 组中的所有账号进行访问。关键字 AllowGroups 和 DenyGroups 就是用于这个目的；它们后面可以跟上一个或多个 Unix 组名：

```
# SSH1, OpenSSH (由空格分隔)
AllowGroups faculty
DenyGroups students secretaries

# 仅对 SSH2 (由逗号分隔)
AllowGroups faculty
DenyGroups students,secretaries
```

这两个关键字的操作类似于 AllowUsers 和 DenyUsers。SSH1 和 OpenSSH 可以在组名中使用通配符 * 和 ?，而 SSH2 使用自己通用的规则表达式（请参看附录一），用户在每一行中都可以给出多个字符串：

```
# SSH1, OpenSSH
AllowGroups ?faculty s*s

# 仅对 SSH2
AllowGroups ?faculty,s*s
```

不幸的是，这些限制都只适用于目标账号的主组（primary group），主组是账号的 *passwd* 记录中所给出的那个组。一个账号同时还可能属于其他组（例如，使用 */etc/groups* 文件或 NIS 映射），但是 SSH 并不会注意到这一点。这十分遗憾：如果可以支持辅助组（supplementary group），那么可以很容易地定义一个 SSH 的子集——可以定义一个组来包括一些账号，比如说 *sshusers*——并使用 AllowGroups *sshusers* 来配置 SSH 服务器。这种特性还可以自动禁止对诸如 *bin*、*news*、*uucp* 等不需要 SSH 的系统账号的访问权限。以后可能有些 SSH 实现会修改这种功能。

缺省情况下，系统允许对所有的组进行访问。如果出现了任何的 AllowGroups 关键字，那么就只允许指定的主组进行访问（当然这还可以使用 DenyGroups 进一步限制）。

就像 AllowUsers 和 DenyUsers 的情况一样，AllowGroups 和 DenyGroups 的冲突也是使用限制权最高的方法来解决的。如果 AllowGroups 和 DenyGroups 行都禁止对一个给定的组进行访问，那么即使另外有一行说明允许这个组访问，最终结果也是不允许对这个组进行访问。和前面一样，配置文件中的 AllowGroups 和 DenyGroups 关键字之后可以出现的字符串也有不超过 256 个的限制。

5.5.2.3 主机名访问控制

在 AllowUsers 和 DenyUsers 的讨论中，我们已经介绍了如何允许或禁止来自一个给定主机（比如说 *example.com*）的 SSH-1 连接：

```
# SSH1, OpenSSH
AllowUsers *@example.com
DenyUsers *@example.com
```

SSH1 和 SSH2 都提供了关键字 AllowHosts 和 DenyHosts 更方便地限制主机的访问，这样就不用再使用账号名通配符了（译注 1）：

```
# SSH1, SSH2
AllowHosts example.com
DenyHosts example.com
```

AllowHosts 和 DenyHosts 关键字分别允许或禁止来自给定主机的 SSH 连接（注 12）。就像 AllowUsers 和 DenyUsers 一样：

- 该值可以包含通配符? 和 * (SSH1、OpenSSH) 或规则表达式 (SSH2, 附录一)。
- 该值可以包含多个字符串，中间使用空格分开 (SSH1、OpenSSH)，或使用逗号分开 (SSH2)。
- 关键字可以在配置文件中出现多次，结果可以累积。
- 主机名和 IP 地址都可以使用。
- 配置文件中 AllowHosts 或 DenyHosts 关键字后面最多都只能跟上 256 个字符串。

AllowHosts 和 DenyHosts 在各种访问控制关键字之中有一个独有的特性。如果 sshd1 根据 AllowHosts 或 DenyHosts 拒绝一个连接，那么它可以给客户端打印一条信息：

```
Sorry, you are not allowed to connect.
```

这个打印操作是由 SilentDeny 关键字控制的。如果该值是 no (缺省值)，就打印这个消息；如果该值是 yes，就不打印消息 (也就是说什么都不打印)：

译注 1：指在 AllowUsers 和 DenyUsers 中使用 hostname 来对主机进行限制。

注 12：使用 *authorized_keys* 的 “from” 选项，我们可以提供更细粒度的控制。每个公钥都可以放在可能会使用该密钥连接到的主机定义中。

```
# 仅对 SSH1  
SilentDeny no
```

SilentDeny还有一种作用，它可以防止失效的连接请求出现在服务器的日志消息中。在关闭 SilentDeny 之后，就可以在日志中看到：

```
log: Connection from client.marceau.net not allowed.  
fatal: Local: Sorry, you are not allowed to connect.
```

在开启 SilentDeny 时，这些消息就不会出现在服务器日志中。SilentDeny 不会影响其他访问控制关键字（DenyUsers、DenySHosts 等），和认证也没有什么关系。

5.5.2.4 shosts 访问控制

不管使用哪种认证类型，AllowHosts 和 DenyHosts 都可以用于所有基于主机名的访问控制。可信主机认证有一种专用的方法与此相似，但访问控制的限制较少。用户可以禁止访问 *.rhosts*、*.shosts*、*/etc/host.equiv* 和 */etc/shosts.equiv* 文件中的主机。这是使用 AllowSHosts 和 DenySHosts 关键字来实现的（注 13）。

例如，下面这一行

```
# SSH1, SSH2  
DenySHosts badguy.com
```

就禁止来自 *badguy.com* 的连接访问，但是只有在使用可信主机认证时才管用。同理，在使用可信主机认证时，AllowSHosts 只允许访问给定的主机。它们的语法与 AllowSHosts 和 DenyHosts 的语法相同。结果是系统管理员可以覆盖用户的 *.rhosts* 和 *.shosts* 文件中设置的值（这一点很好，因为它不能使用 */etc/hosts.equiv* 或 */etc/shosts.equiv* 实现同样的功能）。

就像 AllowSHosts 和 DenyHosts 一样：

- 该值可以包含通配符? 和 * (SSH1) 或规则表达式 (SSH2, 附录一)。
- 该值可以包含多个字符串，中间使用空格分开 (SSH1)，或使用逗号分开 (SSH2)。

注 13： 虽然这个关键字的名字是“SHosts”，但实际上它也可以应用于 *.rhosts* 和 */etc/equiv* 文件。

- 关键字可以在配置文件中出现多次，结果可以累积。
- 主机名和IP地址都可以使用。
- 配置文件中AllowSHosts或DenySHosts关键字后面最多都只能跟上256个字符串。

5.5.2.5 root 的访问控制

sshd 对超级用户专门使用一种特殊访问机制。关键字 PermitRootLogin 可以允许或禁止使用 SSH 访问 root 账号：

```
# SSH1, SSH2, OpenSSH  
PermitRootLogin no
```

该关键字的值可以是 yes (缺省值)，表示允许使用 SSH 访问 root 账号；也可以是 no，表示禁止这种访问；还可以是 nopwd (SSH1、SSH2) 或 without-password (OpenSSH)，表示允许使用除密码认证之外的方法进行访问。

在 SSH1 和 OpenSSH 中，PermitRootLogin 只适用于登录，而不能用于 *authorized_keys* 所指定的强制命令。^[8.2.4] 例如，如果 root 的 *authorized_keys* 文件包含了一个以下面的内容开始的行：

```
command="/bin/dump" ....
```

那么就可以使用 SSH 访问 root 账号，以便运行 *dump* 命令，不管 PermitRootLogin 的值是多少，结果都是如此。这种能力让远程客户端可以运行超级用户的进程，例如备份或文件检测，但是不能无限地进行登录。

服务器在认证完成之后要检查 PermitRootLogin。换而言之，如果 PermitRootLogin 是 no，那么客户端就有机会进行认证（例如，被提示输入密码或口令），但是随后就会关闭。

我们前面已经看到过一个类似的关键字 IgnoreRootRhosts，它使用可信主机认证对 root 账号的访问权限进行限制。它禁止 *~root/.rhosts* 和 *~root/.shosts* 中的项用于认证 root 账号。由于 *sshd* 是在认证完成之后才对 PermitRootLogin 进行检查的，因此它会覆盖 IgnoreRootRhosts 的设置。表 5-2 说明了这两个关键字之间的相互影响。

表 5-2: root 可以登录吗?

	IgnoreRootRhosts yes	IgnoreRootRhosts no
PermitRootLogin yes	可以, 但是可信主机认证不行	可以
PermitRootLogin no	不行	不行
PermitRootLogin nopwd (nopassword)	可以, 但是可信主机和密码认证 不行	可以, 但是密码认证不行

5.5.2.6 使用 chroot 限制目录的访问权限

Unix 系统调用 chroot 会让一个进程把一个给定的目录当成 root 目录。使用 cd 命令切换到这个给定的目录子树之外的尝试都会失败。这对于由于安全原因而把一个用户或进程限制到一个文件系统子集的情况非常有用。

SSH2 提供了两个关键字来利用这种方法对到达的 SSH 客户端进行限制。ChRootUsers 规定 SSH 客户端在访问一个给定的账号时, 权限只能限制于该账号的主目录及其子目录:

```
# 仅对 SSH2
ChRootUsers smith
```

用户可以在一行中指定几个账号, 中间使用逗号隔开, 意思是说在使用 SSH2 访问时, 每个这种账号都单独进行限制:

```
# 仅对 SSH2
ChRootUsers smith,jones,mcnally
```

另外一个关键字 ChRootGroups 的工作原理类似, 但是它对一个给定的 Unix 组中的所有用户进行权限限制:

```
# 仅对 SSH2
ChRootGroups users,wheel,mygroup
```

警告: ChRootGroups 只会检查一个账号的主组; 而不会考虑辅助组。这样就大大降低了它应有的用途。希望将来能实现它的全部功能。

要充分利用这种 chroot 的功能，可能需要把一些系统文件拷贝到这些账号中。否则登录就会失败，因为它不能访问必需的资源，例如共享库。在我们的 Linux 系统中，我们需要拷贝以下程序和库文件到这些账号中：

```
/bin/ls
/bin/bash
/lib/ld-linux.so.2
/lib/libc.so.6
/lib/libtermcap.so.2
```

这种工作可以通过静态链接到 SSH 的可执行文件上实现。SSH2 最近新加入了一个名为 *ssh-chrootmgr* 的程序来帮助实现这个工作；不幸的是，这个程序的发行时间距本书（英文版）的出版时间实在太短，所以我们还没有时间仔细研究它。请参看手册页了解更多细节。

5.5.2.7 认证和访问控制小结

SSH 提供了几种方法用来允许或限制连往特定的账号或来自特定主机的连接。表 5-3 和表 5-4 对这些可用的选项进行了小结。

表 5-3：SSH1 和 OpenSSH 的认证和访问控制小结

如果你是 ...	你想允许或限制 ...	那么就使用 ...
用户	使用公钥认证连往你的账号的连接	<i>authorized_keys</i> [8.2.1]
系统管理员	连到一个账号的连接	<i>AllowUsers</i> , <i>DenyUsers</i>
用户	来自一个主机的连接	<i>authorized_keys from=“...”</i> 选项 [8.2.5.1]
系统管理员	来自一个主机的连接	<i>AllowHosts</i> , <i>DenyHosts</i> (或 <i>AllowUsers</i> , <i>DenyUsers</i>)
用户	使用可信主机认证连往你的账号的连接	<i>.rhosts</i> , <i>.shosts</i>

表 5-3: SSH1 和 OpenSSH 的认证和访问控制小结 (续)

如果你是 ...	你想允许或限制 ...	那么就使用 ...
系统管理员	可信主机认证	RhostsAuthentication, RhostsRSAAuthentication, IgnoreRhosts, AllowSHosts, DenySHosts, /etc/hosts.equiv, /etc/shosts.equiv
系统管理员	root 登录	IgnoreRootRhosts, PermitRootLogin

表 5-4: SSH2 的认证和访问控制小结

如果你是 ...	你想允许或限制 ...	那么就使用 ...
用户	使用公钥认证连往你的账号的连接	<i>authorization file [8.2.2]</i>
系统管理员	连到一个账号的连接	AllowUsers, DenyUsers
用户	来自一个主机的连接	N/A
系统管理员	来自一个主机的连接	AllowHosts, DenyHosts
用户	使用可信主机认证连往你的账号的连接	.rhosts, .shosts
系统管理员	可信主机认证	AllowedAuthentications, AllowSHosts, DenySHosts, /etc/hosts.equiv, /etc/shosts.equiv
系统管理员	root 登录	PermitRootLogin

5.5.3 选择登录程序

另外一种控制对机器进行认证和访问的方法是替换 Unix 登录程序。SSH1 为这样处理而提供了一个钩子程序(hook)，但是这需要对操作系统的登录过程有相当的了解才可以使用。

当一个 SSH1 客户端和服务器发起一个终端会话时，通常服务器都会直接调用本地账号的登录 shell。用户可以在编译时配置中指定 `--with-login[4.1.5.9]` 选项来覆

盖这个设置，从而让服务器调用指定的登录程序（例如，*/bin/login* 或 Kerberos 的 *login.krb5*）（注 14）。

这有什么区别呢？和服务器机器上的操作系统有关。登录程序可以设置其他一些环境变量（例如，在 X Window 系统中设置 DISPLAY 变量），执行其他审查或登录，或者执行 shell 所没有的其他操作。

为了 *sshd* 能调用由 *--with-login* 指定的登录程序，还必须设置 *UseLogin* 关键字，这个关键字也没有专门的文档说明。该值可以是 *yes*（使用其他登录程序）或 *no*（缺省值）：

```
# SSH1, OpenSSH
UseLogin yes
```

OpenSSH 没有 *--with-login* 选项，因此不能指定其他的登录程序。OpenSSH 的 *UseLogin* 语句只能选择 */bin/login* 或登录 shell。

登录程序的行为和登录 shell 相比，它完全是实现专用的，因此我们就不再详细介绍了。如果用户需要使用 *UseLogin*，必须首先详细了解操作系统和登录程序的特性。

5.6 用户登录和账号

在登录过程发生时，SSH 服务器可以执行特殊的操作。我们在这部分中会讨论以下内容：

- 为用户打印欢迎信息。
- 处理账号和密码过期的问题。
- 处理空密码。
- 使用 */etc/sshrc* 执行任意操作。

注 14：如果调用的是 */bin/login*，那么你就会奇怪为什么它没有提示每个 SSH 客户端都输入登录密码呢？原因是服务器运行的命令是 */bin/login -f*，该命令禁止使用登录密码认证。在很多操作系统的 *login* 手册页中都没有给出 *-f* 选项的说明。

5.6.1 用户的欢迎信息

当用户登录时，*sshd*会打印“今日消息”文件（*/etc/motd*）中的消息，以及用户是否有email。这种输出可以在配置文件中开启或关闭。由于大部分 Unix 的 shell 都会在登录时打印这条信息，因此这种 SSH 特性通常都是多余的，都会被关闭。

要启用或禁用这种今日消息，你可以通过把 **PrintMotd** 关键字设置成 yes（缺省值）或 no 来实现：

```
# SSH1, SSH2, OpenSSH
PrintMotd no
```

顺便说一下，*sshd*遵守 Unix 的“*hushlogin*”约定。如果系统中存在 *~/.hushlogin* 文件，那么就不会在登录时打印 */etc/motd* 的内容，不管 **PrintMotd** 值如何设置都是如此。

如果我们把 **CheckMail** 关键字设置成 yes（缺省值），那么就会在登录时打印有关 email 的消息（例如，“You have mail”）；如果该值为 no，就不会打印这种信息：

```
# SSH1, SSH2, OpenSSH
CheckMail yes
```

5.6.2 账号或密码过期

如果一个用户的密码或计算机账号很快就要过期了，*sshd*可以在该用户使用 SSH 登录时打印一条警告信息：

```
WARNING: Your password expires in 7 days
WARNING: Your account expires in 10 days
```

这些消息可以使用关键字 **PasswordExpire-WarningDays** 和 **AccountExpire-WarningDays** 分别加以启用或禁用：

```
# 仅对 SSH1
PasswordExpireWarningDays 7
AccountExpireWarningDays 10
```

关键字后面跟的数字是天数，缺省情况下二者都是 14。该值为 0 表示禁止打印这些消息。注意账号和密码的过期都不是 SSH 的问题，而是主机操作系统的问题（注 15）。

如果一个密码已经过期了，那么 SSH1 服务器就会提示用户在登录之前修改密码。这种特性是由关键字 `ForcedPasswdChange` 控制的，该值可以为 `yes` 或 `no`（缺省值）。如果我们启用了这种特性：

```
# 仅对 SSH1  
ForcedPasswdChange yes
```

而密码又过期了，那么系统就会提示用户修改密码。只有在用户修改密码之后系统才可以接受 SSH 连接。

5.6.3 空密码

如果使用密码认证，并且有个账号没有设定密码，那么 SSH 服务器就可以拒绝访问这个账号。这种特性是由 `PermitEmptyPasswords` 关键字控制的，该值可以为 `yes`（缺省值）或 `no`。如果启用了这种特性：

```
# SSH1, SSH2, OpenSSH  
PermitEmptyPasswords yes
```

就允许使用空密码，否则不行。

SSH1 还要求密码为空的用户修改自己的密码。`ForcedEmptyPasswdChange` 关键字负责控制这种特性，其方式类似于 `ForcedPasswdChange` 对过期密码的控制。`ForcedEmptyPasswdChange` 关键字的值可以为 `yes` 或 `no`（缺省值）：

```
# 仅对 SSH1  
ForcedEmptyPasswdChange yes
```

如果该值是 `yes`，而且密码为空，那么用户在登录时，就会被提示修改自己的密码，只有在修改密码之后才可以登录。

注 15：账号过期要求你的操作系统支持 `/etc/shadow`。密码过期需要 `struct passwd` 结构有一个 FreeBSD 中一样的 `pw_expire` 字段。

没有文档说明)。通常用户都可以通过在客户端命令行中给出命令来调用远程命令。例如,下面这行就调用 Unix 的备份程序 *tar*, 远程把 */home* 目录拷贝到磁带上:

```
# SSH2, OpenSSH/2
$ ssh server.example.com /bin/tar c /home
```

子系统是子服务器机器上预先定义的一组远程命令,这样就可以方便地执行(注16)。这些命令是在服务器的配置文件中定义的,其语法对于OpenSSH和SSH2来说稍有不同。调用前面的备份命令的子系统是:

```
# SSH2
subsystem-backups      /bin/tar c /home

# OpenSSH/2
subsystem backups       /bin/tar c /home
```

注意SSH2使用关键字的格式为“*subsystem-name*”后面跟上一个参数,而OpenSSH使用关键字“*subsystem*”,后面跟上两个参数。SSH2的这个语法很奇怪,和它的配置文件中的其他内容相比显得有些不伦不类;我们不知道如何替代这种方法。

要在服务器上运行这个命令,可以使用 *ssh* 加上 *-s* 选项:

```
# SSH2, OpenSSH/2
$ ssh -s backups server.example.com
```

这个命令和前面显式调用 */bin/tar* 的结果完全相同。

缺省情况下, *sshd2_config* 文件中定义了一个子系统:

```
subsystem-sftp      sftp-server
```

警告: 不要把 *subsystem-sftp* 这一行从 *sshd2_config* 文件中删除:这是 *scp2* 和 *sftp* 工作所必须的。实际上,这两个程序都会运行 *ssh2 -s sftp* 来执行文件传输操作。

子系统是一种十分方便的特性,它预定义了一些命令,可以让SSH客户端方便地调用。系统管理员还可以使用其他几个有用的抽象层,使用它们可以为用户定义并宣

注16: 理论上,子系统不必是一个单独的程序;它可以通过函数名调用SSH服务器中的一个函数。但是目前还没有这种产品。

传使用有用的子系统。假设你的用户使用 SSH2 来加密连接，并运行 Pine email 阅读程序连接到你的 IMAP 服务器上。[11.3] 你不需要通知每个用户都使用命令：

```
$ ssh2 server.example.com /usr/sbin/imapd
```

并向其说明 IMAP 守护进程 *imapd* 的路径，可以定义一个子系统来隐藏这个路径，这样即使以后路径发生了变化，用户也感觉不出来：

```
# 仅对 SSH2
subsystem-imap          /usr/sbin/imapd
```

现在用户可以运行下面的命令：

```
$ ssh2 -s imap server.example.com
```

使用子系统建立安全的 IMAP 连接。

5.7.1 禁用 Shell 启动文件

如果用户的远程 shell 是 C shell 或 *tcsh*，那么在会话开始时通常都要读取远程 shell 启动文件 (*.cshrc*、*.tcshrc*)。启动文件中有些命令，尤其是那些会往标准输出上写入内容的命令，可能会和文件拷贝命令 *scp2* 和 *sftp* 有干扰。在 SSH2 中，文件拷贝是由 *sftp-server* 子系统来完成的，因此 SSH2 就为子系统而禁止读取 *.cshrc* 和 *.tcshrc* 文件。[3.5.2.4] 用户可以使用关键字 *AllowCshrcSourcingWithSubsystems* 重新启用这种特性，该值可以为 yes (允许读取 *.cshrc* 和 *.tcshrc* 文件) 或 no (缺省值)：

```
# 仅对 SSH2
AllowCshrcSourcingWithSubsystems yes
```

SSH2 通过向远程 C shell 或 *tcsh* 调用传递 *-f* 命令行选项来禁止读取 *.cshrc* 和 *.tcshrc* 文件。

5.8 历史记录、日志记录和调试

当 SSH 服务器运行时，它可以记录一些日志消息来说明正在执行什么操作。日志消息可以帮助系统管理员跟踪服务器的行为，检测并诊断问题所在。例如，如果一个

服务器莫名其妙地拒绝了一个本来应该接受的连接，要判断原因，第一个要检查的就是服务器的输出日志。

SSH1、SSH2 和 OpenSSH 的日志都不相同，因此我们会分别加以讨论。

5.8.1 日志和 SSH1

缺省情况下，*sshd1* 会把日志记入 *syslog*，它是 Unix 标准的日志工具（请参看说明栏“Syslog 日志服务”）。例如，服务器启动时会产生这些 *syslog* 项：

```
log: Server listening on port 22.  
log: Generating 768 bit RSA key.  
log: RSA key generation complete.
```

客户端的连接和断连如下：

```
log: Connection from 128.11.22.33 port 1022  
log: Rhosts with RSA host authentication accepted for smith, smith on myhost.net  
log: Closing connection to 128.11.22.33
```

sshd1 允许使用三种方法控制日志：

Fascist Logging 模式

把调试消息打印到系统日志文件中，可以使用 *FascistLogging* 关键字启用。

调试模式

Fascist Logging 模式的一种扩展，可以使用 *-d* 命令行选项启用。

安静模式

只能输出严重错误，而不能输出普通的日志，可以使用 *QuietMode* 关键字或 *-q* 命令行选项启用。

5.8.1.1 SSH1 *Fascist Logging* 模式

Fascist Logging 模式可以让 *sshd1* 把产生的所有调试信息都打印到系统日志文件中。例如：

```
debug: Client protocol version 1.5; client software version 1.2.26  
debug: Sent 768 bit public key and 1024 bit host key.  
debug: Encryption type: idea  
debug: Received session key; encryption turned on.
```

Syslog 日志服务

Syslog 是标准的 Unix 日志服务。程序把自己的日志消息发送给 syslog 守护进程 *syslogd*，后者负责把这些消息转发给其他目的地，例如，终端或文件。目的地是在 syslog 控制文件 */etc/syslog.conf* 中指定的。

syslogd 所接收到的消息是根据这种机制进行处理的，它指明了消息的来源。标准的 syslog 机制包括 KERN (消息来自于操作系统内核)、DAEMON (消息来自于系统守护进程)、USER (消息来自于用户进程)、MAIL (消息来自于 email 系统) 以及其他一些地方。缺省情况下，这种机制对于 SSH 服务器来说是 DAEMON。我们可以使用 *SyslogFacility* 关键字来修改这种设置，它决定了 SSH 消息日志使用的简便的 syslog 代码：

```
# SSH1, SSH2, OpenSSH
SyslogFacility USER
```

其他可以使用的值还有 USER、AUTH、LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、LOCAL5、LOCAL6 和 LOCAL7。请参看 *syslog*、*syslogd* 和 *syslog.conf* 的手册页来了解有关这种日志服务的更多内容。

Fascist Logging 模式在服务器配置文件中是使用 *FascistLogging* 关键字来控制的，该值可以为 yes 或 no (缺省值) (注 17)：

```
# SSH1 (及 SSH2)
FascistLogging yes
```

5.8.1.2 SSH1 调试模式

调试模式和 Fascist Logging 模式一样，它也可以让服务器打印调试信息。缺省情况下这是禁止的，用户可以使用 *sshd* 的 *-d* 命令行选项来启用：

```
# SSH1, OpenSSH
$ sshd -d
```

调试模式打印的诊断信息和 Fascist Logging 模式完全相同，但是还要把这些消息显

注 17：但是只有 SSH2 才可以支持。[5.8.2.5]

示到标准错误设备上。例如，使用调试模式在 TCP 端口 9999 上运行的服务器会产生如下的诊断输出：

```
# SSH1, OpenSSH
$ sshd -d -p 9999
debug: sshd version 1.2.26 [sparc-sun-solaris2.5.1]
debug: Initializing random number generator; seed file /etc/
ssh_random_seed
log: Server listening on port 9999.
log: Generating 768 bit RSA key.
Generating p: .....++ (distance 100)
Generating q: .....++ (distance 122)
Computing the keys...
Testing the keys...
Key generation complete.
log: RSA key generation complete.
```

然后，服务器就在前台等待连接。当一个连接到达时，服务器就打印：

```
debug: Server will not fork when running in debugging mode.
log: Connection from 128.11.22.33 port 1022
debug: Client protocol version 1.5; client software version 1.2.26
debug: Sent 768 bit public key and 1024 bit host key.
debug: Encryption type: idea
debug: Received session key; encryption turned on.
debug: Installing crc compensation attack detector.
debug: Attempting authentication for smith.
debug: Trying rhosts with RSA host authentication for smith
debug: Rhosts RSA authentication: canonical host myhost.net
log: Rhosts with RSA host authentication accepted for smith, smith on
myhost.net
debug: Allocating pty.
debug: Forking shell.
debug: Entering interactive session.
```

当客户端退出时，服务器也退出，因为（正如上面的消息所说明的一样）服务器在调试模式下运行时不会派生子进程，而是使用一个进程来处理一个连接：

```
debug: Received SIGCHLD.
debug: End of interactive session; stdin 13, stdout (read 1244, sent 1244), stderr
0 bytes.
debug: pty_cleanup_proc called
debug: Command exited with status 0.
debug: Received exit confirmation.
log: Closing connection to 128.11.22.33
```

调试模式除具有 Fascist Logging 模式的特点之外，还具有如下特性：

- 同时向标准错误显示日志消息。
- 向标准输出设备打印的消息有些不会写入日志文件，例如，RSA密钥创建的消息。
- 确保服务器是单线程的，禁止服务器派生子进程。（因此才有前面的“Server will not fork when running in debugging mode”。）服务器在处理完一个连接请求之后就会退出。这在诊断错误时非常有用，这样用户就可以把注意力集中到一个客户端的连接上。
- 把 LoginGraceTime 设置为 0，这样在用户调试问题时就不会中断连接。（这一点很有意义。）
- 让 Unix SSH 客户端在连接之前向标准输出上打印服务器范围的环境变量设置。这有助于调试连接的问题。

例如，一个连接使用 9999 端口连接到服务器上，它可以这样显示之前产生的诊断输出：

```
$ ssh -p 9999 myserver.net
[...login output begins...]
Environment:
HOME=/home.smith
USER=smith
LOGNAME=smith
PATH=/bin:/usr/bin:/usr/ucb
MAIL=/var/mail.smith
SHELL=/usr/bin/ksh
TZ=US/Eastern
HZ=100
SSH_CLIENT=128.11.22.33 1022 9999
SSH_TTY=/dev/pts/3
TERM=vt220
REMOTEUSER=smith
[... 登录输出继续 ...]
```

由于这种方便的特性，调试模式通常都比 Fascist Logging 模式更有用。

5.8.1.3 SSH1 安静模式

安静模式禁止从 *sshd1* 中输出诊断消息，这取决于 Fascist Logging 模式和调试模式的设置。表 5-5 说明了安静模式和其他两种模式各种组合所对应的操作。

表 5-5: SSH1 安静模式的操作

安静模式	调试模式	Fascist Logging 模式	结果
No	No	No	缺省日志 (syslog), 无“调试”消息
No	No	Yes	Fascist Logging 模式 (syslog)
No	Yes	Yes/No	调试模式 (syslog, stderr)
Yes	No	No	只记录严重错误日志 (syslog)
Yes	No	Yes	只记录严重错误日志 (syslog)
Yes	Yes	Yes/No	只记录严重错误日志 (syslog, stderr) 和密钥创建消息

安静模式是由服务器配置文件中的 `QuietMode` 关键字控制的，该值可以为 `yes` 或 `no` (缺省值):

```
# SSH1, SSH2
QuietMode yes
```

也可以使用 `-q` 命令行选项控制:

```
# SSH1, SSH2, OpenSSH
$ sshd -q
```

5.8.2 日志和 SSH2

SSH2 的日志模式和 SSH1 不同。虽然二者的关键字和命令行选项外表几乎都一样，但是其行为并不相同:

调试模式

在标准错误设备上打印调试消息。可以使用 `-d` 命令行选项并在后面跟上一个整数 (调试级别) 或一个模块定义 (用于更细粒度的调试) 来启用。

详细模式

第二级调试模式的一个别名。可以使用 `-v` 命令行选项或 `VerboseMode` 关键字来启用。

Fascist Logging 模式

没有文档详细说明，几乎没什么用处。可以使用 `FascistLogging` 关键字启用。

安静模式

只允许记录严重错误，不能记录其他日志。可以使用 QuietMode 关键字或 *-q* 命令行选项来启用。

警告：我们强烈推荐使用 *--enable-debug-heavy* 标记来编译 SSH2，这样可以启用重 (heavy) 调试级别。[4.1.5.14] 这样日志所记录的消息就比缺省打印的内容更详细。

5.8.2.1 SSH2 调试模式（通用模式）

SSH2 的调试模式只能使用命令行选项来启用，而不能使用关键字。和 SSH1 一样，调试模式也是使用 *-d* 命令行选项来控制的。但是又和 SSH1 不同，该选项需要一个参数来指定调试级别，输出可以发送到标准输出 (stderr)。

调试级别可以使用两种方法来指定。第一种方法是使用一个非负整数：

```
# 仅对 SSH2
$ sshd2 -d 1
```

在本书印刷前，系统可以支持的整数级别如例 5-1 所示。把调试级别指定为 *n* 就是说所有小于等于 *n* 的级别的消息都要打印。例如，调试级别 9 表示要打印 0 到 9 级的调试消息。

例 5-1：SSH2 的调试级别

不能在内部循环中使用：

- 0) 软件故障
- 1)
- 2) (0-2 也可以使用 log-event 启用)
- 3) 外部非严重高级错误
 - 从外部接收到的数据的格式不对
 - 协商失败
- 4) 高级信息
 - 协商成功
- 5) 高级或中级操作的开始
 - 协商开始
 - 打开设备
 - 不能用于在内部循环中调用的函数

可以在内部循环中使用：

- 6) 可能是由 bug 所引起的非正常情况

- 7) 容易理解的信息
 - 进入或退出一个函数
 - 低级操作的结果
- 8) 数据块
 - 散列
 - 密钥
 - 证书
 - 其他小数据块
- 9) 协议报文
 - TCP
 - UDP
 - ESP
 - AH
- 10) 中间结果
 - 内部循环
 - 非最终结果
- 11-15) 用子程序自己调试使用
 - 自行判断
 - 只需要一个人来确定

5.8.2.2 SSH2 调试模式（基于模块的模式）

调试级别也可以为每个 SSH2 的源代码“模块”分别进行设置。这允许对日志进行更细粒度的控制，结果是产生大量输出。这种调试只在源代码中 (*lib/sshutil/sshcore/sshdebug.h*) 有说明，因此要有效地使用这种模式，还必须要具有 C 的编程知识。

为了调试方便，SSH2 的源文件被定义成“模块”，这是通过在文件中定义 `SSH_DEBUG_MODULE` 来实现的。例如，文件 *apps/ssh/auths-passwd.c* 中就包含了模块 `Ssh2AuthPasswdServer`，因为它包含了这样一行：

```
#define SSH_DEBUG_MODULE "Ssh2AuthPasswdServer"
```

SSH2 2.3.0 中所有的模块名如表 5-6 所示。

表 5-6: SSH2 模块名

<code>ArcFour</code>	<code>GetOptCompat</code>	<code>Main</code>
<code>Scp2</code>	<code>Sftp2</code>	<code>SftpCwd</code>
<code>SftpPager</code>	<code>Ssh1KeyDecode</code>	<code>Ssh2</code>
<code>Ssh2AuthClient</code>	<code>Ssh2AuthCommonServer</code>	<code>Ssh2AuthHostBasedClient</code>
<code>Ssh2AuthHostBasedRhosts</code>	<code>Ssh2AuthHostBasedServer</code>	<code>Ssh2AuthKerberosClient</code>
<code>Ssh2AuthKerberosServer</code>	<code>Ssh2AuthKerberosTgtClient</code>	<code>Ssh2AuthKerberosTgtServer</code>

表 5-6: SSH2 模块名 (续)

Ssh2AuthPasswdClient	Ssh2AuthPasswdServer	Ssh2AuthPubKeyClient
Ssh2AuthPubKeyServer	Ssh2AuthServer	Ssh2ChannelAgent
Ssh2ChannelSession	Ssh2ChannelSsh1Agent	Ssh2ChannelTcpFwd
Ssh2ChannelX11	Ssh2Client	Ssh2Common
Ssh2PgpPublic	Ssh2PgpSecret	Ssh2PgpUtil
Ssh2Trans	Ssh2Transport	SshADT
SshADTArray	SshADTAssoc	SshADTList
SshADTMap	SshADTTest	SshAdd
SshAgent	SshAgentClient	SshAgentPath
SshAppCommon	SshAskPass	SshAuthMethodClient
SshAuthMethodServer	SshBufZIP	SshBuffer
SshBufferAux	SshConfig	SshConnection
SshDSprintf	SshDebug	SshDecay
SshDirectory	SshEPrintf	SshEncode
SshEventLoop	SshFCGlob	SshFCRecurse
SshFCTransfer	SshFSM	SshFastalloc
SshFileBuffer	SshFileCopy	SshFileCopyConn
SshFileXferClient	SshFilterStream	SshGenCiph
SshGenMP	SshGetCwd	SshGlob
SshInet	SshKeyGen	SshPacketImplementation
SshPacketWrapper	SshPgpCipher	SshPgpFile
SshPgpGen	SshPgpKey	SshPgpKeyDB
SshPgpPacket	SshPgpStringToKey	SshProbe
SshProtoSshCrDown	SshProtoSshCrup	SshProtoTrKex
SshReadLine	SshReadPass	SshRegex
SshSprintf	SshServer	SshServerProbe
SshSftpServer	SshSigner2	SshStdIOFilter
SshStream	SshStreamPair	SshStreamstub
SshTUserAuth	SshTime	SshTimeMeasure
SshTimeMeasureTest	SshTtyFlags	SshUdp
SshUdpGeneric	SshUnixConfig	SshUnixPtyStream

表 5-6: SSH2 模块名 (续)

SshUnixTcp	SshUnixUser	SshUnixUserFiles
SshUserFileBuffer	SshUserFiles	Sshd2
TestMod	TestSshFileCopy	TestSshGlob
TestTtyFlags	t-fsm	

要从源代码中得出所有的模块名，可以在 SSH2 发行版本的根目录中的源文件中查找 `SSH_DEBUG_MODULE`:

```
$ find . -type f -exec grep SSH_DEBUG_MODULE '{}' \;
```

得到想要的模块名之后，用户就可以给出模块名和调试级别，从而以调试模式运行服务器了：

```
$ sshd2 -d "module_name=debug_level_integer"
```

这可以让给定的模块使用给定的调试级别打印日志消息。例如：

```
$ sshd2 -d "Ssh2AuthPasswdServer=2"
```

就让 `Ssh2AuthPasswdServer` 模块使用调试级别 2 来记录日志。这些消息提供了它们碰到的函数名和函数代码所在的文件名。

用户可以指定多个模块，使用逗号彼此进行分隔，并为每个模块都设置一个调试级别：

```
$ sshd2 -d "Ssh2AuthPasswdServer=2,SshAdd=3,SshSftp=1"
```

另外，在指定多个模块名时，可以使用通配符 * 和 ?:

```
$ sshd2 -d 'Ssh2*=3'
```

记住要把一个模式使用一对单引号封装起来，这样就可以防止 Unix shell 破坏它的完整性。

注意这只是因为源代码文件有相关的调试模块名，但是它不表示可以实际记录的任何消息日志。你可能会发现为特定的模块启用调试并不会产生任何更多的调试输出。

5.8.2.3 调试 sshd2 -i

如果 SSH2 是通过 *inetd* 调用的，那么对其进行调试就需要一些技巧了。如果用户没有采用任何其他方法，那么调试输出就随着普通的协议信息一起发往客户端了，这样将把连接弄得乱七八糟，最终导致失败。用户需要做的工作是把 *sshd* 的标准错误重定向到一个文件中。理想情况下，可以在 */etc/inetd.conf* 文件中实现这个功能：

```
ssh stream tcp nowait root /bin/sh /bin/sh -c "/usr/sbin/sshd2 -i -d2 2> /tmp/foo"
```

但是，很多 *inetd* 都不允许在程序参数中嵌入空格（也就是说，它不能识别上面这个例子引号中的内容）。可以使用一个单独的脚本来解决这个问题，例如：

```
/etc/inetd.conf
  ssh stream tcp nowait root /path/to/debug-sshd2-i debug-sshd2-i

debug-sshd2-i
  #!/bin/sh
  # 重定向 sshd2 标准错误到一文件
  exec /usr/local/sbin/sshd2 -i -d2 2> /tmp/sshd2.debug
```

5.8.2.4 SSH2 的详细模式

详细模式和第二级调试模式完全相同。它可以使用 *sshd2* 的 *-v* 命令行选项启用：

```
# 仅对 SSH2
$ sshd2 -v                                使用 -v
$ sshd2 -d 2                                与上一行相同
```

也可以在服务器配置文件中使用的 *VerboseMode* 关键字进行控制，该值可以为 yes 或 no (缺省值)：

```
# 仅对 SSH2
VerboseMode yes
```

5.8.2.5 SSH2 的 Fascist Logging 模式

Fascist Logging 模式在 SSH2 中没有文档进行说明。其惟一目的似乎是用来覆盖安静模式。[5.8.2.6] 该值可以是 yes 和 no (缺省值)：

```
# SSH1, SSH2
FascistLogging yes
```

5.8.2.6 SSH2 安静模式

在安静模式中，只有严重错误才会记录在日志中。这可以使用没有文档说明的 Fascist Logging 模式来覆盖。和 SSH1 中一样，安静模式是由服务器范围的配置文件中的 QuietMode 关键字控制的，该值可以是 yes 或 no (缺省值)：

```
# SSH1, SSH2  
QuietMode yes
```

也可以使用 *sshd* 的 *-q* 命令行选项来启用：

```
# SSH1, SSH2, OpenSSH  
$ sshd -q
```

5.8.3 日志和 OpenSSH

OpenSSH 中的日志是通过 *syslog* 来处理的，由两个配置关键字来控制：*SyslogFacility* 和 *LogLevel*。*SyslogFacility* 决定了在向 *syslog* 服务发送消息时使用的“facility”代码；根据 *syslog* 的配置，这有助于确定如何处理日志消息（写入终端、存储到文件中等等）。*LogLevel* 决定了所记录的日志提供的详细程度。该值按照详细程度递增的顺序依次是：

```
QUIET, FATAL, ERROR, INFO, VERBOSE, DEBUG
```

使用 *DEBUG* 级别记录日志会侵犯用户的隐私权，这个级别只能用于诊断问题，而不能用于普通操作。

如果 *sshd* 是以调试模式 (*-d*) 运行的，那么日志消息就发往标准错误设备，而不会发往 *syslog*。安静模式 (*LogLevel Quiet* 或 *sshd -q*) 不会向系统日志发送任何内容（但是 OpenSSH 操作的一些结果仍然会记录到系统日志中，例如从 PAM 发来的一些消息）。

5.8.3.1 不用 RSA 支持

如果 OpenSSH 限制使用协议 2，那么在编译 OpenSSH 时就不用包含 RSA 的支持，但是如果缺少这种支持，*sshd* 就会打印一条错误消息。要禁止打印这种错误消息，可以使用 *-Q* 选项：

```
# 仅对OpenSSH  
$ sshd -Q
```

5.9 SSH-1 和 SSH-2 服务器的兼容性

OpenSSH版本2可以同时支持SSH-1和SSH-2协议，它可以在一个守护进程中同时接受两种连接。但是对于SSH1和SSH2来说，这个问题就变得复杂了。

SSH2服务器可以接受来自SSH1客户端的连接。这种兼容性是这样实现的：在有SSH-1连接请求时，让SSH2服务器运行SSH1服务器程序。这种兼容特性可以使用SSH2的Ssh1Compatibility关键字来启用或禁用，该值可以为yes或no：

```
# 仅对SSH2  
Ssh1Compatibility yes
```

当启用了Ssh1Compatibility并有SSH-1客户端连接到SSH2服务器上时，这两个程序会交换字符串来指明版本号。^[3.4.1]然后，sshd2就通过检查Sshd1Path关键字的值来定位sshd1的可执行程序：

```
# 仅对SSH2  
Sshd1Path /usr/local/bin/sshd1
```

然后sshd2调用sshd1进程，并使用-V命令行选项向sshd1传递客户端的版本号字符串（注18）：

```
# 仅对SSH2，自动由sshd2调用  
/usr/local/bin/sshd1 -V "client version string" <other arguments>
```

-V命令行选项只能在sshd2内部使用。这一点是必须的，因为当sshd1这样启动时，客户端已经发出了自己的原始版本声明，而sshd1需要取得这个信息。我们想不到有什么实际原因要手工使用这个选项，但是为了保持完整性起见，我们还是在这里对其进行介绍。

当用户编译并安装SSH2时，如果系统中已经安装了SSH1，那么配置脚本^[4.1.4]就会自动设置一些编译内置的一些缺省值：把Ssh1Compatibility设置成yes，把

注18：注意，至少要使用SSH1的1.2.26版本(F-Secure 1.3.6)才可以使用兼容模式，因为这个选项在更早的版本中都没有实现。

`Sshd1Path` 设置成 `sshd1` 的路径。如果系统中还没有安装 SSH1，那么编译缺省值 `Ssh1Compatibility` 就是 no，`Sshd1Path` 就是空字符串。

OpenSSH 服务器也实现了 `-V` 选项，这样用户就可以在 SSH2 的向后兼容模式中使用 OpenSSH 来代替 SSH1。

警告：`sshd2` 可以接受 SSH1 客户端连接并对其重新路由，反之则不然：`sshd1` 不能接受 SSH2 的连接。

5.9.1 SSH2 中使用 SSH-1 兼容模式的安全性问题

如果你正在 SSH2 中使用 SSH-1 的兼容特性，那么千万要记住一点：你必须维护两份单独的 SSH 服务器配置。当 `sshd2` 开始调用 `sshd1` 时，它完全是个新进程，拥有自己的 SSH1 服务器配置文件。你的 SSH2 服务器配置中的任何限制都不能适用于这个进程。即使本来应该可以使用的一些限制（例如，`AllowHosts`）也不行，因为 `sshd2` 是在执行这种检测之前调用 `sshd1` 的。

这就是说，为了保证安全性，必须保持这两个配置文件同步。否则，攻击者就可以绕开你仔细配置过的 SSH2，而直接简单地使用一个 SSH-1 客户端连接上来。

5.10 小结

正如我们看到的一样，SSH 服务器有很多的配置选项，在某些情况下，有很多方法是异曲同工。然而，这种强大的功能是需要一定的代价的。当设置一个安全系统时，必须要仔细考虑所有的选项，并为其选择适当的值。千万不要在这方面偷工减料、节省脑力：你的系统的安全性可能依靠于它。第十章给出了 SSH1、SSH2 和 OpenSSH 的配置。另外，本章中的所有关键字和选项都在附录二中给出。

记住，服务器范围的配置只是影响服务器行为的一种途径而已。我们在第四章讨论的编译时配置和第八章中讨论的每账号配置也都可以影响服务器行为。

本章内容

- 什么是身份标识
- 创建一个身份标识
- SSH代理
- 使用多个身份标识
- 小结

第六章

密钥管理与代理

私钥是一种非常有价值的东西。当用户使用公钥认证时，私钥就会向SSH服务器证明用户的身份。我们已经看到过几个与密钥有关的程序：

ssh-keygen

这个程序可生成密钥对。

ssh-agent

它把私钥保存在内存中，就不用再反复输入口令了。

ssh-add

它把私钥加载到代理中的程序。

不过我们才仅仅涉及了几个最常用的基本密钥操作，还未谈到十分深入的内容。现在将开始仔细研究这些概念和程序的细节。

我们首先概括介绍一下SSH身份标识和用于表示这些身份标识的密钥。然后彻底剖析SSH代理及其众多的功能。最后向读者展示使用多个SSH身份标识的好处。读者如果曾经用过一点儿密钥和代理的功能，就会发现这儿有很多非常棒的东西。图6-1对密钥管理在整个配置过程中的位置进行了小结。

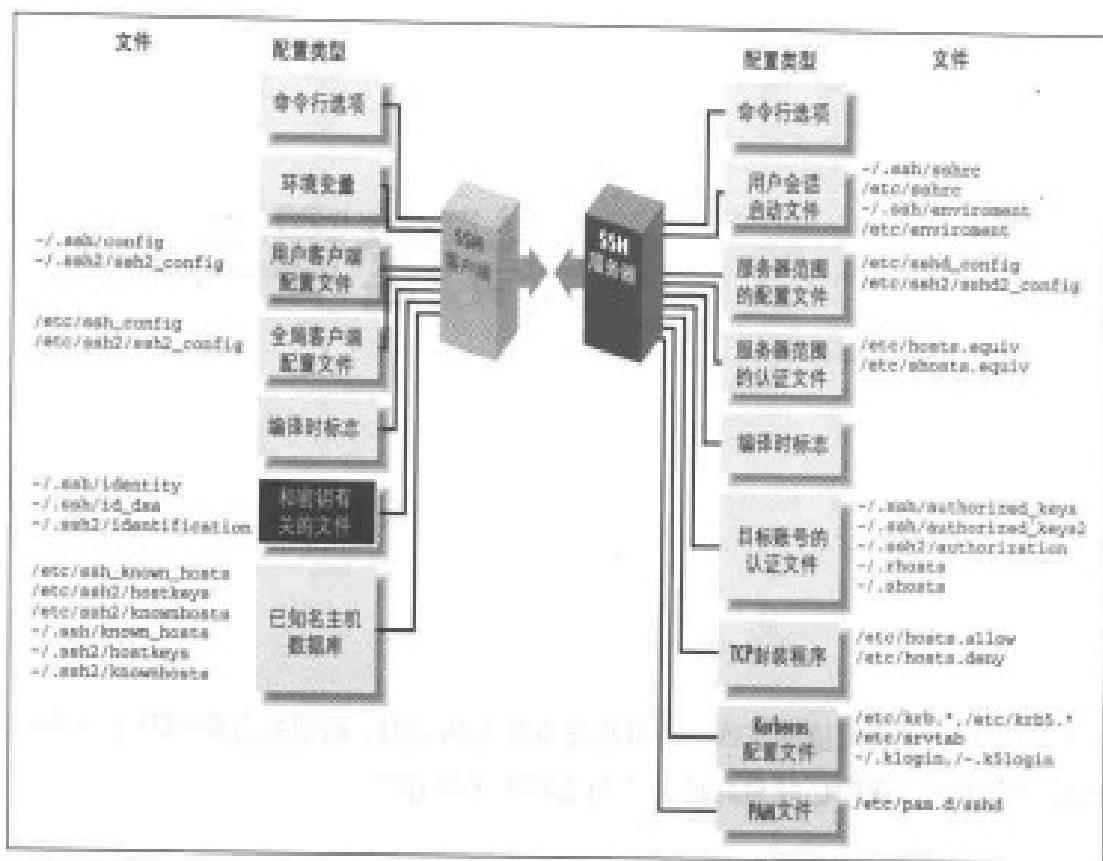


图 6-1: SSH 用户密钥和代理配置 (高亮显示部分)

本章是高级 SSH 功能系列中的第一章，专门为终端用户（而非系统管理员）编写。当学习完本章介绍的密钥管理知识以后，我们还会指引读者继续学习客户端配置、服务器配置以及转发的知识。

6.1 什么是身份标识

SSH 身份标识是一个位串，试图说明“我真的是我”。它是一个数学结构，SSH 客户端凭借它向 SSH 服务器证实自己的身份，接下来 SSH 会跟它说：“啊，我知道了，你确实是 you。你已经通过认证了，进来吧。”

一个身份标识由两部分组成，分别称为私钥（Private Key）和公钥（Public Key）。这两部分合称一个密钥对。

私钥负责在外发出的 SSH 连接中证实用户的身份。当用户在自己的账号下运行一

个 SSH 客户端（如，*ssh* 或 *scp*）时，程序向 SSH 服务器请求建立连接，然后客户端就使用这个私钥向服务器证实用户的身份。

警告。私钥应该保存在安全的地方。入侵者要是得到了你的私钥，就可以像你一样轻松地访问你的账号了。

公钥在到达你账号的连接中标识你的身份。SSH 客户端请求访问你的账号时，会用一个私钥作为其身份的标识，SSH 服务器将检查相应的公钥。如果（根据某种加密测试[3.4.2.2]）两个密钥是“匹配”的，那么认证就通过了，将继续连接。公钥不必保密；攻击者不能使用公钥攻破一个账号。

典型的密钥对保存在一对名字相关的文件中（注 1）。SSH 中公钥文件的名字是私钥文件名后面加一个后缀 *.pub*。例如，如果 *mykey* 中保存着私钥，那么对应的公钥就在 *mykey.pub* 里（注 2）。

只要你愿意，就可以有多个 SSH 身份标识。多数 SSH-1 及 SSH-2 的实现产品会让你指定一个缺省标识，在不特别说明的情况下，客户端程序就用缺省值。要使用其他身份标识，必须使用命令行参数、配置文件或其他配置工具修改设置。

SSH1、SSH2 和 OpenSSH 身份标识文件的格式各不相同，因此我们分别加以说明。它们在文件系统中的位置如图 6-2（私钥）和图 6-3（公钥）所示。

6.1.1 SSH1 身份标识

SSH1 身份标识存储在两个文件里。SSH1 缺省设置中，私钥存储在文件 *identity* 中，公钥存储在文件 *identity.pub* 中。这个密钥对存放在 *~/.ssh* 目录下，如不另外说明，它就是客户端使用的缺省身份标识。

注 1：有些 Windows 版的产品，如 F-Secure SSH Client 与此大不相同，它们把密钥存储在 Windows 注册表里。

注 2：实际上出于完整性的考虑，SSH1 中所谓的私钥文件里面也保存了公钥。只有保存私钥的那一部分才是用口令加密的。不过私钥文件使用一种专用的二进制格式表示；而为了人们使用方便，公钥文件可以用文本编辑器打开，还可以很容易地把公钥增加到一个 *authorized_keys* 文件中。

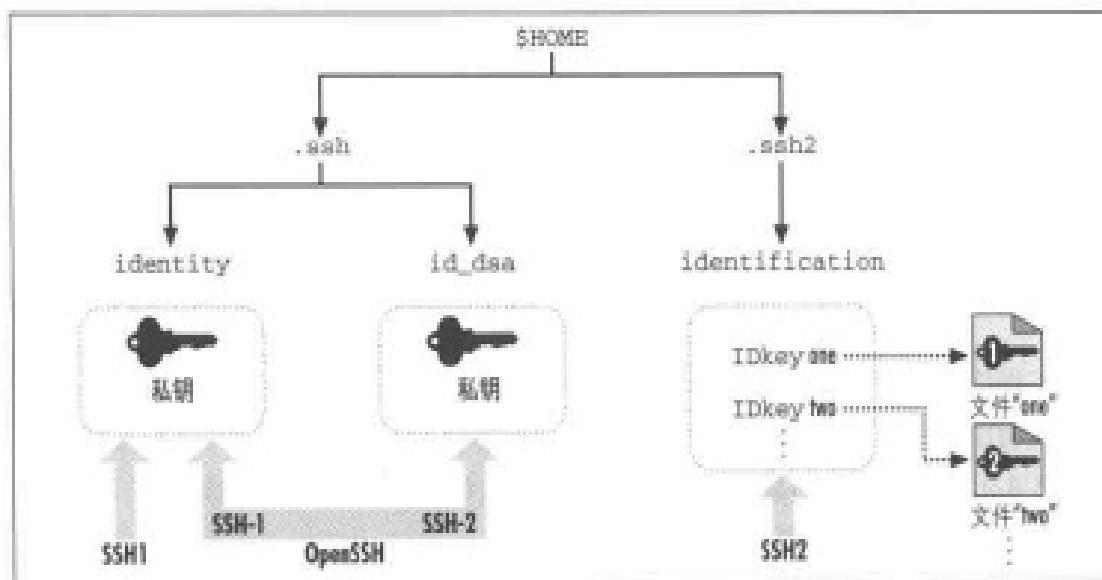


图 6-2: SSH 身份标识文件 (私钥) 及使用它们的程序

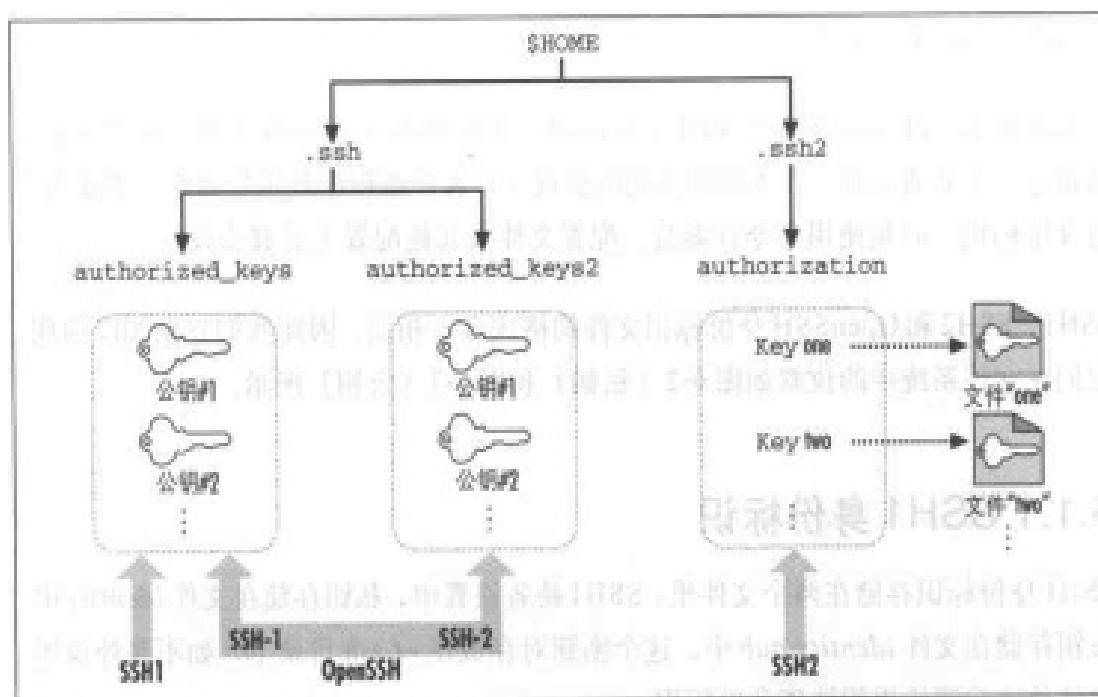


图 6-3: SSH 身份标识文件 (公钥) 及使用它们的程序

.pub 文件包含的公钥本身不具备任何功能。在其用于认证之前，首先要把这个公钥拷贝到 SSH-1 服务器上的一个认证文件里，比如 SSH1 和 OpenSSH 中的 `~/.ssh/authorized_keys`。此后，当 SSH-1 客户端请求连接你在服务器上的账号时，就会用一个私钥作为证明身份的标识。SSH1 服务器则在 `authorized_keys` 文件中寻找与之匹配的公钥。[\[3.4.2.2\]](#)

6.1.2 SSH2 身份标识

SSH2 密钥对与其 SSH1 的前身一样，分别保存在名字相关的两个文件中（即，私钥文件名后面加上.pub 就是公钥的文件名）。SSH2 密钥文件的命名通常都是根据该密钥使用的加密算法的性质。例如，一个用 DSA 加密的 1024 位密钥生成时其缺省文件名是 *id_dsa_1024_a* 和 *id_dsa_1024_a.pub*。

SSH2 与 SSH1 的不同之处在于，一个 SSH2 身份标识不是单独一个密钥，而是一组密钥。SSH2 客户端请求认证时，可能用到这一组中所有的密钥。如果用第一个密钥认证失败，SSH2 客户端自动试用第二个，依此类推，直至认证成功或全部失败为止。

用户必须把私钥放在一个名为 *identification* 的文件中，才能在 SSH2 中创建一个身份标识。缺省的身份标识存储在 *~/.ssh2/identification* 中（注 3）。一个私钥在这个文件里占用一行。在公钥认证中，每一行的开头都有一个关键字 *IdKey*，后面跟着私钥文件名。例如：

```
# SSH2 identification 文件
# 下列文件名均相对于路径 ~/.ssh2
IdKey id_dsa_1024_a
IdKey my-other-ssh2-key
# 绝对路径在 SSH2 2.1.0 后有效
IdKey /usr/local/etc/third-key
```

读者可能还记得，SSH2 可以支持 PGP 密钥认证。[5.5.1.6] *identification* 文件中也可以包含 PGP 相关的关键字：

```
# SSH2 identification 文件
PgpSecretKeyFile my-file.pgp
IdPgpKeyName my-key-name
```

使用单独一个 *identification* 文件可能显得很麻烦，不过这也使 SSH2 具有了一些 SSH1 无法提供的灵活性。我们说过，这样一个身份标识中就可以包含多个密钥，其中的任何一个都可以用来进行认证。SSH2 系统的另外一个好处是删除起来容易。要删除不用的 SSH2 私钥，只需要把它从 *identification* 文件中删掉或注释掉就可以了。而 SSH1 要达到同样的目的，就必须把私钥文件重命名。

注 3： 该缺省值可以用 *IdentityFile* 关键字来改变。[7.4.2]

SSH2与SSH1一样，也为到达的连接准备了一个认证文件，但是有一点区别。SSH2的认证文件中不包含公钥的实际拷贝，只是把公钥文件列出来，前面用关键字 Key 标识：

```
# SSH2 授权文件  
Key id_dsa_1024_a.pub  
Key something-else.pub
```

这样维护起来比SSH1使用的*authorized_keys*更容易，因为每一个公钥只存在一个拷贝。SSH1和OpenSSH与此形成鲜明对比，它们的*.pub*文件和*authorized_keys*是重复的。[8.2.2]

6.1.3 OpenSSH 身份标识

OpenSSH对SSH-1连接使用的标识和授权文件和SSH1完全相同。对于SSH-2连接，缺省的密钥则存储在`~/.ssh/id_dsa`（私钥）和`~/.ssh/id_dsa.pub`（公钥）中。OpenSSH的SSH-2授权文件为`~/.ssh/authorized_keys2`，格式与`~/.ssh/authorized_keys`相似。[8.2.1]

6.2 创建一个身份标识

多数SSH产品都包含一个创建密钥对的程序。下面我们讨论SSH1、SSH2及OpenSSH中的`ssh-keygen`。

6.2.1 生成SSH1的RSA密钥

SSH1及其派生软件都使用`ssh-keygen1`程序来生成密钥对。[2.4.2]这个程序也可能被称为`ssh-keygen`，这与SSH1的安装方式有关。现在让我们更深入地研究一下这个程序。附录二对`ssh-keygen`的选项进行了总结。

`ssh-keygen1`既可以生成新密钥，也可以修改已经存在的密钥。生成新密钥时，可以在命令行中指定下列内容：

- `-b`，密钥的位数。缺省值为1024位。

```
$ ssh-keygen1 -b 2048
```

- *-f*, 要生成的私钥文件的名字。这个名字是相对于当前目录的。回想一下, 公钥文件的命名方法是在私钥文件名后面加上`.pub`。如果忽略这个选项, 程序会要求输入文件名。

```
$ ssh-keygen -f mykey          创建mykey和mykey.pub  
$ ssh-keygen  
Enter file in which to save the key (/home/barrett/.ssh/identity): mykey
```

- *-N*, 破解密钥的口令。若忽略此项, 程序在生成密钥之后会要求输入口令。

```
$ ssh-keygen -N secretword  
$ ssh-keygen  
Enter passphrase: [nothing is echoed]  
Enter the same passphrase again: [nothing is echoed]
```

- *-C*, 给密钥关联一个文本格式的注释。若忽略此项, 密钥的注释即为“`username@host`”, 其中 `username` 是本机用户名, `host` 为本机的完整域名。

```
$ ssh-keygen -C "my favorite key"
```

如果同时指定了 *-f* (输出文件) 和 *-N* (口令), `ssh-keygen` 就不会要求用户输入任何内容。因此, 可用这些选项使密钥的生成过程自动化, 还能把输出重定向到 `/dev/null`。

```
$ ssh-keygen -f mykey -N secretword > /dev/null 2> /dev/null
```

用户可以在自动化操作中运用这项技术, 出于某种目的可一次生成一大批密钥。不过在安全性要求很高的机器上还是要小心使用, 因为同一台 Unix 主机上的其他用户只要用 `ps` 或类似的命令, 就能看到显示在命令行中的密码; 如果用脚本实现该技术, 显然就不应该把口令放在文件里。

除了生成密钥之外, `ssh-keygen` 还可以修改已有的密钥, 方法如下:

- *-p*, 修改已有密钥的口令。可分别用 *-f*、*-P*、*-N* 指定文件名、旧口令和新口令:

```
$ ssh-keygen -p -f mykey -P secretword -N newword
```

若忽略这些选项, 系统会有提示:

```
$ ssh-keygen -p  
Enter file key is in (/home/barrett/.ssh/identity): mykey  
Enter old passphrase: [nothing is echoed]  
Key has comment 'my favorite key'  
Enter new passphrase: [nothing is echoed]
```

Enter the same passphrase again:

注意，这只改变了口令，而没有改变密钥本身，只是用新的口令重新对密钥进行了加密。因此，相应的公钥也没有改变，也不必更换SSH服务器上已经拷贝好的公钥。

警告： 使用任何一个把口令显示在 shell 命令行中的选项都要小心，比如 *-N* 或 *-P*，它们可能会损害系统的安全。因为口令显示在了屏幕上，所以旁边的人能看到。在程序执行的过程中，系统的进程列表也能反映出来。由于口令是由命令行输入的，因此同一台主机上的其他用户只要运行 ps 命令就可以看到。此外，如果 shell 创建记录了输入命令的历史文件，那么口令也会插入到某个历史文件中，可被第三方读取。

同理，如果你觉得自己有理由不指定口令，只输入回车，那么还是要请你三思。这样做就相当于在主目录下用一个文件 *MY-PASSWORD.PLEASE-STEAL-ME* 存放密码。如果你不想被迫一遍遍输入口令，那么正确的选择是 *ssh-agent*、可信主机认证或 Kerberos。只有少数情况下，由于要运行无人参与的任务（如，*cron* 任务），才允许客户端密钥的口令为明文，或不使用口令。[\[11.1\]](#)

- *-c*，改变已有密钥的注释。用 *-f*、*-P*、*-C* 分别指定文件名、口令和新的注释，不然系统也会提示输入这些内容。

```
$ ssh-keygen -c -f mykey -P secretword -C "my second-favorite key"
$ ssh-keygen -c
Enter file key is in (/home/barrett/.ssh/identity): mykey
Enter passphrase: [nothing is echoed]
Key now has comment 'my favorite key'
Enter new comment: my second-favorite key
The comment in your key file has been changed.
```

- *-u*，升级一个旧的 SSH1 密钥，以使其适应当前版本的 SSH1。旧版 SSH1 用口令加密密钥时使用了 IDEA 算法，但是现在的 SSH1 都使用 3DES，因此这些旧的密钥就不能用了。*-u* 选项通知 *ssh-keygen1* 将一个密钥解密，再用 SSH1 的缺省加密算法（假设为 3DES）重新加密，这样该密钥就能够再在当前版本的 SSH1 中使用。

```
$ ssh-keygen1 -u -f mykey -P secretword
$ ssh-keygen1 -u
Enter file key is in (/home/barrett/.ssh/identity): mykey
Enter passphrase: [nothing is echoed]
Key's cipher has been updated.
```

当更改一个密钥的相关信息（如口令或注释）时，实际改变的只是密钥文件。如果 SSH 代理已经加载了某个密钥，那么代理中的密钥不会改变。如果用户在修改注释

之后用 `ssh-add -l` (小写的字母 L) 列出代理中的所有密钥，就会看到代理中该密钥的注释还是旧的。要使更改在代理中生效，必须把密钥卸载掉，然后重新加载。

6.2.2 生成 SSH2 的 RSA/DSA 密钥

SSH2 及其派生软件用 `ssh-keygen2` 创建密钥对。这个程序也可能名为 `ssh-keygen`，这与 SSH2 的安装方式有关。它与 `ssh-keygen1` 相同的地方是既可以创建新密钥，也可以修改已有密钥；不过命令行参数却有很大差别。`ssh-keygen` 还包含了另外一些用于打印诊断信息的选项。

创建新密钥时，可以在命令行的末尾指明要生成的私钥文件的名字：

```
$ ssh-keygen2 mykey          创建 mykey 和 mykey.pub
```

这个文件名是相对于用户的当前路径的，按照常规，公钥文件的名字是私钥文件名后加`.pub`。如果不指定文件名，那么生成的文件将保存于`~/.ssh2`下，名字是与加密算法和密钥位数有关的一个字符串。如 `id_dsa_1024_a`，表示由 DSA 算法生成，共 1024 位。

命令行中还可以指定以下选项：

- `-b`，密钥位数。缺省值 1024。

```
$ ssh-keygen2 -b 2048
```

- `-t`，生成密钥的算法，如 DSA 或 RSA。SSH2 中缺省值（也是唯一的值）是 DSA（记为“`dsa`”）（注 4）：

```
$ ssh-keygen2 -t dsa
```

- `-c`，给密钥关联一个文本格式的注释。

```
$ ssh-keygen2 -c "my favorite SSH2 key"
```

若忽略这个选项，则自动生成一个密钥注释，其中说明了密钥的创建者和创建方式。例如：

```
"1024-bit dsa, barrett@server.example.com, Tue Feb 22 2000 02:03:36"
```

注 4： F-Secure SSH2 服务器可以支持 RSA（参数为“`rsa`”），但是支持有限。[3.9]

- **-p**, 给密钥加密的口令。若忽略该选项，则密钥生成之后会要求输入口令。

```
$ ssh-keygen2 -p secretword
```

还可以用 **-P** 指派一个空口令。通常不应该这么做，不过在某些情况下还是合适的。[11.1.2.2]

```
$ ssh-keygen2 -P
```

除了创建密钥之外，*ssh-keygen2* 还能以下列方式对已有密钥进行操作：

- **-e**, 修改已有密钥的口令和注释。*ssh-keygen2* 通过一个交互式的过程让用户输入新的信息。这个过程的实现方式很原始，因为要回答至少 10 个问题才能改变口令和注释，因此人们很容易厌烦，不过确实能达到目的：

```
$ ssh-keygen2 -e mykey
Passphrase needed for key "my favorite SSH2 key"
Passphrase : [nothing is echoed]
Do you want to edit key "my favorite SSH2 key" (yes or no)? yes
Your key comment is "my favorite SSH2 key".
Do you want to edit it (yes or no)? yes
New key comment: this is tedious
Do you want to edit passphrase (yes or no)? yes
New passphrase : [nothing is echoed]
Again      : [nothing is echoed]
Do you want to continue editing key "this is tedious" (yes or no)? god no
(yes or no)? no
Do you want to save key "this is tedious" to file mykey (yes or no)? yes
```

与*ss-keygen1*类似，更改的只是密钥文件，不会波及已经加载到代理中的密钥。（因此如果用*ssh-add2 -l*列出所有密钥，就会看到原来的注释。）

- **-D**, 打印公钥。如果公钥丢失了，还能由私钥生成公钥，但这只有在你的私钥没有丢失的情况下才可以。

```
$ ssh-keygen2 -D mykeyfile
Passphrase : *****
Public key saved to mykeyfile.pub
```

- **-I** (数字 1, 不是小写的字母 L), 把 SSH-1 格式的密钥转换成 SSH-2 格式。当前版本尚未实现。

```
$ ssh-keygen2 -I ssh1key
```

使用*ssh-keygen2* 也可以控制输入、输出及诊断信息：

- **-F**, 打印给定密钥文件的指纹。请参看“密钥的指纹”。指纹可以根据公钥进行计算：

```
# 仅对SSH2
$ ssh-keygen2 -F stevekey.pub
Fingerprint for key:
xitot-larit-gumet-fyfim-sozev-vyned-cigeb-sariv-tekuk-badus-bexax
```

- **-V**, 打印程序的版本号：

```
$ ssh-keygen2 -V
ssh2: SSH Secure Shell 2.1.0 (noncommercial version)
```

- **-h**或**-?**, 打印一条帮助信息。多数 Unix shell 要求将问号转义，以防 shell 把问号解释成一个通配符。

```
$ ssh-keygen2 -h
$ ssh-keygen2 -\? 对问号进行转义
```

- **-q**, 取消进度指示。进度指示是 *ssh-keygen2* 运行过程中显示出来的一串 O 和句点，如：.oOo.oOo.oOo.oOo。

```
$ ssh-keygen2
Generating 1024-bit dsa key pair
.oOo.oOo.oOo.oOo
Key generated.
$ ssh-keygen2 -q
Generating 1024-bit dsa key pair
Key generated.
```

- **-i**, 显示已经存在的密钥的相关信息：

```
$ ssh-keygen2 -i mykey
```

这个选项目前尚未实现。

最后，*ssh-keygen2* 还有一个十分高级的选项 **-r**，它可以影响生成密钥时使用的随机数。这个选项让 *ssh-keygen2* 根据用户在标准输入设备输入的数据修改 *~/.ssh2/random_seed*。^[3.7] *SSH2* 手册页称之为“把数据搅拌到随机池中”。请注意，程序并不会主动要求输入数据，它只是静静地等待，好像挂起了一样。这时，用户可以任意输入数据，然后按 EOF（多数 shell 中为 Ctrl-D）结束。

```
$ ssh-keygen2 -r
I am stirring the random pool.
blah blah blah
^D
Stirred in 46 bytes.
```

6.2.3 生成 OpenSSH 的 RSA/DSA 密钥

OpenSSH 的 *ssh-keygen* 程序可以支持 SSH1 中相应程序的全部功能和选项。它还新增了为 SSH-2 连接生成 DSA 密钥的能力，以及其他一些选项。

- *-d*, 生成 DSA 密钥:

```
# 仅对 OpenSSH
$ ssh-keygen -d
```

- *-x*、*-X* 和 *-y*, 在 SSH2 与 OpenSSH 的密钥存储格式之间转换。下表详细说明了这一点:

选项	转换源	转换目标
<i>-x</i>	OpenSSH DSA 私钥文件	SSH2 公钥
<i>-X</i>	SSH2 公钥文件	OpenSSH DSA 公钥
<i>-y</i>	OpenSSH DSA 私钥文件	OpenSSH DSA 公钥

OpenSSH 私钥文件实际上包含了整个的公钥私钥对，所以 *-x* 和 *-y* 只从中取出公钥，然后将其输出为指定的格式。用户应该用 *-x* 把一个 OpenSSH 公钥加入到 SSH2 服务器主机的 *~/.ssh2/authorization* 文件中，用 *-X* 完成相反的任务。如果不小心删掉了 OpenSSH 公钥文件，可用 *-y* 恢复。

此处还缺少一项转换私钥的功能。这项功能在下面这种情况下可能很有用：已经有了一台 OpenSSH 服务器主机，还想在上面运行 SSH2，而又希望这两个 SSH 服务器能共享一个主机密钥。

- *-l*, 打印给定密钥文件的指纹。参看“密钥的指纹”。指纹可以从公钥中计算出来:

```
# 仅对 OpenSSH
$ ssh-keygen -l -f stevekey.pub
1024 5c:f6:e2:15:39:14:1a:8b:4c:93:44:57:6b:c6:f4:17 steve@sshbook.com
```

- *-R*, 监测 OpenSSH 是否支持 RSA 密钥。2000 年 9 月之前 RSA 是受专利保护的技术，因此某个特定的 OpenSSH 可能支持，也可能不支持这种算法。[\[3.9.1.1\]](#) 如果用 *-R* 选项调用 *ssh-keygen*，那么如果系统支持 RSA，就立刻返回一个 0，否则返回 1。

密钥的指纹

指纹是鉴别位置不同的两个密钥是否相同的一种密码学技术，这项技术用于不可能逐一对比两个密钥的情况。OpenSSH 和 SSH2 能计算指纹。

假设 Steve 想用 SSH 访问 Judy 的账号。他用 email 把自己的公钥发给 Judy，然后 Judy 将这个公钥安装在自己的 SSH 授权文件里。这个过程看起来很简单，却很不安全：第三方会在中途截获 Steve 的密钥，用自己的密钥将其换掉，这样他就能访问 Judy 的账号了。

为了防止冒险，Judy 得使用某种方法鉴别她收到的密钥是不是 Steve 的。她可以给 Steve 打电话检查一下，但是在电话里读一个 500 字节的加密公钥是既烦人又容易出错的。这正是指纹技术存在的原因。

指纹，是根据密钥计算出来的一个长度较短的值。它的原理与校验和类似，用于验证一串信息（在我们的例子中是密钥）的不可替换性。Steve 和 Judy 可以按照下面的步骤，使用指纹技术检查密钥的合法性：

1. Judy 收到一个号称是 Steve 传来的密钥，将其保存在 *stevekey.pub* 中。
2. Judy 和 Steve 分别查看密钥的指纹：

```
# 仅对 OpenSSH
$ ssh-add -l stevekey.pub
1024 5c:f6:e2:15:39:14:1a:8b:4c:93:44:57:6b:c6:f4:17
Steve@sshbook.com
```

```
# 仅对 SSH2
$ ssh-keygen2 -F stevekey.pub
Fingerprint for key:
xitot-larit-gumet-fyfim-sozev-vyned-cigeb-sariv-tekuk-badus-bexax
```

3. Judy 让 Steve 在电话里读出密钥的指纹，自己检查一下收到的密钥的指纹是否与之相符。指纹不是唯一的，但对任意两个密钥来说，指纹相同的可能性非常小。因此，指纹是检查密钥是否被替换的迅速而方便的手段。

我们可以看到，OpenSSH 与 SSH2 使用了不同的指纹输出格式。OpenSSH 使用的是数字格式，这种方式更符合传统，用过 PGP 的人应该对此很熟悉。而

SSH2 运用了一种称为“气泡流 (Bubble Babble)”的文本格式，能满足易读和易记的要求。

当 SSH 服务器的主机密钥发生变化时，指纹技术也能发挥作用。这时，OpenSSH 打印出一条提示信息及新密钥的指纹，如果有旧的指纹，就可以方便地对二者进行比较了。

```
# 仅对OpenSSH, 有RSA支持  
$ ssh-keygen -R; echo $?  
0  
  
# 仅对OpenSSH, 无RSA支持  
$ ssh-keygen -R; echo $?  
1
```

6.2.4 选择适当的口令

请仔细选择口令。至少应该有 10 个字符长，最好包含大写和小写字母、数字及非文字的符号。同时，口令应该既容易记忆，又要别人难以猜测。在口令中不要使用姓名、电话号码或其他容易猜的信息。找出一个有效的口令会很麻烦，但是为了增强安全性，这是很值得的。

如果忘记了口令，那你的运气可是够差的：由于无法解密，相应的 SSH 私钥就不能用了。保护 SSH 安全的加密机制同样也保证了口令的不可破解性。只能丢弃现在的 SSH 密钥，再重新生成一个，然后为它选择一个新的口令。同时，必须在所有安装旧密钥的机器上安装新生成的公钥。

6.3 SSH 代理

SSH 代理是一个管理私钥缓冲区、并对 SSH 客户端发出的认证请求作出响应的程序。^[2.5] 它能处理所有与密钥有关的操作，使用户不必重复输入口令，因此极大地节省了你的劳动。

与代理有关的程序有 *ssh-agent* 和 *ssh-add*。*ssh-agent* 运行代理，*ssh-add* 向代理的密钥缓冲区中插入和移出密钥。典型的应用如下：

```
* 启动代理  
$ ssh-agent $SHELL  
* 装载用户的缺省身份  
$ ssh-add  
Need passphrase for /home/barrett/.ssh/identity (barrett@example.com).  
Enter passphrase: *****
```

用户只需输入一次口令，代理就可以把解密的私钥存储在内存中。从现在开始，直到终止运行代理或退出登录之前，SSH 客户端进行所有与密钥有关的操作时，都会自动与代理进行联系，再也不需要输入口令了。

我们首先讨论代理是如何工作的，然后举实例说明启动代理的两种方式——单 shell 方法和子 shell 方法，以及多项配置选项和几种自动向代理中加载密钥的技术。最后我们会介绍代理的安全性和代理转发，以及 SSH-1 与 SSH-2 代理的兼容性问题。

6.3.1 代理不会泄漏密钥

代理有两项任务：

1. 在内存中存储私钥。
2. 回答 SSH 客户端提出的和密钥有关的问题。

然而，代理并不会把私钥发送到别的地方去。理解这一点非常重要。私钥只要加载进来，就一直存在于代理中，其他 SSH 客户端看不到这一点。客户端要访问密钥，就说，“你好，代理！我需要你的帮助。请为我执行这个与密钥有关的操作。”代理就根据这个客户端的请求进行操作，并把结果发给客户端（请参看图 6-4）。

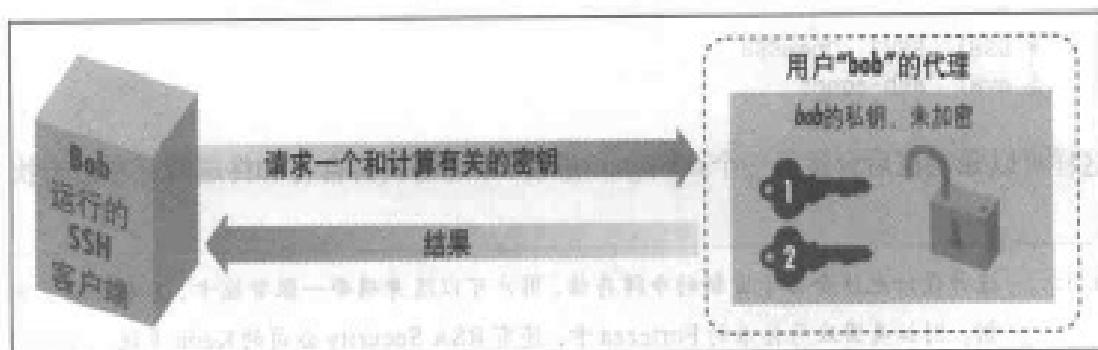


图 6-4：SSH 代理如何与它的客户端合作

例如，如果 *ssh* 要给某个认证者签名，就给代理发送一条签名请求，其中包含认证者的信息和应该使用哪个密钥的一条指示。代理自己执行加密操作，并返回签名。

SSH 客户端在这样使用代理的过程中甚至都没有看到代理中的私钥。这种技术比起把密钥传给客户端处理来说更加安全。存储或发送私钥的地方越少，就越不容易被窃（注 5）。

6.3.2 启动代理

在登录的账号中调用代理有两种方法：

1. 单 shell 方式，使用当前登录的 shell。
2. 子 shell 方式，派生出一个子 shell，并继承父 shell 的某些环境变量。

警告：不要想当然地使用下面的命令：

```
$ ssh-agent
```

尽管这样代理也可以运行，但是 SSH 客户端将无法与其联系，如果有些环境变量设置不正确，即使终止命令 (*ssh-agent -k*) 也无法将其除去。

6.3.2.1 单 shell 方式

单 shell 方式在你当前的登录 shell 中运行代理。如果用户正在一个终端上运行登录 shell，而不是 Unix 的 X Window 系统，那么这种方式就非常方便。输入：

```
# SSH1, SSH2, OpenSSH  
$ eval `ssh-agent`  
...  
$
```

这样可以强制在后台运行一个 *ssh-agent* 进程。该进程就会自行和终端断开在后台执

注 5：这种设计也适合基于密钥的令牌存储，用户可以随身携带一张智能卡，其中存储了密钥。例如美国政府标准的 Fortezza 卡，还有 RSA Security 公司的 Keon 系统。与代理类似，智能卡也会对和密钥有关的请求进行响应，而不会泄漏密钥，因此它们和 SSH 集成的方法非常简单。虽然现在使用令牌的速度非常慢，但是我们相信这个很快就会解决。

行，并返还 shell 提示符，这样用户就不必手工将其转入后台运行了（即在命令后面跟上一个 ‘&’）。注意，*ssh-agent* 外面的是 “`”，而不是 ‘`’（单引号）。

eval 命令的目的是什么呢？嗯，*ssh-agent* 运行时，不仅强制转入后台执行，而且会通过输出一些 shell 命令来设置使用代理时要用到的几个环境变量。这些变量是 SSH_AUTH_SOCK (SSH1 和 OpenSSH) 或 SSH2_AUTH_SOCK (SSH2)，还有 SSH_AGENT_PID (SSH1、OpenSSH) 或 SSH2_AGENT_PID (SSH2) (注 6)。*eval* 命令可以让当前的 shell 截获 *ssh-agent* 的输出结果，并进行解释，从而设置环境变量。如果忽略了 *eval*，那么这些命令就会在调用 *ssh-agent* 时打印到标准输出上。例如：

```
$ ssh-agent  
SSH_AUTH_SOCK=/tmp/ssh-barrett/ssh-22841-agent; export SSH_AUTH_SOCK;  
SSH_AGENT_PID=22842; export SSH_AGENT_PID;  
echo Agent pid 22842;
```

现在用户已经成功运行了一个代理，但是却不能访问 shell 了。可以使用前面显示的 PID 将其除掉：

```
$ kill 22842
```

也可以按照下面的内容设置环境变量，并手工连接到 shell 上：

```
$ SSH_AUTH_SOCK=/tmp/ssh-barrett/ssh-22841-agent; export SSH_AUTH_SOCK;  
$ SSH_AGENT_PID=22842; export SSH_AGENT_PID;
```

但是，使用单 shell 方法的命令更简单，反正一切都已经为你设置好了（注 7）。

要结束代理，可以除掉它的 PID：

注 6： 旧版本的 SSH1 使用 SSH_AUTHENTICATION_SOCKET，而不使用 SSH_AUTH_SOCK。如果用户是使用这种版本，我们建议自己设置 SSH_AUTH_SOCK，例如（以 C shell 为例）：

```
if ( "SSH_AUTHENTICATION_SOCKET" != "" ) then  
    setenv SSH_AUTH_SOCK $SSH_AUTHENTICATION_SOCKET  
endif
```

这样即使升级了 SSH 软件，账号也依然可用。

注 7： 为什么不让 *ssh-agent* 自己设置这些环境变量呢？这是因为 Unix 中一个程序不能设置其父 shell 的环境变量。

```
#SSH1, SSH2, OpenSSH
$ kill 22842
```

并取消环境变量的设置：

```
$ unset SSH_AUTH_SOCK          # SSH2 使用 SSH2_AUTH_SOCK
$ unset SSH_AGENT_PID          # SSH2 使用 SSH2_AGENT_PID
```

对于 SSH1 或 OpenSSH 来说，可以使用更方便的 `-k` 命令行选项：

```
# SSH1, OpenSSH
$ eval `ssh-agent -k`
```

该命令可以在标准输出上打印出结束命令，这样 `eval` 就可以调用它了。如果中断了 `eval`，也还可以去掉代理，但是环境变量就不能自动取消设置了。

```
# SSH1, OpenSSH
$ ssh-agent1 -k
unset SSH_AUTH_SOCK;           # This won't get unset,
unset SSH_AGENT_PID;          # and neither will this,
echo Agent pid 22848 killed;   # but the agent gets killed.
```

以单 shell 方式运行代理的方法和我们下面要介绍的方法（派生一个子 shell）相比有一个问题。当用户的登录会话结束时，`ssh-agent` 进程还不会死掉。在几次登录之后，会看到运行了很多代理，而这些代理都没什么用处（注 8）。

```
$ /usr/ucb/ps uax | grep ssh-agent
barrett  7833  0.4  0.4  828  608 pts/1      S 21:06:10  0:00 grep agent
barrett  4189  0.0  0.6 1460  844 ?          S  Feb 21  0:06 ssh-agent
barrett  6134  0.0  0.6 1448  828 ?          S 23:11:41  0:00 ssh-agent
barrett  6167  0.0  0.6 1448  828 ?          S 23:24:19  0:00 ssh-agent
barrett  7719  0.0  0.6 1456  840 ?          S 20:42:25  0:02 ssh-agent
```

用户可以在退出时运行 `ss-agent -k` 命令来自动解决这个问题。在 Bourne 风格的 shell (`sh`、`ksh`、`bash`) 中，这都可以在 `~/.profile` 的开头使用 Unix 的信号 0 设置一个陷阱来实现：

```
# ~/.profile
trap '
  test -n "$SSH_AGENT_PID" && eval `ssh-agent1 -k`;
  test -n "$SSH2_AGENT_PID" && kill $SSH2_AGENT_PID
' 0
```

注 8：实际上，你可以把 `SSH_AUTH_SOCK` 变量修改成指向原来的 socket，从而重新连接到之前登录时所启动的那个代理上。

对于 C shell 和 *tcsh* 来说，可以在 *~/.logout* 文件中结束代理。

```
# ~/.logout
if ( "$SSH_AGENT_PID" != "" ) then
    eval `ssh-agent -k`
endif
if ( "$SSH2_AGENT_PID" != "" ) then
    kill $SSH2_AGENT_PID
endif
```

一旦设置了这个陷阱，*ssh-agent*进程就会在你退出时被自动去掉，同时会打印这样一条消息：

Agent pid 8090 killed

6.3.2.2 子 shell 方法

调用代理的第二种方法是派生一个子 shell。用户可以为 *ssh-agent* 提供一个参数，该参数是一个 shell 或 shell 脚本的路径。例如：

```
$ ssh-agent /bin/sh  
$ ssh-agent /bin/csh  
$ ssh-agent $SHELL  
$ ssh-agent my-shell-script # 运行 shell 脚本而非 shell
```

这时 *ssg-agent* 不用强制在后台运行，而是可以在前台运行，派生一个子 shell，并自动设置上述环境变量。此后用户的登录会话都是在这个子 shell 中运行的。当结束登录会话时，*ssh-agent* 也就同时结束了。后面我们就会看到，如果使用 X window 系统并在初始化文件（例如，*~/.xsession*）里调用了代理，那么这种方法就非常方便。而且这种方法对于单终端的登录来说也是足够理想的。

在使用子 shell 方法时，可以在适当的时机调用代理。我们建议在登录初始化文件（例如，`~/.profile` 或 `~/.login`）的最后一行调用，也可以在登录之后马上输入一个命令调用。否则，如果先在 shell 中运行了一些后台进程，然后又调用了代理，那么最初那些后台进程只有在结束代理子 shell 之后才可以访问。例如，如果运行了 vi 编辑器，并将其挂起，然后又运行代理，那么只有在结束代理之后才能再访问 vi。

```
$ vi myfile          # 运行编辑器  
^Z                  # 将其挂起  
$ jobs              # 观看后台进程  
[1] + Stopped (SIGTSTP) vi
```

```

$ ssh-agent $SHELL          # 运行子 shell
$ jobs                      # 这里没有任务！它们在父 shell 中
$ exit                      # 终止代理的子 shell
$ jobs                      # 现在又能看见进程了
[1] + Stopped (SIGTSTP) vi

```

这两种方法的优缺点如表 6-1 所示。

表 6-1：调用代理优缺点

方法	优点	缺点
<code>eval `ssh-agent`</code>	简单，直观	代理必须手工结束
<code>ssh-agent \$SHELL</code>	代理的环境变量是自动传播的，在用户退出登录时代理可以自动结束。可以很好地用于 X Window	登录 shell 要依赖于代理的状态。如果代理死掉了，那么登录会话也就可能死掉

6.3.2.3 环境变量命令的格式

正如我们前面所说的一样，`ssh-agent` 会打印几个 shell 命令来设置环境变量。这些命令的语法和所使用的 shell 的语法可能有所不同。用户可以使用 `-s` 或 `-c` 选项分别强制命令使用 Bourne 风格或 C-shell 风格的语法。

```

# Bourne-shell 风格的命令
$ ssh-agent -s
SSH_AUTH_SOCK=/tmp/ssh-barrett/ssh-3654-agent; export SSH_AUTH_SOCK;
SSH_AGENT_PID=3655; export SSH_AGENT_PID;
echo Agent pid 3655;

# C-shell 风格的命令
$ ssh-agent -c
setenv SSH_AUTH_SOCK /tmp/ssh-barrett/ssh-3654-agent;
setenv SSH_AGENT_PID 3655;
echo Agent pid 3655;

```

通常 `ssh-agent` 会对用户的登录 shell 进行检测，并打印适当的行来设置环境变量，因此根本不用使用 `-c` 或 `-s` 选项。但是在一种情况中用户必须使用这两个选项：如果在一个 shell 脚本中调用 `ssh-agent`，然而所使用的脚本的 shell 和登录 shell 不同。例如，如果登录 shell 是 `/bin/csh`，并调用这个脚本：

```

#!/bin/sh
`ssh-agent`
```

ssh-agent 就会输出 C-shell 风格的命令，这样就会导致失败。因此应该使用：

```
#!/bin/sh  
`ssh-agent -s`
```

如果在 X 下运行代理，并且执行 *~/.xsession* 文件（或其他启动文件）使用的 shell 和登录 shell 不同，那么这就非常重要了。

6.3.2.4 SSH-1 和 SSH-2 代理的兼容性

SSH-1 的代理不能处理 SSH-2 客户端的请求；但是 SSH-2 的代理却可以处理 SSH-1 的服务请求。如果 *ssh-agent2* 是使用 *-l*（此处是数字 1，而不是小写的 L）选项调用的，那么这个代理就可以处理 SSH-1 客户端的请求，即使客户端使用了 *ssh-add1* 也可以。这只能用于支持 RSA 的 SSH-2 实现，因为 SSH-1 使用 RSA 密钥。在本书截稿时，只有 F-Secure SSH2 服务器可以兼容 SSH-1 代理。

```
# 在 SSH1 兼容模式调用 SSH2 代理  
$ eval `ssh-agent2 -l`  
  
# 添加 SSH1 密钥  
$ ssh-add1  
Need passphrase for /home/smith/.ssh/identity (smith SSH1 key).  
Enter passphrase: ****  
Identity added (smith SSH1 key).  
  
# 添加 SSH2 密钥  
$ ssh-add2  
Adding identity: /home/smith/.ssh2/id_dsa_1024_a.pub  
Need passphrase for /home/smith/.ssh2/id_dsa_1024_a  
(1024-bit dsa, smith SSH2 key, Thu Dec 02 1999 22:25:09-0500).  
Enter passphrase: *****  
  
# ssh-add1 仅列出 SSH1 密钥  
$ ssh-add1 -l  
1024 37 1425047358166328978851045774063877571270... and so forth  
  
# ssh-add2 列出两种密钥  
# 仅对 F-Secure SSH Server  
$ ssh-add2 -l  
Listing identities.  
The authorization agent has 2 keys:  
id_dsa_1024_a: 1024-bit dsa, smith SSH2 key, Thu Dec 02 1999 22:25:09-  
0500  
smith SSH1 key
```

现在 SSH-1 客户端就可以和 *ssh-agent2* 透明地进行联系了，认为它是一个 SSH-1 代理：

```
$ ssh1 server.example.com  
[no passphrase prompt appears]
```

ssh-agent2 可以通过设置和 *ssh-agent1* 相同的环境变量来实现兼容性：这些环境变量有 `SSH_AUTH_SOCK` 和 `SSH_AGENT_PID`。这样所有的对 SSH-1 代理的请求都被重定向到 *ssh-agent2* 上。

警告：如果正在运行一个 *ssh-agent1* 进程，又调用了 *ssh-agent2 -I*，那么那个老的 *ssh-agent1* 进程就不能访问了，因为 *ssh-agent2* 覆盖了它的环境变量。

只有当 SSH2 是使用 `--with-ssh-agent1-compat` 标志编译的时候，才能使用代理兼容性。[\[4.1.5.13\]](#) 它还和客户端配置关键字 `Ssh1AgentCompatibility` 的设置有关。[\[7.4.13\]](#)

6.3.3 使用 *ssh-add* 加载密钥

ssh-add 程序是用户和 *ssh-agent* 进程之间的一个私人通信通道。（这个命令在 SSH1 和 SSH2 中分别是 *ssh-add1* 和 *ssh-add2*，*ssh-add* 只是到这两个程序的一个链接而已。）

在首次调用一个 SSH 代理时，这个代理中还没有任何密钥。顾名思义，*ssh-add* 可以把私钥加入 SSH 代理中。但是这个名字容易让人产生一些误解，因为它还可以对代理执行其他操作，例如显示现有的密钥、删除密钥以及锁定代理不再接受新的密钥。

如果用户在调用 *ssh-add* 时没有指定参数，系统就会提示为缺省 SSH 密钥输入口令，然后将其加载到代理中。例如：

```
$ ssh-add1  
Need passphrase for /home.smith/.ssh/identity (smith@client).  
Enter passphrase: *****  
Identity added: /home.smith/.ssh/identity (smith@client).
```

```
$ ssh-add2  
Adding identity: /home.smith/.ssh2/id_dsa_1024_a.pub  
Need passphrase for /home.smith/.ssh2/id_dsa_1024_a  
(1024-bit dsa, smith@client, Thu Dec 02 1999 22:25:09-0500).  
Enter passphrase: *****
```

通常，*ssh-add*都会从用户终端中读取口令。但是如果标准输入不是终端，并设置了系统的DISPLAY环境变量，那么*ssh-add*就会调用一个X window图形化程序*ssh-askpass*，它会弹出一个窗口来读取口令。这在*xdm*的启动脚本中使用十分方便（注9）。

*ssh-add1*和*ssh-add2*都可以支持以下的命令行选项，可以用来显示密钥、删除密钥和读取口令：

- *-l*，显示代理中已经加载的所有密钥：

```
$ ssh-add1 -l  
1024 35 1604921766775161379181745950571099412502846... and so forth  
1024 37 1236194621955474376584658921922152150472844... and so forth  
  
$ ssh-add2 -l  
Listing identities.  
The authorization agent has one key:  
id_dsa_1024_a: 1024-bit dsa, smith@client, Thu Dec 02 1999 22:25:09-0500
```

对于OpenSSH来说，*-l*选项的操作有些不同，它不会显示公钥，而是显示密钥的指纹。（更详细的内容请参看“密钥的指纹”）。

```
# 仅对OpenSSH  
$ ssh-add -l  
1024 1c:3d:cc:1a:db:74:f8:e6:46:6f:55:57:9e:ec:d5:fc smith@client
```

要让OpenSSH显示公钥，可以使用*-L*选项：

```
# 仅对OpenSSH  
$ ssh-add -L  
1024 35 1604921766775161379181745950571099412502846... and so forth  
1024 37 1236194621955474376584658921922152150472844... and so forth
```

- *-d*，从代理中删除密钥。

注9：当然，X也有自己的安全问题。如果某人能连接到你的X服务器上，那么他也可以监视你所有的击键，包括你的口令。这在使用*ssh-askpass*中是否是一个关键问题要取决于你的系统和安全性的要求。

```
$ ssh-add -d ~/.ssh/second_id
Identity removed: /home.smith/.ssh/second_id (my alternative key)
```

```
$ ssh-add2 -d ~/.ssh2/id_dsa_1024_a
Deleting identity: id_dsa_1024_a.pub
```

如果没有指定密钥文件，*ssh-add1* 就会从代理中删除缺省的密钥：

```
$ ssh-add -d
Identity removed: /home.smith/.ssh/identity (smith@client)
```

另一方面，*ssh-add2* 要求必须指定密钥文件：

```
$ ssh-add2 -d
(nothing happens)
```

- *-D*，从代理中删除所有的密钥。这可以把现在加载的所有密钥都清空，留下代理空荡荡地运行。

```
$ ssh-add -D
All identities removed.
```

```
$ ssh-add2 -D
Deleting all identities.
```

- *-p*，从标准输入读取口令，而不是直接从tty读取口令。这在使用以下的Perl格式的脚本向*ssh-add*发送口令时十分有用：

```
open(SSHADD, "| ssh-add -p") || die "can't start ssh-add";
print SSHADD $passphrase;
close(SSHADD);
```

另外，*ssh-add2* 还有很多可以使用命令行选项控制的特性：

- 使用一个密码对代理进行加锁或解锁，这可以分别使用 *-L* 和 *-U* 选项实现。代理加锁之后就拒绝再接受所有的*ssh-add2* 操作，除非使用 *-U* 选项对其解锁。具体说来：

- 如果想修改代理的状态（增删密钥），系统就会显示：

```
The requested operation was denied.
```

- 如果想显示代理中的密钥，系统就会显示：

```
The authorization agent has no keys.
```

要对代理加锁，请输入：

```
$ ssh-add2 -L
Enter lock password: ****
```

```
Again: ****
```

解锁请输入：

```
$ ssh-add2 -U  
Enter lock password: ****
```

如果要离开计算机而不想退出登录，那么使用加锁操作就可以非常方便地对代理进行保护。当然也可以使用 *ssh-add -D* 命令清空所有的密钥，但是这样在回来时就必须重新加载这些密钥。如果只有一个密钥，使用这两种方法区别不大，但是如果有很多密钥，这可就是件让人头疼的事情了。不幸的是，加锁机制并不是绝对安全的。*ssh-agent2* 只是把加锁密码保存在内存中，此后礼貌地拒绝其他连接请求，直到接收到一条包含相同密码的解锁消息为止。但是加锁的代理仍然很容易受到攻击：如果有一个人侵者获得了对账号（或 root 账号）的访问权限，那么他就可以把代理的地址空间存储到文件中，并从中提取出密钥。加锁特性当然可以防止滥用代理，但是却不能防止这种攻击。如果用户非常关心密钥泄漏的问题，那么在完全依赖加锁机制之前可要考虑清楚。我们建议使用加锁密码对代理加载的密钥进行加密。这样同样可以方便用户使用，但是提供了更好的保护性能。

- 使用 *-t* 选项对密钥设置超时时间。通常在加载密钥之后，这个密钥不能确定是否还存在于代理中，直到代理结束或手工清空代理之后它才确定不再被使用了。*-t* 选项可以指定密钥的生命期，单位是分钟。在达到这段时间之后，代理就可以自动卸载密钥。

```
# 30分钟后卸载密钥  
$ ssh-add2 -t 30 mykey
```

- 使用 *-f* 和 *-F* 选项对代理转发进行限制。（代理转发可以在主机之间传输代理请求，我们很快就会介绍。）*-f* 选项让用户可以限制给定的密钥可以传输的距离。如果请求双方太远了（以主机之间的跳数来衡量），那么请求就会失败。把跳数（hop）设置为 0 就禁止转发这个密钥。

```
# 装载只能本地使用的密钥  
$ ssh-agent2 -f 0 mykey  
  
# 从不多于3跳的地方装载密钥并接收请求  
$ ssh-agent2 -f 3 mykey
```

-F 选项可以让用户对发起和这个密钥有关的请求的主机进行限制。这个选项有一个参数，该参数是一组可能会发起请求或转发请求的主机名、域名和 IP 地

址。这些参数使用逗号分隔开，中间可以使用通配符模式，用法和服务器范围的配置关键字 AllowHosts 和 DenyHosts 相同。[5.5.2.3]

```
# 允许密钥的请求转发仅在 example.com 域中进行
$ ssh-agent2 -F '*.example.com' mykey

# 允许从 server.example.com 和 harvard.edu 域中转发
$ ssh-agent2 -F 'server.example.com,*.harvard.edu' mykey

# 同上述命令，但限制转发 2 跳
$ ssh-agent2 -F 'server.example.com,*.harvard.edu' -f 2 mykey
```

警告：SSH1 代理不支持这种特性。如果用户正以 SSH1 兼容模式使用 SSH2 代理，就不能使用这种转发特性。

- 如果 *ssh-agent2* 正以 SSH1 兼容模式运行，可以使用 *-I*（是 I，而不是小写的 L）选项指定 SSH-1 客户端请求不能使用给定的密钥。这个密钥必须是一个 RSA 密钥，因为所有的 SSH1 公钥都是 RSA 密钥，而且唯一支持 RSA 密钥的 SSH-2 实现（在发布时）是 F-Secure SSH2 服务器。我们下面使用一个例子来说明这个特性。

- a. 首先生成一个 SSH2 RSA 密钥 *my-rsa-key*:

```
$ ssh-keygen2 -t rsa my-rsa-key
```

- b. 然后以 SSH1 兼容模式运行一个代理:

```
$ eval `ssh-agent2 -I`
```

- c. 把密钥加载到代理中:

```
$ ssh-add2 my-rsa-key
Enter passphrase: *****
```

在以 SSH1 兼容模式运行代理时，要注意密钥对于 SSH1 客户端和 SSH2 客户端都是可见的：

```
$ ssh-add1 -l
1023 33 753030143250178784431763590... my-rsa-key ...
$ ssh-add2 -l
Listing identities.
The authorization agent has one key:
my-rsa-key: 1024-bit rsa, smith@client, Mon Jun 05 2000 23:37:19 -040
```

现在让我们把这个密钥卸载掉并恢复实验环境：

```
$ ssh-add2 -D  
Deleting all identities.
```

这一次我们使用 *-I* 选项加载密钥，这样 SSH1 客户端就看不到这个密钥了：

```
$ ssh-add2 -l my-rsa-key  
Enter passphrase: *****
```

注意 SSH2 客户端仍然可以看到这个密钥：

```
$ ssh-add2 -l  
Listing identities.  
The authorization agent has one key:  
my-rsa-key: 1024-bit rsa, smith@client, Mon Jun 05 2000 23:37:19 -040
```

而 SSH1 客户端看不到这个密钥：

```
$ ssh-add1 -l  
The agent has no identities.
```

- 执行 PGP 密钥操作。*ssh-add2* 手册页中给出了 OpenPGP 密钥环操作使用的选项 *-R*、*-N*、*-P* 和 *-F*，但在本书（英文版）印刷时都还没有实现。

6.3.3.1 自动化代理加载（单 shell 方法）

每次在登录时都手工调用 *ssh-agent* 和 / 或 *ssh-add* 实在是件让人头疼的事情。在登录初始化文件中巧妙地设置一些内容，就可以自动调用一个代理并加载缺省的密钥。现在我们使用两种方法来介绍代理调用：单 shell 方法和子 shell 方法。

使用单 shell 方法，步骤如下：

1. 确保现在没有运行代理，这可以通过测试环境变量 *SSH_AUTH_SOCK* 或 *SSH2_AUTH_SOCK* 实现。
2. 使用 *eval* 运行代理 *ssh-agent1* 或 *ssh-agent2*。
3. 如果 shell 被连接到一个 tty 上，就使用 *ssh-add1* 或 *ssh-add2* 加载缺省的密钥。

对于 Bourne shell 及其派生的 shell (*ksh*、*bash*) 来说，可以在 *~/.profile* 中使用以下内容：

```
# 确保退出时 ssh-agent1 和 ssh-agent2 失效  
trap '  
    test -n "$SSH_AGENT_PID" && eval `ssh-agent1 -k` ;  
    test -n "$SSH2_AGENT_PID" && kill $SSH2_AGENT_PID  
' 0
```

```
# 如果没有运行代理且有一个终端，运行 ssh-agent 和 ssh-add
# (对 SSH2 而言，将其变化为使用 SSH2_AUTH_SOCK, ssh-agent2 和 ssh-add2)
if [ "$SSH_AUTH_SOCK" = "" ]
then
    eval `ssh-agent`
    /usr/bin/tty > /dev/null && ssh-add
fi
```

对于 C shell 和 tcsh 来说，可以在 *~/.login* 中加入以下内容：

```
# 使用 SSH2_AUTH_SOCK 代替 SSH2
if ( ! $SSH_AUTH_SOCK ) then
    eval `ssh-agent`
    /usr/bin/tty > /dev/null && ssh-add
endif
```

并在 *~/.logout* 文件末尾加入以下内容：

```
# ~/.logout
if ( "$SSH_AGENT_PID" != "" ) eval `ssh-agent -k`
if ( "$SSH2_AGENT_PID" != "" ) kill $SSH2_AGENT_PID
```

6.3.3.2 自动化代理加载（子 shell 方法）

在登录时加载代理的第二种方法是使用子 shell 方法调用代理。这一次用户需要在自己的登录初始化文件 (*~/.profile* 或 *~/.login*，也可以使用其他初始化文件) 和 shell 初始化文件 (*~/.cshrc*、*~/.bashrc* 等) 中增加一些内容。这种方法不能用于 Bourne shell，因为它没有 shell 初始化文件。

1. 在登录初始化文件中，要确保还没有运行代理，这可以通过测试环境变量 *SSH_AUTH_SOCK* 或 *SSH2_AUTH_SOCK* 实现。
2. 在登录初始化文件的最后一行加入 *exec ssh-agent*，从而派生一个子 shell。也可以运行另外一个初始化文件来配置这个子 shell。
3. 在 shell 初始化文件中，检查是否这个 shell 被连接到一个 tty 上以及这个代理是否还没有加载密钥。如果是，就使用 *ssh-add1* 或 *ssh-add2* 加载缺省的密钥。

现在让我们看一下如何使用 Bourne shell 和 C shell 以及 X Window 系统进行设置。对于由 Bourne 派生出来的 shell 来说，要在 *~/.profile* 文件的末尾加入以下内容：

```
test -n "$SSH_AUTH_SOCK" && exec ssh-agent $SHELL
```

这样会运行代理，并派生一个子 shell。如果希望定制这个子 shell 的环境变量，可以创建一个脚本（例如，`~/.profile2`）来实现，并这样使用：

```
test -n "$SSH_AUTH_SOCK" && exec ssh-agent $SHELL $HOME/.profile2
```

然后，如果代理中还没有加载密钥，就在 shell 初始化文件（对于 *ksh* 来说是 `$ENV`，对于 *bash* 来说是 `~/.bashrc`）中加入以下内容来加载缺省的密钥：

```
# 确定连到了tty上
if /usr/bin/tty > /dev/null
then
    # 为身份检查输出 "ssh-add -l"
    # 对 SSH2 而言，使用如下行：
    # ssh-add2 -l | grep 'no keys' > /dev/null
    #
    ssh-add1 -l | grep 'no identities' > /dev/null
    if [ $? -eq 0 ]
    then
        # 装载缺省身份。对 SSH2 而言使用 ssh-add2
        ssh-add1
    fi
fi
```

6.3.3.3 自动加载代理 (X Window 系统)

如果用户正在使用 X 并想运行代理来自动加载缺省的密钥，这非常简单，可以使用单 shell 方法。例如，在 X 启动文件 (`~/.xsession`) 中使用以下内容：

```
eval `ssh-agent`
ssh-add
```

6.3.4 代理和安全性

正如我们前面所述，代理不会把私钥泄漏给 SSH 客户端。相反，它可以响应客户端对密钥的请求。这种方法的安全性远远高于传递密钥的方法，但是仍然有一些安全问题。在完全相信代理模型之前必须正确理解这些问题：

- 代理依赖于外部访问控制机制。
- 代理也可能被攻击。

6.3.4.1 访问控制

当代理加载了私钥之后，就会产生一个潜在的安全性问题。此时代理如何来区分来自你的SSH客户端的合法请求和来自非授权地的非法请求呢？令人吃惊的是，代理的确能正确区分所有的连接！代理不会对客户端进行认证，它会通过自己的IPC通道（是一个 Unix 域 socket）对接收到的所有合式（well-formed）的请求进行响应。

那么代理又是如何来实现安全性的呢？实际上，主机操作系统负责保护IPC通道不被非授权地使用。对于 Unix 来说，这种保护是通过对 socket 设置文件权限来实现的。SSH1 和 SSH2 都会把代理套接字保存在一个系统保护的目录 */tmp/ssh-USERNAME* 中，其中 *USERNAME* 是你的登录名；而 OpenSSH 对应的目录是 */tmp/ssh-STRING*，其中 *STRING* 是根据代理的 PID 生成的一个随机文本。在这两种情况下，该目录的权限都设置为归你所有，其他用户都不能访问（700）：

```
$ ls -la /tmp/ssh-smith/
drwx----- 2 smith    smith      1024 Feb 17 18:18 .
drwxrwxrwt  9 root     root      1024 Feb 17 18:01 ..
srwx----- 1 smith    smith      0 May 14 1999 agent-socket-328
s-w--w--w-  1 root     root      0 Feb 14 14:30 ssh-24649-agent
srw----- 1 smith    smith      0 Dec  3 00:34 ssh2-29614-agent
```

在本例中，用户 *smith* 在这个目录中有几个和代理有关的套接字。属主为 *smith* 的两个套接字是在代理运行时创建的，归 *smith* 所有。第三个套接字归 *root* 所有，所有的用户对其都具有写入权限，它是由 SSH 服务器创建的，用来实现代理转发（注 10）。[6.3.5]。

系统把用户的套接字放入一个单独的目录中，这不仅仅是为了组织清晰，而且还为了满足安全性和可移植性的要求，因为不同的操作系统对套接字权限的处理方法不同。例如，Solaris 会将其完全忽略；即使把一个套接字的权限设置成 000（所有用户都不能访问），它也可以接收所有的连接。Linux 遵守套接字权限，但是把套接字设置成只写权限实际上是允许对这个套接字进行读写操作。为了处理各种实现的差异，SSH 实现就把用户的套接字单独放到一个由你所有的目录中，并对该目录的权限进行设置，禁止其他用户访问该目录中的套接字。

注 10：即使所有用户都对这个套接字具有可写权限，也只有用户 *smith* 可以访问它，因为我们对其父目录 */tmp/ssh-smith* 设置了权限。

之所以要使用`/tmp`中的一个子目录，而不使用`/tmp`本身，就是为了防止一种称为临时竞争（temp race）的攻击。临时竞争攻击利用了 Unix `/tmp` 目录中通常采用的一种“粘滞位（sticky）”所固有的竞争条件，从而允许任何用户都可以创建文件，并可以删除自己的文件。

6.3.4.2 攻击代理

如果运行代理的机器被攻击了，那么攻击者就可以轻而易举地获得对 IPC 通道的访问权限，从而又获得代理的访问权限。这就允许入侵者向代理发起请求，至少在一段时间内可以。每次退出或把密钥从代理中卸载掉时，这个安全漏洞就关上了。因此，用户应该只在可信的主机上运行代理，如果要离开计算机很长一段时间（例如晚上），就应该把代理卸载掉 (`ssh-agent -D`)。

由于代理不会主动释放密钥，因此如果一台机器被攻击了，代理会认为这个攻击者也是安全的。唉，实际上可不是这么回事情。经验丰富的攻击者一旦进入系统，就有很多法子可以获得密钥，例如：

- 窃取私钥文件，并试图猜测口令。
- 跟踪运行的进程，在输入口令时将其截获。
- 特洛依（trojan）木马攻击：给用户安装一个修改过的系统程序（例如登录程序、shell 或 SSH），从而窃取口令。
- 获得正在运行代理的内存空间的一份拷贝，直接从中分析出密钥（这种方法比其他方法更困难）。

最终的结论只有一个：只在可信主机上运行代理。如果系统其他方面除了问题，那么 SSH 也无能为力。

6.3.5 代理转发

迄今为止，我们的 SSH 客户端已经和同一台机器上的 SSH 代理进行了协商。使用一种称为代理转发（agent forwarding）的特性，客户端还可以和远程机器上的代理进行通信。这既是一种方便的特性（允许多台主机上的客户端使用一个代理），又是用来解决和防火墙有关的一种方法。

6.3.5.1 防火墙样例

假设你想从自己家里的计算机 H 连接到办公室的计算机 W 上，和很多企业的计算机一样，W 都位于一个网络防火墙之后，不能从 Internet 上直接访问，因此你就不能建立一个从 H 到 W 的 SSH 连接。嗯，这又如何是好呢？你打电话给技术支持部门，他们告诉你一个好消息。在你的公司有一个网关 B，或者称为一台“堡垒（bastion）”主机，它可以从 Internet 上访问，而且它上面运行了 SSH 服务器。这就是说既然防火墙禁止你直接用 SSH 连接到自己的计算机上，那么你就只好先建立一个从 H 到 B 的连接，然后再建立一个从 B 到 W 的连接，这样就可以访问 W 了。技术支持部门给你在主机 B 上增加一个账号，问题似乎解决了……果真如此吗？

出于安全性的考虑，公司限制你只能使用公钥认证访问自己的计算机。因此，在你家里的主机 H 上使用私钥建立 SSH 连接时，你能成功连接到主机 B 上，但是现在你又碰到一个障碍：同样是出于安全性的考虑，公司禁止用户把自己的 SSH 密钥保存在主机 B 上，因为主机 B 是暴露在 Internet 上的，如果 B 被攻击了，那么你的密钥就有可能被窃取。这可是个坏消息，因为 B 上的 SSH 客户端必须要使用你的密钥才能连接到主机 W 上你的账号中；而你的密钥现在是在家里的主机 H 上（图 6-5 说明了这个问题）。现在该怎么办呢？

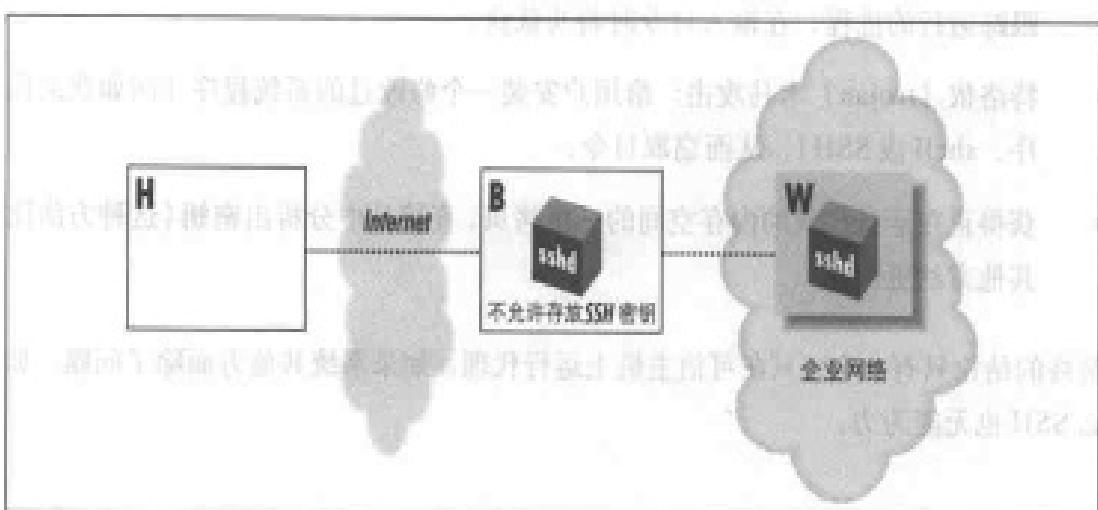


图 6-5：堡垒主机的情景

安装驱动 8.8.0

W (当然, 这并不安全) (注 11)。为了能找到一种解决方案, 我们重新回到 SSH 代理和代理转发上来。

SSH 代理转发允许在远程主机 (例如 B) 上运行的程序透明地访问你在 H 上的 *ssh-agent*, 条件是在主机 B 上也运行了代理。这样, B 上运行的远程 SSH 客户端就可以使用你在 H 上的密钥对数据进行签名和加密, 如图 6-6 所示。最终你可以从主机 B 上调用一个 SSH 会话来访问你的工作机器 W, 这样问题就完美地解决了。

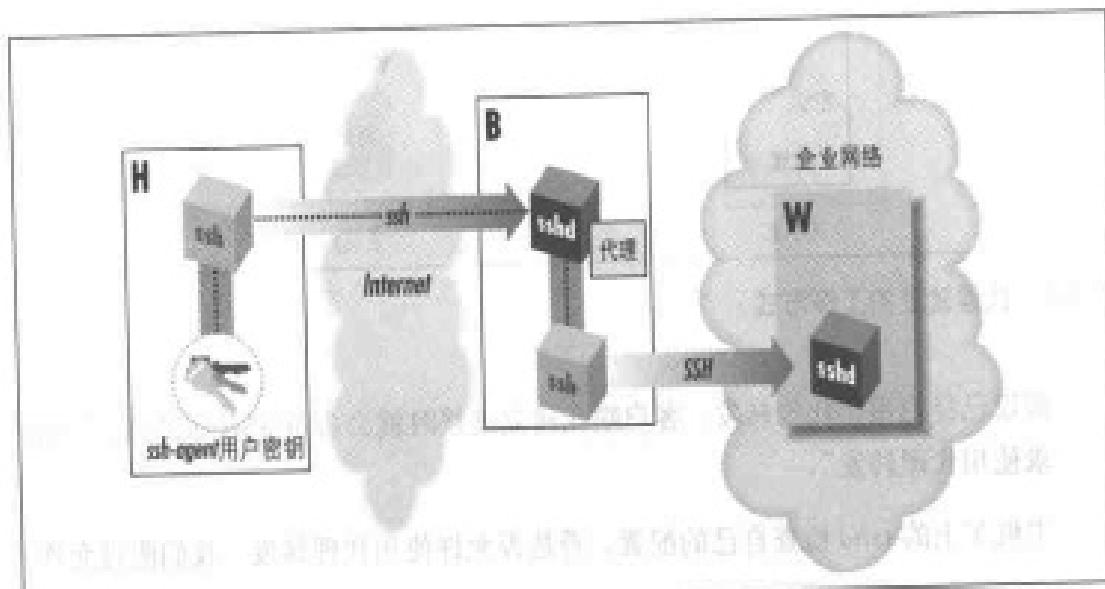


图 6-6: 使用 SSH 代理转发的解决方案

6.3.5.2 代理转发的工作方式

代理转发和所有的 SSH 转发 (第九章) 一样, 都是“幕后的英雄”。在这种情况下, SSH 客户端可以把对自己代理的请求, 通过一个预先建立的 SSH 会话, 转发到一个拥有所需要的密钥的代理上, 如图 6-7 所示。当然, 所有的传输都是在一个安全的 SSH 连接上进行的。下面让我们仔细研究一下要执行的每个步骤:

1. 假设已经登录到主机 X 上, 现在调用 ssh 建立一个到主机 Y 的远程终端会话:

```
# 在机器 X 上
$ ssh Y
```

注 11: 这种密钥分布的问题也可以使用网络文件共享协议 (例如, NFS、SMB 或 AFP) 来解决, 但是在我们现在讨论的远程访问的环境中通常都不能使用这种方法。

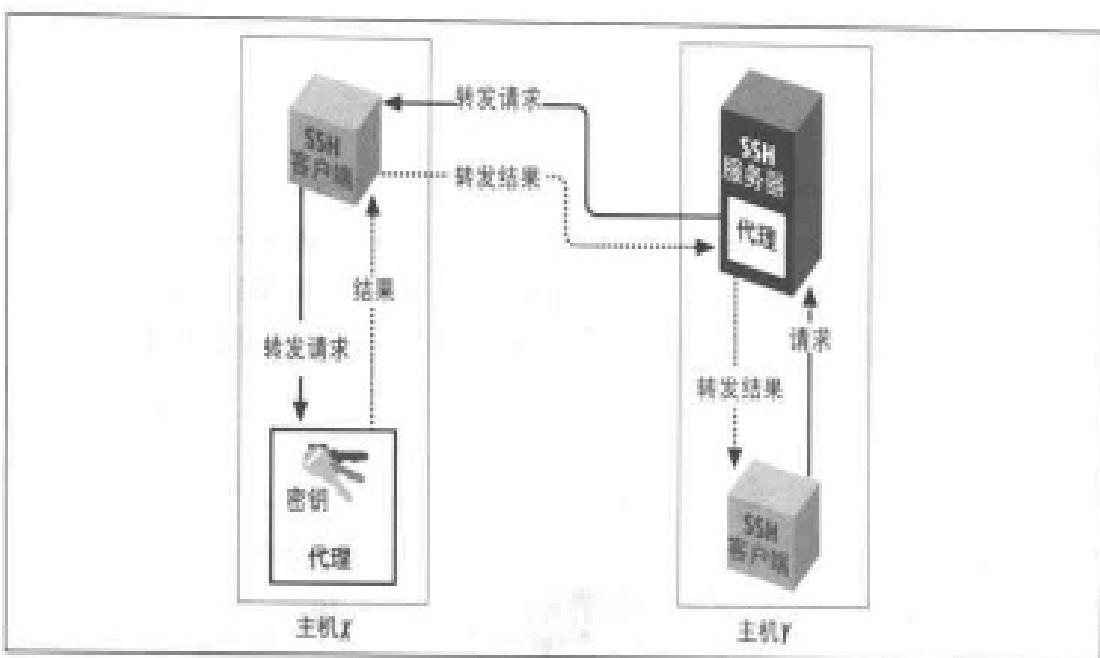


图 6-7：代理转发的工作方式

2. 假设已经启用了代理转发，客户端在建立连接时就会告诉 SSH 服务器，“我请求使用代理转发”。
3. 主机 Y 上的 *sshd* 检查自己的配置，看是否允许使用代理转发。我们假设允许。
4. 主机 Y 上的 *sshd* 创建一些 Unix 域套接字，并设置一些环境变量，从而建立一条到 Y 的本地进程间通信（IPC）通道。[6.3.2.1] IPC 机制的结果和 *ssh-agent* 要建立的通道一样。结果是 *sshd* 已经准备好担当 SSH 代理的重任了。
5. 现在在 X 和 Y 之间建立 SSH 会话。方式并不限于这种方式
6. 然后，你从主机 Y 上再运行一个 *ssh* 命令，和第三个主机 Z 建立一个 SSH 会话：
 * 在机器 Y 上
 \$ ssh Z
从这里开始的步骤，然后，从命令行输入，主机 Y 的用户名及密码
7. 现在这个新 *ssh* 客户端需要使用一个密钥才能连接到 Z 上。它知道主机 Y 上正在运行代理（就是由 *sshd* 充当一个代理），因此就通过代理的 IPC 通道发起一个认证请求。
8. *ssh* 获得这个请求，自己扮演一个代理，说，“你好，我是代理。你想做什么呢？”这个过程是透明的，客户端相信自己正在和一个代理进行通信。
9. 然后 *sshd* 就把这个和代理有关的请求通过 X 和 Y 之间建立的安全连接转发回

最初的机器 X。主机 X 上的代理接收到这个请求并访问自己的本地密钥，其响应被转发回主机 Y 上的 *sshd*。

10. 主机 Y 上的 *sshd* 把这个响应传递给客户端，这样到主机 Z 的连接就可以成功建立了。

使用代理转发，用户就可以从主机 Y 上透明地访问主机 X 上的任何密钥。因此，只要主机 X 上的密钥允许访问一个主机，那么主机 Y 上的任何 SSH 客户端就都可以访问这个主机。要测试这个功能，可以在主机 Y 上运行下面的命令显示密钥：

```
* 在机器 Y 上  
$ ssh-agent -l
```

用户会看到所有的密钥都是在主机 X 的代理中所加载的密钥。

要注意代理转发的这种关系是可以传递的：如果不间断重复这个过程，把一台台上机构成一个 SSH 链，那么最后一台主机上的客户端也可以访问第一台主机（X）上的密钥。（当然，这假设中间每台主机上的 *sshd* 都启用了代理转发。）

6.3.5.3 启用代理转发

在 SSH 客户端可以使用代理转发之前，必须先启用这个特性。不同的 SSH 实现对这个特性的缺省设置也可能不同，当然系统管理员可以对其进行修改。如果需要，用户也可以在客户端配置文件 *~/.ssh/config* 中使用 *ForwardAgent*（注 12）关键字来手工启用，该值可以为 yes（缺省值）或 no：

```
# SSH1, SSH2, OpenSSH  
ForwardAgent yes
```

同理，也可以使用 *-o* 命令行选项，它后面可以使用所有的配置关键字和值：

```
# SSH1, SSH2, OpenSSH  
$ ssh -o "ForwardAgent yes" ...
```

ssh 的 *-a* 选项可以禁用代理转发：

```
# SSH1, SSH2, OpenSSH  
$ ssh -a ...
```

注 12：SSH2 支持关键字 *AllowAgentForwarding*，它是 *ForwardAgent* 的一个同义词。

另外，*ssh2* 和 OpenSSH 的 *ssh* 都还可以使用一些选项来启用代理转发，但是这些都不是缺省设置：

```
# 仅对 SSH2  
$ ssh2 -a ...  
  
# 仅对 OpenSSH  
$ ssh -A ...
```

6.3.6 代理使用的 CPU

在结束代理的讨论之前，我们最后再介绍一下和性能有关的问题。代理承担了所有本来应该是 SSH 客户端要执行的加密操作。这就是说代理可能会占用很多 CPU 时间。在我们曾经见过的一种情况中，有些朋友使用 SSH1 来执行很多自动操作，在一行中运行了几百个存活期很短的 SSH 会话。这些朋友很惊奇地发现，这些 SSH 会话所使用的单个 *ssh-agent* 蚕食了这台机器上的大量 CPU。

6.4 使用多个身份标识

到目前为止，我们一直都假设你在一个 SSH 服务器上只有一个 SSH 标识。你确实有一个缺省的身份标识（前面的 *ssh-add* 例子就是对缺省的身份标识进行操作），但实际上可以想要多少个身份标识就可以有多少个。

为什么要使用多个身份标识呢？毕竟只需要有一个 SSH 标识就可以使用一个口令连接到远程机器上，这样既简单又方便，何乐而不为呢？实际上，大部分人只需要一个标识就能满足所有的要求。但是多个身份标识有很重要的用途：

提供附加安全性

如果使用不同的 SSH 密钥来访问不同的远程账号，万一其中一个被攻击了，也只会危及这一个远程账号而已。

增加批处理任务的安全性

使用空口令的 SSH 密钥，可以在进行通信的计算机之间建立一条安全连接，自动执行批处理任务，例如无人参与的备份操作。[\[11.1.2.2\]](#) 但是显然你不会想让普通的登录也使用未加密的私钥，因此就需要为这种目的再生成一个密钥。

对不同的账号进行不同的设置

用户可以根据连接所使用的密钥来配置自己的远程账号，从而执行不同的操作。例如，你可能会让自己的 Unix 登录会话根据使用的密钥运行不同的启动文件。

触发远程程序执行

用户的远程账号可能会在使用其他密钥时通过强制命令运行特殊的程序。

[8.2.4]

为了使用多个标识，需要知道如何在这些标识之间进行切换。这有两种方法：手工切换和使用代理自动切换。

6.4.1 手工切换身份标识

ssh 和 *scp* 都可以使用 *-i* 命令行选项和 *IdentityFile* 配置关键字切换自己的身份标识。不管使用哪种方法，都要提供要切换到的私钥文件（SSH1、OpenSSH）或 *identification* 文件（SSH2）。[7.4.2] 其语法小结如下：

表 6-2：语法小结

版本	<i>ssh</i>	<i>scp</i>	<i>IdentityFile</i> 关键字
SSH1, OpenSSH	<i>ssh1 -i key_file ...</i>	<i>scp1 -i key_file ...</i>	<i>IdentityFile key_file</i>
SSH2	<i>ssh2 -i id_file ...</i>	<i>scp2 -i id_file ...</i>	<i>IdentityFile id_file</i>

6.4.2 使用代理切换身份标识

如果使用了 SSH 代理，就可以自动切换身份标识。用户可以简单地使用 *ssh-add* 把所需要的标识加载到代理中。这样在建立连接时，SSH 客户端就可以向代理发送请求，并从代理中获得一些标识。然后客户端就依次试验每个标识，直到有一个标识认证成功或所有的标识都失败为止。即使有 10 个密钥用于 10 个 SSH 服务器，也需要一个代理（其中加载了这些密钥）就可以向 SSH 客户端提供适当的密钥信息，从而无缝地和这 10 个服务器进行认证。

所有这些都是透明的，对其他部分没有任何影响。是的，“几乎”没什么影响。但如果两个 SSH 标识可以连接到同一台 SSH 服务器上，就可能会产生两个问题。

第一个问题的产生是由于代理存储标识的顺序就是它从 *ssh-add* 中加载的顺序。正如我们前面所说的一样，SSH 客户端就会“依次”（也就是说按照从代理中获得标识的顺序）试验这些标识。否则，如果两个或多个标识都可以适用于一种情况，那么 SSH 客户端就可能会使用错误的标识进行认证。

例如，假设有两个 SSH1 标识分别保存在文件 *id-normal* 和 *id-backups* 中。用户使用 *id-normal* 标识和 *server.example.com* 进行普通的终端会话，使用 *id-backups* 标识来调用 *server.example.com* 上的远程备份程序（例如，使用强制命令[8.2.4]）。每天登录时，都要把这两个密钥加载到代理中，这可以使用一个脚本来定位并加载给定目录中的所有密钥文件实现：

```
#!/bin/csh
cd ~/.ssh/my-keys          # 一个目录样例
foreach keyfile (*)
    ssh-add $keyfile
end
```

当调用 SSH 客户端时又会发生什么情况呢？

```
$ ssh server.example.com
```

在这种情况下，在使用 *id-backups* 文件中的密钥进行认证之后，就可以运行远程备份程序。用户知道脚本中的通配符会以字母顺序返回一组密钥文件，因此要把 *id-backups* 放在 *id-normal* 之前，好比说应该这样输入：

```
$ ssh-add id-backups
$ ssh-add id-normal
```

因此，SSH 客户端在连接到 *server.example.com* 上时就会一直使用 *id-backups* 中的密钥，因为代理首先使用它来响应客户端的请求。你可能并不想这样。

第二个问题只会影响到以下行为：代理中的标识的优先级高于手工使用的标识。如果代理可以使用一个标识成功进行认证，就无法使用 *-i* 命令行选项或 *IdentityFile* 关键字来覆盖代理的设置。因此在我们前面那个例子中，就根本无法使用 *id-normal* 标识。即使这样使用：

```
$ ssh -i id-normal server.example.com
```

也仍然只能使用 *id-backups* 进行认证，因为它是最先加载到代理中的。即使没有加载的标识也不能覆盖代理的选择。例如，如果你现在只向代理中加载了一个标识，又要使用另外…个标识进行认证：

```
$ ssh-add id_normal  
$ ssh -i id-backups server.example.com
```

那么 *ssh* 连接就会依然使用早已加载的标识进行认证（此处是 *id-normal*），*-i* 选项根本没什么用处（注 13）。

因此我们可以总结出一条基本规则：如果在一个 SSH 服务器上有两个有效的 SSH 标识，那就不要把任何一个标识加载到代理中。否则，这两个标识就有一个不能访问服务器。

6.4.3 根据标识定制会话

虽然多个标识有上节介绍的那些缺点，但它仍然非常有用。特别是用户可以配置自己的远程账号，对不同的标识进行不同的响应。这可以分为三个步骤：

1. 生成一个新 SSH 标识，这一点我们在本章中已经讨论过了。
2. 按照需要使用新标识详细设置客户端配置。这是第七章的重点。
3. 对 SSH 服务器中的账号进行设置，让其以想要的方式对新标识进行响应。这方面的内容将在第八章中详细介绍。

我们强烈建议读者自行试验一下这种技术。通过这种方法来使用 SSH，可以实现十分强大而有趣的功能。如果用户只是使用 SSH 来运行简单的终端会话，那就不能享受其中的乐趣了。

注 13：这种行为没有任何文档进行说明，我们在明白到底是怎么一回事之前都快被它搞得崩溃了。在 SSH1 中使用 Kerberos 认证也是如此。如果你有 Kerberos 证书可以登录，同时你正在运行代理，现在你使用 *-i* 选项指定一个密钥，那么在你销毁 Kerberos 证书之前（或者采取某种方式将其设置为不可用，例如通过设置 KRB5CCNAME 变量将其隐藏），这个密钥一直都不能使用，因为 SSH 总是先使用 Kerberos 证书进行认证。

6.5 小结

在本章中我们介绍了如何创建和使用 SSH 标识，它使用密钥对表示，可能是单个密钥对（SSH-1），也可能是一组密钥对（SSH-2）。密钥是由 *ssh-keygen* 创建的，客户端可以根据需要进行访问。SSH-2 提供了另外一个配置层次：标识文件，用户可以把一组标识作为一个标识；也可以随意使用多个标识。

SSH 代理机制可以避免让用户重复输入口令，从而节省用户的时间。输入口令这种操作非常繁琐，一旦你对其感到厌烦了，就会义无反顾地使用代理。

第七章

客户端的 高级用法

本章内容

- 如何配置客户端
- 优先级
- 调用模式简介
- 深入客户端配置
- 使用SSH安全传输文件
- 小结

SSH客户端可以非常灵活地进行配置。第二章介绍了一些远程登录和文件拷贝的知识，但是那些内容不过是冰山一角而已。用户可以使用多个SSH标识连接服务器，使用各种认证和加密技术，体验控制TCP/IP的设置，并按照自己的喜好来定制SSH客户端的操作，甚至可以把一些SSH通用设置保存在配置文件中从而简化使用。

下面我们开始重点介绍SSH的用法，我们最终会用SSH客户端连接到远程主机上，所使用的组件如图7-1高亮显示的部分所示。有一个相关的主题本章没有介绍，即：如何控制到达的SSH连接到你的账号上。这种访问控制是SSH服务器的功能，而不是客户端的功能，这部分内容将在第八章中介绍。

7.1 如何配置客户端

客户端*ssh*和*scp*都可以很灵活地进行配置，它们有很多设置，用户可以按照自己的需要进行修改。如果想修改这些客户端的行为，可以使用三种通用的技术：

环境变量

对*scp*的行为进行较小的修改。

命令行选项

改变一次调用*ssh*或*scp*的操作。

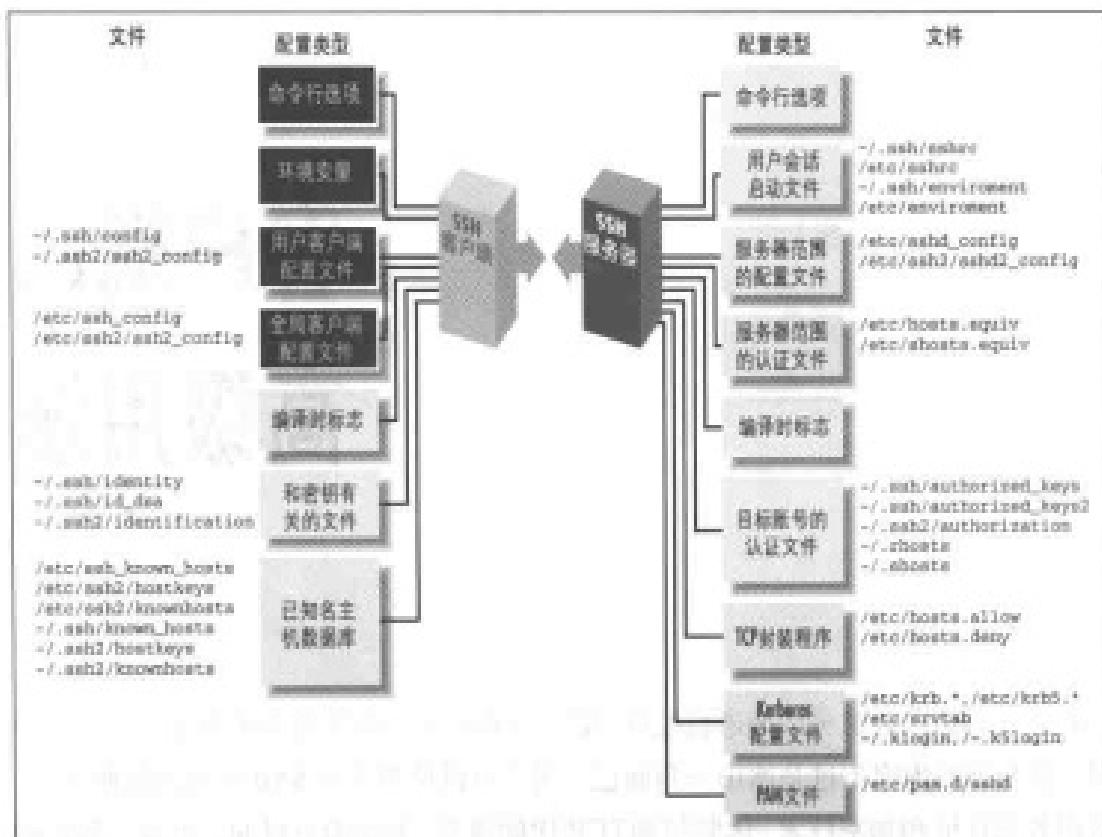


图 7-1：客户端配置（高亮显示部分）

配置关键字

强制进行修改，之后一直生效，直到再次修改为止；这些设置保存在客户端配置文件中。

现在我们就简要介绍一下这三种方法。

7.1.1 环境变量

客户端有一些特性可以使用环境变量控制。例如，您如果设置了环境变量 `SSH_ALL_SCP_STATS`，那么 `scp` 就会打印出它所传输的每个文件的统计信息。用户可以在自己当前的 shell 中使用标准的方法设置环境变量：

```
# C shell族 (csh, tcsh)
$ setenv SSH_ALL_SCP_STATS 1

# Bourne shell族 (sh, ksh, bash)
$ SSH_ALL_SCP_STATS=1
$ export SSH_ALL_SCP_STATS
```

另外，环境变量及其值也可以在文件中设置。系统管理员可以在`/etc/environment`中为所有的用户设置环境变量，用户也可以在自己的`~/.ssh/environment`（SSH1、OpenSSH）或`~/.ssh2/environment`（SSH2）中设置环境变量。这些文件都可以包含如下格式的行：

```
NAME=VALUE
```

其中`NAME`是环境变量名，而`VALUE`是该变量的值。该变量的值被设置成本行等号后面的内容。但是不要把值使用引号括起来，即使其中有空格也不要这样做，除非想把引号也作为该值的一部分。

7.1.2 命令行选项

使用命令行选项可以在调用前修改客户端的行为。例如，如果用户正通过一个慢速的modem连接使用`ssh1`，那么就可以使用`-C`命令行选项通知SSH1对数据进行压缩：

```
$ ssh1 -C server.example.com
```

`ssh`、`scp`和它们支持的大部分程序在使用`-h`命令行选项调用时，都会打印一条帮助消息，其中显示了所有可用的命令行选项。例如：

```
# SSH1, SSH2
$ ssh -h
$ ssh-keygen2 -h
```

7.1.3 客户端配置文件

如果不反复输入命令行选项，那么配置文件现在就让你可以修改客户端的行为，并在以后一直这样使用，直到再次修改配置文件为止。例如，用户可以在客户端的配置文件中插入这样一行，让调用的所有客户端都启用压缩功能：

```
Compression yes
```

在客户端的配置文件中，客户端的设置可以通过指定关键字的值来修改。在我们前面这个例子中，使用了关键字`Compression`，并将其值设置成`yes`。在SSH1和OpenSSH中，可以使用一个等号把关键字和其值分隔开：

```
Compression = yes
```

但是SSH2并不支持这种语法，它使用更简单的方法：使用“关键字<空格>值”的格式来避免引起混淆。

用户可以把客户端配置成访问每个远程主机都执行不同操作。这可以使用命令行选项来完成，但是用法却非常复杂，需要输入很长的命令，很快你就会讨厌这种不方便的用法了，例如：

```
$ ssh1 -a -p 220 -c blowfish -l sally -i myself server.example.com
```

用户也可以在配置文件中设置这些选项。以下这些设置可以实现和上面的命令行选项相同的功能，我们把这些选项都放到“myserver”之后：

```
# SSH1, OpenSSH
Host myserver
ForwardAgent no
Port 220
Cipher blowfish
User sally
IdentityFile myself
HostName server.example.com
```

要使用这些选项运行客户端，只需要简单地输入：

```
$ ssh1 myserver
```

配置文件要花些时间来设置，但是长期运行下来就会节省很多时间。

我们已经向你展示了一个配置文件的结构的格式：一个Host（主机）规范，后面跟着一长串关键字/值对。在后面的各节中，我们会继续使用这种方法，在解释每个关键字的具体含义之前先定义它的结构和基本规则。在介绍了共性的东西之后，我们就会具体介绍各个关键字。听起来不错吧？好，让我们开始吧。

7.1.3.1 关键字和命令行选项的比较

正如我们在很多配置关键字中会碰到的问题一样，如果需要，使用命令行可以提供所有的功能。-o命令行选项就是用于此目的。对于配置文件中这样一行内容来说：

Keyword	Value
Compression	yes

也可以输入（注 1）：

```
# SSH1, SSH2, OpenSSH
$ ssh -o "Keyword Value" ...
```

例如，这样的配置行：

```
User sally
Port 220
```

也可以这样使用：

```
# SSH1, SSH2, OpenSSH
$ ssh -o "User sally" -o "Port 220" server.example.com
```

SSH1 要求在关键字和值之间使用一个等号分隔符：

```
$ ssh1 -o User=sally -o Port=220 server.example.com
```

这个例子说明在一个命令行中可以多次使用 *-o* 选项。该选项在 SSH1 和 OpenSSH 中也可以用于 *scp*：

```
# SSH1, OpenSSH
$ scp -o "User sally" -o "Port 220" myfile server.example.com:
```

命令行选项和配置关键字之间的另外一种关系表现在 *-F* 选项（只能用于 SSH2）上。这个选项通知 SSH2 客户端使用另外一个配置文件，而不使用 *~/.ssh2/ssh2_config*。例如：

```
$ ssh2 -F ~/.ssh2/other_config
```

不幸的是 SSH1 和 OpenSSH 客户端没有相应的选项。

7.1.3.2 全局文件和本地文件

客户端的配置文件有两种风格：第一种是一个全局客户端配置文件，通常是由系统管理员创建的，用来维护整台计算机上客户端的行为。该文件通常是 */etc/ssh_config* (SSH1、OpenSSH) 或 */etc/ssh2/ssh2_config* (SSH2)。（不要把这些文件和相同目

注 1： 此处 SSH1 和 OpenSSH 还是允许在关键字及其值之间使用等号 (=)，以忽略命令行中的引号：*ssh -o Keyword = Value*。

求下的服务器配置文件混淆了。)每个用户都可以在自己的账号中创建本地客户端配置文件，通常是`~/.ssh/config` (SSH1、OpenSSH) 或`~/.ssh2/ssh2_config` (SSH2)。该文件对在用户登录会话中运行的客户端的行为进行控制 (注 2)。

用户本地文件中设置的优先级高于全局文件中的设置。例如，如果全局文件启用了数据压缩，而本地文件中又禁用了数据压缩，那么对于在你的账号中运行的客户端来说，就使用本地文件中的设置 (即不使用数据压缩)。我们很快就会更详细地介绍这些内容。[7.2]

7.1.3.3 配置文件段

客户端的配置文件可以划分成段。每个段都包含一个远程主机或一组相关的远程主机 (例如在一个给定域中的所有主机) 的设置。

不同的 SSH 实现使用的段开始标记也不同。对于 SSH1 和 OpenSSH 来说，新段以`Host` 关键字开始，后面跟着一个字符串，称为主机规范 (host specification)。该字符串可以是一个主机名：

```
Host server.example.com
```

也可以是一个 IP 地址：

```
Host 123.61.4.10
```

还可以是一个主机名别名：[7.1.3.5]

```
Host my-nickname
```

或者是一个表示一组主机名的通配符模式，其中“?”可以匹配一个任意字符，而“*”可以匹配一组字符串 (就像大家喜爱的 Unix shell 中的文件名通配符一样)：

```
Host *.example.com  
Host 128.220.19.*
```

注 2：系统管理员可能会用编译时标志`--with-etcdir` [4.1.5.1] 或服务器范围设置的关键字`UserConfigDirectory` [5.4.1.5] 改变客户端配置文件的位置。如果你的计算机中这些文件没在它们的缺省位置上，请与你的系统管理员联系。

下面是一些通配符的例子：

Host *.edu	edu 域中的任意主机名
Host a*	以 a 开头的任意主机名
Host *1*	名字或 IP 地址中包含 1 的任意主机
Host *	任意主机或 IP 地址

对于 SSH2 来说，新段是用一个主机规范字符串后面跟上一个冒号进行标记。这个字符串就像是上面的 Host 参数，可以是一个计算机名：

server.example.com:

也可以是一个 IP 地址：

123.61.4.10:

还可以是一个别名：

my-nickname:

或通配符模式：

*.example.com:

128.220.19.*:

主机规范行后面是一个或多个设置(即配置关键字和值)，这一点我们在前面的例子中已经看到过了。下表对 SSH1 和 SSH2 的配置文件进行了比较：

SSH1, OpenSSH	SSH2
Host myserver	myserver:
User sally	User sally
IdentityFile myself	IdentityFile myself
ForwardAgent no	ForwardAgent no
Port 220	Port 220
Cipher blowfish	Ciphers blowfish

这些设置都可以应用于主机规范中的主机名。本段结束于下一段的开始，或文件的末尾，取二者中靠前的一个。

7.1.3.4 多次匹配

因为在主机规范中可以使用通配符，所以一个主机名可以和配置文件中的两个或更多个段匹配。例如，如果一个段开头的内容为（注 3）：

```
Host *.edu
```

而另外一个段开头的内容为：

```
Host *.harvard.edu
```

现在要连接到 *server.harvard.edu* 上，应该使用哪一个段呢？不管你相信不相信，这两个段都会起作用。每个匹配段都起作用；而如果一个关键字被多次设成不同的值，却只有一个值管用。对于 SSH1 和 OpenSSH 来说，最先出现的值优先级最高；而对于 SSH2 来说，则是最后一个值优先级最高。

假设客户端的配置文件中包含了两个段，都会对数据压缩、密码认证和密码提示进行控制：

```
Host *.edu
  Compression yes
  PasswordAuthentication yes

Host *.harvard.edu
  Compression no
  PasswordPromptLogin no
```

现在你想连到 *server.harvard.edu* 上：

```
$ ssh server.harvard.edu
```

注意字符串 *server.harvard.edu* 可以同时匹配 Host 模式 **.edu* 和 **.harvard.edu*。正如我们前面已经说过的一样，这两个段中的所有关键字都会对用户的连接起作用。因此，上面这个 *ssh* 命令会设置关键字 *Compression*、*PasswordAuthentication* 和 *PasswordPromptLogin* 的值。

但是要注意，这个例子中的两个段对 *Compression* 的设置不同。这会导致什么结果

注 3：为了叙述的简洁性，我们这里只使用了 SSH1 文件的语法。不过其中的道理对 SSH2 也适用。

呢？规则是第一个值的优先级高，在本例中，Compression被设置成yes，因此在前面这个例子中，*server.harvard.edu*所使用的设置是：

Compression yes	第一个 Compression 的设置
PasswordAuthentication yes	第一段惟一设置
PasswordPromptLogin no	第二段惟一设置

结果如图 7-2 所示。Compression no 会被忽略，因为这是配置文件中的第二个 Compression 行。同理，如果有 10 个不同的 Host 行可以匹配 *server.harvard.edu*，那么这 10 个段的设置都起作用；而如果一个特定的关键字被设置了多次，那么只有一个值最终管用。

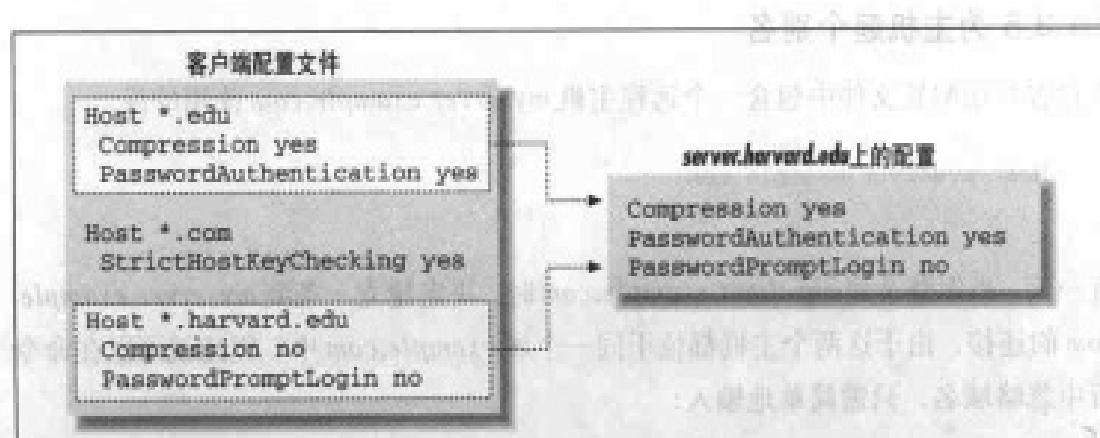


图 7-2：可以多次匹配的 SSH1 客户端配置文件（没有给出 SSH2）

虽然这种特性看起来似乎有些让人困惑不解，但它有一种非常有用的属性。假设想让某些关键字的设置可以应用于所有的远程主机，就可以简单地创建这样一个段：

*Host **

并在这里设置通用的设置。该段应该是配置文件中的第一段或最后一段。如果是第一段，那么其设置的优先级就最高。这可以用于防止自己会出现的错误。例如，如果用户想确保自己不会不小心把 SSH 会话又配置成不安全的 rsh 协议，就可以在配置文件的开头这样设置：

```
* 文件首段
Host *
FallBackToRsh no
```

反之，如果把 *Host ** 当作配置文件的最后一段，那么其设置只有在没有其他段会

覆盖本段中的设置时才会管用。这可以用于修改 SSH 的缺省行为，同时仍能允许对这些设置进行覆盖。例如，缺省情况下数据加密的特性是被禁用的，可以在配置文件的最后这样设置，从而缺省地启用这个特性：

```
# 文件末段  
Host *  
Compression yes
```

如何？你已经修改了自己的账号使用 *ssh* 和 *scp* 时的缺省行为了！配置文件中在此之前出现的任何段都可以简单地把 *Compression* 设置成 *no* 来覆盖这种缺省设置。

7.1.3.5 为主机起个别名

假设客户端配置文件中包含一个远程主机 *myserver.example.com* 使用的段：

```
Host myserver.example.com  
...  
...
```

有一天，当你登录到 *ourclient.example.com* 时，决定建立一条到 *myserver.example.com* 的连接。由于这两个主机都位于同一个域 *example.com* 中，因此就可以在命令行中忽略域名，只需简单地输入：

```
$ ssh myserver
```

这的确可以建立 SSH 连接，但是你所使用的配置文件中的设置就有些不对了。*ssh* 会把命令行中的字符串“*myserver*”和主机字符串“*myserver.example.com*”进行比较，确定这两个字符串不能匹配，因此就不会把配置文件中的这段内容应用于你的连接。是的，软件需要对命令行和配置文件中的主机名进行精确的文本匹配。

你可以通过把 *myserver* 声明成 *myserver.example.com* 的一个别名来解决这个问题。在 SSH1 和 OpenSSH 中，这是使用 *Host* 和 *HostName* 关键字实现的。可以在 *Host* 后面跟上一个别名，然后在 *HostName* 后面给出该主机的完整主机名：

```
# SSH1, OpenSSH  
Host myserver  
HostName myserver.example.com  
...  
...
```

现在 *ssh* 就可以识别出该段可以应用于命令 *ssh myserver*。可以为一个给定的计算机任意起别名，即使别名和原来的主机名没有任何关系也无妨：

```
# SSH1, OpenSSH
Host simple
  HostName myserver.example.com
  ...
```

然后就可以在命令行中使用别名了：

```
$ ssh1 simple
```

对于 SSH2 来说，语法有些不同，但效果一样。用法是在主机规范中使用别名，并把 Host 关键字设置成完整的主机名。

```
# 仅对 SSH2
simple:
  Host myserver.example.com
  ...
```

然后输入：

```
$ ssh2 simple
```

别名可以方便地用于测试新客户端的设置。假设在配置文件中有一段对 *server.example.com* 进行设置：

```
Host server.example.com
  ...
```

而且你想试验一下各种不同的设置。你可以直接修改相应的设置，但是如果设置不能生效，就只好再不辞辛苦地把设置修改回来。以下这些步骤提供了一种更方便的方法：

1. 在配置文件中，把要修改的段拷贝一份：

```
# 最初
Host server.example.com
  ...
# 为测试而拷贝
Host server.example.com
  ...
```

2. 在拷贝的副本中把“Host”修改成“HostName”：

```
# 最初
Host server.example.com
  ...
# 为测试而拷贝
```

```
HostName server.example.com
```

```
...
```

3. 在副本的开头加入一个新的 Host 行，并使用别名，例如“Host my-test”：

```
# 最初
```

```
Host server.example.com
```

```
...
```

```
# 为测试而拷贝
```

```
Host my-test
```

```
HostName server.example.com
```

```
...
```

4. 设置完成。在副本 (my-test) 中执行想要的修改操作并使用 `ssh my-test` 进行连接。可以通过运行 `ssh server.example.com` 和 `ssh my-test` 来方便地对原来和现在的行为进行比较。如果决定不再使用新的设置，只需要简单地删除 my-test 段即可。如果想保留所修改的内容，把它们拷贝到原来的段中（或把原来的段删除并保留这个副本）即可。

对于 SSH2 也可以进行类似地处理：

```
# 最初
```

```
server.example.com:
```

```
...
```

```
# 为测试而拷贝
```

```
my-test:
```

```
Host server.example.com
```

7.1.3.6 注释、缩进和风格

在前面的例子中，你可能已经注意到我们使用 # 来表示注释：

```
# 这是注释
```

实际上，配置文件中所有以 # 开始的行都被当作注释而忽略。同理，空行（该行为空或只有空格）也会被忽略。

大家可能也已经注意到主机规范后面的行都采用了缩进：

```
# SSH1, OpenSSH
Host server.example.com
    Keyword1 value1
    Keyword2 value2
```

```
# 仅对 SSH2
server.example.com:
    Keyword1 value1
    Keyword2 value2
```

缩进是一种很好的编程风格，因为它可以很直观地显示一个新段的开始。虽然缩进不是必需的，但是我们推荐使用缩进风格。

7.2 优先级

也许你会纳闷：如果有些配置设置相互冲突会导致什么后果呢？例如，如果使用 `Compression` 关键字禁用了压缩功能，又使用 `-C` 命令行选项来启用压缩功能，到底哪种设置会有效呢？换而言之，究竟哪一个设置的优先级高呢？

对 SSH1、SSH2 和 OpenSSH 客户端来说，优先级从高到低的顺序依次是：

1. 命令行选项。
2. 用户的本地配置文件。
3. 全局客户端配置文件（注 4）。

命令行选项的优先级最高，它可以覆盖所有客户端配置文件的设置。用户的本地文件的优先级仅次于命令行选项，全局文件的优先级最低。因此在我们这个压缩的例子中，`-C` 的优先级高于 `Compression` 关键字的优先级，因此最终会启用压缩功能。如果我们既没有使用关键字也没有使用命令行来修改一个设置，那么它就使用客户端的缺省设置。

记住，我们正在讨论的问题是从客户端发出的外发连接。所有到达的连接都是由 SSH 服务器控制的，其优先级的规则与此不同。对于服务器来说，用户的本地配置文件绝对不会覆盖全局文件；否则，用户就可以覆盖全局服务器的设置，这样会造成很多安全漏洞和其他缺陷。[\[8.1.1\]](#)

注 4： 这里没有提到环境变量，因为环境变量不参与优先级确定。环境变量控制的功能与命令行和配置文件并没有重叠。

7.3 详细模式简介

现在我们已经介绍了命令行选项和配置文件的基本知识，接下来我们将对配置进行深入讨论。在开始之前，让我们预先做些防范工作。在试验这些选项时，用户有时候得到的结果并不是自己想要的。不管什么时候发生这种情况，用户的本能反应应该是：使用 `-v` 命令行选项打开详细模式来查看问题之所在：

```
# SSH1, SSH2, OpenSSH
$ ssh -v server.example.com
```

在详细模式中，客户端会随着处理的进行不断打印消息，这样就为我们解决问题提供了线索。在碰到问题时，SSH的新用户（也有一些老用户）经常会忘记或忽视使用详细模式。别犹豫了！我们在 Usenet SSH 新闻组 *comp.security.ssh* [12.3] 中所见到的很多问题只要运行 `ssh -v` 检查一下输出结果就立即能得到答案。

假设用户刚刚在 *server.example.com* 上安装了公钥，现在想试验使用这个公钥进行认证。奇怪的是，系统提示输入登录密码，而不是公钥口令：

```
$ ssh server.example.com
barrett@server.example.com's password:
```

别只坐在那里挠头叹气，让详细模式来拯救你吧：

```
$ ssh -v server.example.com
SSH Version 1.2.27 [sparc-sun-solaris2.5.1], protocol version 1.5.
client: Connecting to server.example.com [128.9.176.249] port 22.
client: Connection established.
client: Trying RSA authentication with key 'barrett@client'
client: Remote: Bad file modes for /users/barrett/.ssh 噢，问题就在这儿!
client: Server refused our key.
client: Doing password authentication.
barrett@server.example.com's password:
```

这些消息（本例中已经精简过了）显示 SSH 连接已经成功了，而公钥认证失败了。其原因是“文件模式不对”：远程 SSH 目录 */home/barrett/.ssh* 的权限不对。赶快登录到服务器上，运行一个适当的 `chmod` 命令，问题就解决了：

```
# 在服务器上
$ chmod 700 ~/.ssh
```

scp 也可以使用详细模式：

```
$ scp -v myfile server.example.com:  
Executing: host belvedere, user (unspecified), command scp -v -t .  
SSH Version 1.2.27 [sparc-sun-solaris2.5.1], protocol version 1.5.  
...
```

详细模式是你的朋友，大方地使用它吧。现在我们已经准备好去学习更多的选项了。

7.4 深入客户端配置

ssh 和 *scp* 都会受到命令行选项、配置文件关键字及环境变量的影响。SSH1、SSH2 和 OpenSSH 客户端的行为不同，使用的设置也不同，但是我们通常都对其进行同时介绍。当一个设置只能由一些指定产品支持时，我们会单独进行说明。

7.4.1 远程账号名

ssh 和 *scp* 假设你的本地用户名和远程用户名相同。如果本地用户名是 *henry*，并运行：

```
# SSH1, SSH2, OpenSSH  
$ ssh server.example.com
```

那么 *ssh* 就会认为远程用户名也是 *henry*，而且你请求的连接是对 *server.example.com* 上的 *henry* 账号发起的。如果远程账号名和本地账号名不同，那么你必须把远程账号名告诉 SSH 客户端。如果 *henry* 要连接到远程账号 *sally* 上，他可以使用 *-l* 命令行选项：

```
# SSH1, SSH2, OpenSSH  
$ ssh -l sally server.example.com
```

如果使用 *scp* 拷贝文件，那么用来指定远程账号名的语法也不相同，它看起来像是一个 email 地址。[\[7.5.1\]](#) 要把文件 *myfile* 拷贝到 *server.example.com* 上的远程账号 *sally* 中，可以这样使用：

```
# SSH1, SSH2, OpenSSH  
$ scp myfile sally@server.example.com:
```

如果要频繁地使用一个不同的用户名连接到一个远程机器上，那么就不应该使用命令行选项了，而是应该在客户端的配置文件中指明远程用户名。*User* 关键字就是用

于这个目的的，*ssh*和*scp*都可以使用这个关键字。下表显示了如何在给定的远程主机上声明用户名是sally：

SSH1, OpenSSH	SSH2
Host server.example.com	server.example.com:
User sally	User sally

现在，再连接*server.example.com*时，就不用指定远程用户名是sally了：

```
# 将自动使用远程用户名sally
$ ssh server.example.com
```

7.4.1.1 欺骗远程账号名

使用User和主机别名，可以显著地缩短在*ssh*和*scp*中输入的命令行的长度。我们继续介绍“sally”的例子，如果配置如下表所示：

SSH1, OpenSSH	SSH2
Host simple	simple:
HostName server.example.com	Host server.example.com
User sally	User sally

那么这些长命令：

```
$ ssh server.example.com -l sally
$ scp myfile sally@server.example.com:
```

就可以简化成：

```
$ ssh simple
$ scp myfile simple:
```

下表显示了在不同的主机上如何指定多个不同的账号，每个账号都使用配置文件中自己的一个段：

SSH1, OpenSSH	SSH2
Host server.example.com	server.example.com:
User sally	User sally
...	...

SSH1, OpenSSH	SSH2
Host another.example.com	another.example.com:
User sharon	User Sharon
...	...

如果在每个远程主机上都只有一个账号，那么这种技术就十分方便。但是假设在 *server.example.com* 上有两个账号，分别为 sally 和 sally2，有什么办法可以在配置文件中指定这两个账号呢？下面这种用法是不行的（我们只给出 SSH1 的语法）：

```
# 这样并不能正确生效
Host server.example.com
User sally
User sally2
Compression yes
```

因为只有第一个值（sally）才能管用。要绕开这种限制，可以在配置文件中使用别名：一个主机创建两个段，每个段指定一个不同的用户：

```
# SSH1, OpenSSH
# Section 1: 对 sally 账号快捷访问
Host sally-account
HostName server.example.com
User sally
Compression yes

# Section 2: 对 sally2 账号快捷访问
Host sally2-account
HostName server.example.com
User sally2
Compression yes
```

现在就可以使用别名来方便地访问这两个账号了：

```
$ ssh sally-account
$ ssh sally2-account
```

这种方法的确可以，但是却不够理想。如果这样使用，就必须在每段中都复制一些设置（HostName 和 Compression）。复制增加了配置文件维护的难度，因为以后不管更改什么内容都需要更改两次。（通常，复制并不符合软件工程的思想。）难道注定只能使用复制么？不，实际上还有更好的方法。在这两个段之后，可以紧跟着创建第三个段，其中使用 Host 通配符，让它可以匹配 sally-account 和 sally2-account。假设使用 sally*-account，并把所有重复的设置都移到这个新段中：

```
# SSH1, OpenSSH
Host sally*-account
  HostName server.example.com
  Compression yes
```

最终结果如下表所示：

SSH1, OpenSSH	SSH2
Host sally-account	sally-account:
User sally	User sally
Host sally2-account	sally2-account:
User sally2	User sally2
Host sally*-account	sally*-account:
HostName server.example.com	Host server.example.com
Compression yes	Compression yes

由于 `sally*-account` 和两个段都可以匹配，因此其中的完整主机名和压缩的设置对 `sally-account` 和 `sally2-account` 都管用。`sally-account` 和 `sally2-account` 之间不同的设置（本例中是 `User`）都被保留在各自的段中。现在不用复制设置就已经达到了和前一个例子相同的结果——两个账号在同一台远程主机上具有不同的设置。

7.4.2 用户身份标识

SSH 使用一个密钥对（SSH-1）或一组密钥对（SSH-2）来标识用户的身份。^[6.1]通常，SSH 客户端使用缺省的密钥文件（SSH1、OpenSSH）或缺省的身份标识文件（SSH2）来建立认证连接。然而，如果已经创建了其他密钥，那么也可以通知 SSH 客户端使用这些密钥来构建身份标识。有一个命令行选项（`-i`）和一个配置关键字（`IdentityFile`）都是用于这个目的的。

例如，在 SSH1 和 OpenSSH 中，如果有一个名为 `my-key` 的私钥文件，就可以使用下面的命令让客户端使用这个私钥文件：

```
$ ssh1 -i my-key server.example.com
$ scp1 -i my-key myfile server.example.com:
```

也可以使用配置关键字：

```
IdentityFile my-key
```

该文件的位置假设是相对于当前目录的相对路径，也就是说，上面的例子中该文件是 `./my-key`。

SSH2 也有 `-i` 命令行选项和 `IdentityFile` 关键字，但是其用法稍有不同。它不使用密钥文件，而是使用身份标识文件名（注 5）：

```
$ ssh2 -i my-id-file server.example.com  
IdentityFile my-id-file
```

注意一下 `ssh1` 和 `ssh2` 的这种区别。如果用户向 `ssh2` 错误地提供了一个密钥文件名，那么客户端就试图将这个密钥文件当作一个身份标识文件读取，并向 SSH2 服务器发送一个随机结果。结果认证就神秘地失败了，同时可能会在日志中记录“`No further authentication methods available`”，或者提示用户输入登录密码，而不是公钥口令。

使用多个身份标识也十分有用。[6.4]例如，在使用第二个密钥时，用户可以设置远程账号运行特定的程序。通常使用命令：

```
$ ssh server.example.com
```

来初始化一个普通的登录会话，但是：

```
$ ssh -i other_identity server.example.com
```

可以在 `server.example.com` 上运行一个复杂的批处理任务。使用配置关键字，可以通过指定如下表所示的另外一个身份标识来达到相同的效果：

SSH1, OpenSSH	SSH2
<code>Host SomeComplexAction</code> <code>HostName server.example.com</code> <code>IdentityFile other_identity</code> <code>...</code>	<code>SomeComplexAction:</code> <code>Host server.example.com</code> <code>IdentityFile other_identity</code> <code>...</code>

注 5： 在 SSH2 2.0.13 及更早的版本中，`-i` 选项和 `IdentityFile` 要求身份标识文件得在用户自己的 SSH2 目录 `~/.ssh2` 下，SSH2 2.1.0 以后才接受了绝对路径名；所有不以斜线（/）开始的路径都被当作相对于 `~/.ssh2` 的路径处理。

然后就可以调用：

```
$ ssh SomeComplexAction
```

SSH1 和 OpenSSH 都可以在一个命令行中指定多个身份标识（注 6）：

```
# SSH1, OpenSSH
$ ssh -i id1 -i id2 -i id3 server.example.com
```

或者在配置文件中指定：

```
# SSH1, OpenSSH
Host server.example.com
IdentityFile id1
IdentityFile id2
IdentityFile id3
```

所提供的多个身份标识会依次尝试，直到有一个能成功认证为止。但是，SSH1 和 OpenSSH 限制用户在每个命令中最多只能提供 100 个身份标识。

如果想频繁地使用多个身份标识，就要记得使用 SSH 代理。使用 *ssh-add* 简单地把每个身份标识的密钥装载到代理中，以后在工作时就不用反复输入多个口令了。

7.4.3 主机密钥和已知名主机数据库

每个 SSH 服务器都有一个主机密钥，[3.3]这个主机密钥用来向客户端唯一地标识服务器。它有助于防止伪装攻击。当 SSH 客户端请求建立一个连接并接收服务器主机密钥时，这个客户端就要根据自己本地的已知名密钥数据库对这个密钥进行验证。如果密钥可以匹配，就允许连接继续进行。如果不能匹配，客户端就根据用户控制的几个选项的设置进行相应的操作。

在 SSH1 和 OpenSSH 中，主机密钥数据库有一部分是在服务器范围 (*/etc/ssh_known_hosts*) 进行维护的；另一部分在用户的 SSH 目录中 (*~/.ssh/known_hosts*) 维护（注 7）。在 SSH2 中有两个主机密钥数据库，用于认证服务器 (*/etc/ssh2/hostkeys* 中

注 6： SSH2 的身份标识文件中可包含多个密钥，也能完成同样的功能。

注 7： OpenSSH 还把 SSH-2 的已知名主机密钥保存在 *~/.ssh/known_hosts2* 文件中。

的“hostkeys”映射)和客户端(“knownhosts”映射),在本节中我们只介绍前者。和SSH1类似,SSH2的主机密钥映射也是在服务器范围(*/etc/ssh2/hostkeys/*)和每账号目录(*~/.ssh2/hostkeys/*)中维护的。在本节中,我们使用SSH1、SSH2和OpenSSH映射来介绍主机密钥数据库。

7.4.3.1 严格检查主机密钥

假设你请求和*server.example.com*建立一个SSH连接,它在响应时把自己的主机密钥发给你。你的客户端在自己的主机密钥数据库中查找*server.example.com*。理想情况下,可以找到匹配项并继续执行连接。但是如果没有任何可以匹配的项怎么办呢?这可能出现两种情况:

情况1

在数据库中找到了一个*server.example.com*使用的主机密钥,但是和接收到的密钥不匹配。这说明可能正有人对你进行攻击,或者*server.example.com*已经修改了自己的主机密钥,后一种情况也是合理的。[\[3.10.4\]](#)

情况2

数据库中没有*server.example.com*的主机密钥。在这种情况下,SSH客户端是第一次遇到*server.example.com*。

在这两种情况下,客户端应该继续进行连接还是中断连接呢?它是否应该把这个新主机密钥保存到数据库中呢?这些决定都是由**StrictHostKeyChecking**关键字控制的,它可以有三个值:

yes

严格检查。如果密钥未知或已经改变,那么连接就失败。该值最安全,但是通常在连接到新主机上或远程主机密钥会频繁改变时很不方便,也很烦人。

no

不严格检查。如果密钥未知,就将其自动加入用户数据库并继续处理连接。如果密钥改变了,就不修改现在的主机密钥,而是打印一条警告消息,并允许连接继续进行。这是最不安全的一种设置。

ask

提示用户决定。如果密钥未知，就询问用户是否应该将其加入用户数据库，以及是否允许这个连接继续。如果密钥改变了，就询问是否允许连接。这是缺省值，对于经验丰富的用户是个很好的方法。（经验较少的用户可能不明白问题是什么，因此可能会做出错误的决定。）

这里有一个例子：

```
# SSH1, SSH2, OpenSSH
StrictHostKeyChecking yes
```

表 7-1 对 SSH 的 StrictHostKeyChecking 的行为进行了总结。

表 7-1: StrictHostKeyChecking 的行为

找到密钥否？	匹配否？	是否进行严格检查	动作
Yes	Yes	-	连接
Yes	No	Yes	警告，连接失败
Yes	No	No	警告，连接继续
Yes	No	Ask	警报并询问是否允许连接
No	-	Yes	警告，连接失败
No	-	No	添加密钥，连接继续
No	-	Ask	询问是否允许添加密钥，连接继续

OpenSSH 还有另外一个关键字 CheckHostIP，可以让客户端根据数据库中的值对 SSH 服务器的 IP 地址进行验证。该值可以是 yes（缺省值，验证地址）或 no。该值为 yes 可以防止名字服务欺骗攻击。[3.10.2]

```
# 仅对 OpenSSH
CheckHostIP no
```

7.4.3.2 移动已知名主机文件

SSH1 和 OpenSSH 允许使用配置关键字来修改主机密钥数据库的位置，服务器范围和每账号所维护的部分都可以修改。GlobalKnownHostsFile 为服务器范围的文件指定其他位置。它实际上并不能移动文件（只有系统管理员可以执行这个操作），但是它可以强制客户端使用其他文件。如果文件过期了，或者想让客户端忽略服务器

范围的配置文件，那么这个关键字就非常有用，特别是在你已经被客户端上有关密钥改变的警告消息搞得头晕脑胀时更是如此。

```
# SSH1, OpenSSH
GlobalKnownHostsFile /users/smith/.ssh/my_global_hosts_file
```

同理，可以使用 UserKnownHostsFile 关键字来修改每账号配置使用的数据库的位置：

```
# SSH1, OpenSSH
UserKnownHostsFile /users/smith/.ssh/my_local_hosts_file
```

7.4.4 TCP/IP 设置

SSH 使用 TCP/IP 作为传输机制。大部分时间都不用修改缺省的 TCP 设置，但是在以下这些特殊情况下，我们必须修改这些设置：

- 连接到其他 TCP 端口的 SSH 服务器上。
- 使用特权端口和非特权端口。
- 通过发送 keepalive 消息把一个空闲连接一直保持打开状态。
- 启用 Nagle 算法 (TCP_NODELAY)。
- 需要指定 IP 地址是 IPv4 或 IPv6。

7.4.4.1 选择远程端口

大部分 SSH 服务器都监听 TCP 端口 22，因此客户端缺省就是连接这个端口。然而，有时用户需要连接到其他端口的 SSH 服务器上。例如，如果你是系统管理员，要测试一个新 SSH 服务器，那么可以在其他端口上运行 SSH 服务器，这样就不会和现有的服务器造成冲突。然后，你的客户端就需要连接到这个端口上。这可以使用客户端的 Port 关键字来设置，后面跟上一个端口号：

```
# SSH1, SSH2, OpenSSH
Port 2035
```

也可以使用 -p 命令行选项，后面跟上一个端口号：

```
# SSH1, SSH2, OpenSSH
$ ssh -p 2035 server.example.com
```

还可以为 *scp* 指定其他的端口号，但是要使用 *-P* 命令行选项来替代 *-p*（注 8）：

```
# SSH1, SSH2, OpenSSH
$ scp -P 2035 myfile server.example.com:
```

在 SSH2 2.1.0 及更新的版本中，可以提供一个端口号作为用户和主机规范的一部分，前面使用一个 # 号。例如，下面这些命令：

```
# 仅对 SSH2
$ ssh2 server.example.com#2035
$ ssh2 smith@server.example.com#2035
$ scp2 smith@server.example.com#2035:myfile localfile
```

每个都创建一个 SSH-2 连接，连往远程端口 2035。（我们很少能看到这种语法的使用，但它的确可以使用。）

在连接到服务器之后，*ssh*就在远程 shell 中设置一个环境变量，用来存放端口信息。对 SSH1 和 OpenSSH 来说，这个变量名为 *SSH_CLIENT*；对 SSH2 来说，这个变量名为 *SSH2_CLIENT*。该变量值为一个字符串，由三个值组成（客户端的 IP 地址、客户端的 TCP 端口以及服务器的 TCP 端口），中间使用空格分隔开。例如，如果你的客户端来自 IP 地址 24.128.23.102 的端口 1016，要连往服务器的端口 22，该值就是：

```
# SSH1, OpenSSH
$ echo $SSH_CLIENT
24.128.23.102 1016 22

# 仅对 SSH2
$ echo $SSH2_CLIENT
24.128.23.102 1016 22
```

这些变量在编写脚本时非常有用。用户可以在 shell 启动文件（例如，*~/.profile* 和 *~/.login*）中对这些变量进行测试，如果变量存在就执行相应的操作。例如：

```
#!/bin/sh
# 测试 SSH_CLIENT 值是否不为零
if [ -n "$SSH_CLIENT" ]
then
# 通过 SSH 登录
```

注 8： *scp* 也有一个 *-p* 选项，与 *rcp* 相应选项的含义相同，都是“保持文件权限”。

```

echo 'Welcome, SSH-1 user!'
# 从 SSH_CLIENT 提取 IP 地址
IP=`echo $SSH_CLIENT | awk '{print $1}'`
# 将其转换为主机名
HOSTNAME=`host $IP | grep Name: | awk '{print $2}'`
echo "I see you are connecting from $HOSTNAME."
else
# 不通过 SSH 登录，而是通过其他方式
echo 'Welcome, O clueless one. Feeling insecure today?'
fi

```

7.4.4.2 强制使用本地非特权端口

SSH 连接在本地要使用特权端口，其端口号小于 1024。[3.4.2.3] 如果需要覆盖这个特性（比如，连接必须要通过一个防火墙，而这个防火墙不允许使用特权源端口），那么就需要使用 *-P* 命令行选项：

```

# SSH1, SSH2, OpenSSH
$ ssh -P server.example.com

```

-P 命令行选项可以让 *ssh* 选择一个本地非特权端口（注 9）。现在让我们分使用和不使用 *-P* 选项两种情况，在远程主机上打印 *SSH_CLIENT* 的值，从而观察一下这个选项是如何工作的。回想一下 *SSH_CLIENT* 中依次包含客户端的 IP 地址、客户端端口号和服务器端口号：

```

# 缺省：绑定至特权端口
$ ssh server.example.com 'echo $SSH_CLIENT'
128.119.240.87 1022 22                                1022 < 1024

# 绑定至非特权端口
$ ssh -P server.example.com 'echo $SSH_CLIENT'
128.119.240.87 36885 22                               36885 >= 1024

```

配置关键字 *UsePrivilegedPort* (SSH1、OpenSSH) 的功能和 *-P* 选项相同，该值可以是 *yes* (使用特权端口，缺省值) 和 *no* (使用非特权端口)：

```

# SSH1, OpenSSH
UsePrivilegedPort no

```

scp 还可以允许把非特权端口和这些配置关键字绑定在一起。然而，*scp* 所使用的命

注 9： 从直觉上讲 *-P* 真的不应该表示“非特权”，但事实的确如此。

命令行选项和 *ssh* 不同。对于 *scp1* 来说，*-L* 选项的意思是绑定一个非特权端口，这和把 *UsePrivilegedPort* 设置成 *no* 效果相同（注 10）：

```
# 仅对 SSH1  
$ scp1 -L myfile server.example.com:
```

scp2 没有对应的命令行选项。

对于可信主机认证来说，必须使用特权端口。换而言之，如果使用 *-P* 命令行选项或把 *UsePrivilegedPort* 设置成 *no*，那么就禁用了 *Rhosts* 和 *RhostsRSA* 认证。
[3.4.2.3]

7.4.4.3 *keepalive* 消息

KeepAlive 关键字规定如果出现连接问题（例如网络超时或服务器崩溃），客户端应该如何处理：

```
# SSH1, SSH2, OpenSSH  
KeepAlive yes
```

如果该值为 *yes*（缺省值）就通知客户端周期性地传送 *keepalive* 消息，并期望周期性地接收到 *keepalive* 消息。如果客户端检测到没有接收到这些消息的响应，那么它就关闭连接。该值为 *no* 说明不使用 *keepalive* 消息。

keepalive 消息代表了一种折衷。如果启用了这种特性，连接一旦出错就会被关闭，即使问题只是暂时的也是如此。然而，这种特性所基于的 TCP 的 *keepalive* 超时时间通常都是几个小时，因此这并不是什么大问题。如果禁用了 *keepalive* 特性，那么即使一个连接发生故障，只要它很久没用了，也会一直保持。

通常对 SSH 服务器来说 *KeepAlive* 更有用，因为如果连接没有响应了，客户端的用户肯定就会注意到。但是，SSH 可以同时连接两个程序，其中一个运行 SSH 客户端，专门等待从另外一方获得输入。在这种情况下，就必须能够检测到连接出现故障的情况。

注 10： *-P* 选项已经用于设置端口号了。对源代码的分析显示 *-L* 的含义可能是“较大的本地端口号”。

KeepAlive 并不是专门用来处理 SSH 段由于防火墙、代理、NAT 和 IP 伪装超时所引起的中断问题的。[5.4.3.4]

7.4.4.4 控制 TCP_NODELAY

TCP/IP 有一种特性称为 Nagle 算法，它是一种优化算法，可以减少只发送少量数据的 TCP 段的数目。[4.1.5.3] 我们可以使用 NoDelay 关键字来通知 SSH2 客户端启用或禁用 Nagle 算法：

```
# 仅对 SSH2  
NoDelay yes
```

其合法值可以是 yes（禁用该算法）和 no（启用该算法，缺省值）。

7.4.4.5 指定使用 IPv4 和 IPv6

OpenSSH 可以强制客户端使用 IP 版本 4 (IPv4) 或版本 6 (IPv6) 地址。IPv4 是现在 Internet 上使用的 IP 的版本号；IPv6 是下一代 IP 版本号，它支持的地址数远远大于 IPv4 可以支持的地址数。要了解关于 IP 地址的更多信息请访问：

<http://www.ipv6.org>

要强制使用 IPv4 地址，可以使用 -4 标志：

```
# 仅对 OpenSSH  
$ ssh -4 server.example.com
```

同理，要使用 IPv6 地址，可以使用 -6 选项：

```
# 仅对 OpenSSH  
$ ssh -6 server.example.com
```

7.4.5 建立连接

理想情况下，SSH 客户端执行的步骤依次为：试图建立一个安全连接，成功建立连接，获得认证证书，然后执行请求的命令，这可能是一个 shell 或其他一些程序。这个过程中的很多步骤都是可以配置的，其中包括：

- 客户端可以尝试建立连接的次数。
- 密码提示符及其行为（只能用于密码认证）。
- 禁止提示任何信息。
- 与终端交互运行远程命令。
- 在后台运行远程命令。
- 如果不能建立安全连接，是否再使用不安全连接。
- 中断和重用 SSH 连接所使用的转义字符。

7.4.5.1 连接尝试次数

如果用户使用的是 SSH1 或 OpenSSH 客户端，而没能建立安全连接，那么它就会反复重试几次。缺省情况下，会连续重试 4 次。可以使用关键字 ConnectionAttempts 来修改这种行为：

```
# SSH1, OpenSSH
ConnectionAttempts 10
```

在本例中，*ssh1* 在最终失败之前可以尝试 10 次，此后它或者退出，或者重新使用不安全的 *rsh* 建立一个连接。我们在讨论关键字 FallBackToRsh 时会重新讨论这个问题。[\[7.4.5.8\]](#)

大部分人都不怎么会使用这个关键字，但是如果网络不可知，这个关键字就很有用了。开个玩笑，你可以把 ConnectionAttempts 设置成 0，从而强制 *ssh1* 立即失败，放弃连接：

```
# SSH1, OpenSSH
$ ssh -o ConnectionAttempts=0 server.example.com
Secure connection to server.example.com refused.
```

7.4.5.2 SSH1 中的密码提示

如果用户在 SSH1 中使用密码认证，那么客户端就会提示输入密码：

```
smith@server.example.com's password:
```

用户可以定制系统提示的内容。为了保障数据的私密性，用户可能不想在屏幕上显示自己的用户名和主机名。配置关键字 `PasswordPromptLogin` 可以控制是否显示用户名，该值可以是 `yes`（缺省值）或 `no`。例如：

```
# 仅对 SSH1  
PasswordPromptLogin no
```

会使提示符不显示用户名：

```
server.example.com password:
```

同理，`PasswordPromptHost` 关键字可以控制是否允许显示主机名，其值也可以是 `yes`（缺省值）或 `no`。下面这行：

```
# 仅对 SSH1  
PasswordPromptHost no
```

可以禁止提示符中显示主机名：

```
smith's password:
```

如果这两个关键字都被设置成 `no`，那么提示符就缩减为：

```
Password:
```

记住，这种情况仅仅适用于密码认证。如果使用公钥认证，口令提示符就完全不同了，它也不是由这几个关键字控制的：

```
Enter passphrase for RSA key 'Dave Smith's Home PC':
```

用户还可以控制在输错密码时可以重试的次数。缺省情况下，系统只会提示输入一次密码，如果输错了密码，客户端就会退出。关键字 `NumberOfPasswordPrompts` 可以把该值修改成 1 到 5 次（注 11）：

```
* SSH1, OpenSSH  
NumberOfPasswordPrompts 3
```

现在客户端就有 3 次机会输入密码了。

注 11：SSH 服务器限制重试次数的上限为 5。

7.4.5.3 SSH2 中的密码提示

SSH2增加了设置密码提示的灵活性。用户可以不使用固定的密码提示字符串，而是使用 PasswordPrompt 关键字设置自己的密码提示，从而代替系统现有的密码提示符：

```
# 仅对 SSH2
PasswordPrompt Enter your password right now, infidel:
```

可以使用符号 %U (远程用户名) 和 %H (远程主机名) 在密码提示符中插入远程用户名或远程主机名。例如，要模仿上面的 SSH1 的密码提示符，可以这样使用：

```
# SSH2 only
PasswordPrompt "%U@%H's password:"
```

也可以任意发挥自己的想像力来设置密码提示：

```
# 仅对 SSH2
PasswordPrompt "Welcome %U! Please enter your %H password:"
```

7.4.5.4 批模式：禁止显示密码提示

在有些情况下，用户并不想让系统提示输入密码或 RSA 口令。例如，如果 *ssh* 是通过一个自动的 shell 脚本调用的，那么就不会有人会用键盘输入密码。这就是 SSH 批模式存在的原因。在批模式中，所有对认证证书进行提示的消息都被禁止显示。关键字 BatchMode 的值可以是 yes (禁止显示提示) 或 no (缺省值，允许显示提示)：

```
# SSH1, SSH2, OpenSSH
BatchMode yes
```

在 *scp* 中，也可以使用 -B 命令行选项启用批模式：

```
* SSH1, SSH2, OpenSSH
$ scp1 -B myfile server.example.com:
```

批模式并不能代替认证。如果认证过程需要密码或口令，而又禁止显示提示，那么就无从输入密码，也就不能登录系统。如果试图登录服务器，客户端就会显示一个错误消息，例如 “*permission denied*”。为了能使批模式正常工作，用户必须保证认证不需要密码或口令，比如，使用可信主机认证或 SSH 代理。[\[11.1\]](#)

7.4.5.5 伪终端分配 (PTY/PTY/PTTY)

Unix的tty(发音为T-T-Y)是用来表示计算机终端的一个软件抽象，它是“teletype”的缩写。tty是 Unix 机器进行交互式会话的一个组成部分，它被分配用来处理键盘输入，以给定的行数和列数显示屏幕输出，以及处理和终端有关的操作。大部分类似终端的连接都不会涉及真正的硬件终端，而是使用一个窗口来处理这种连接，窗口是一个称为伪终端(*pseudo-tty*，或称为*pty*，其发音为P-T-Y)的软件结构。

当客户端请求建立一个SSH连接时，服务器并不需要给这个客户端分配一个pty。当然，如果这个客户端请求的是交互式终端会话(即只是*ssh host*)，那么服务器就要为客户端分配一个伪终端了。但是如果只是要让*ssh*在远程服务器上运行一个简单的命令，例如*ls*:

```
$ ssh remote.server.com /bin/ls
```

就不需要交互式终端会话，只需要把*ls*的结果发送给客户端即可。实际上，缺省情况下*sshd*不会为这种命令分配pty。另一方面，如果试图这样运行诸如文本编辑器Emacs之类的交互式命令，那么就会看到一个错误消息：

```
$ ssh remote.server.com emacs -nw  
emacs: standard input is not a tty
```

这是因为Emacs是专门为终端设置的一个基于屏幕的程序。在这种情况下，可以使用`-t`选项请求SSH分配一个pty：

```
# SSH1, SSH2, OpenSSH  
$ ssh -t server.example.com emacs
```

SSH2还有一个关键字ForcePTYAllocation，其作用与`-t`命令行选项完全相同(注12)。

如果SSH分配了一个pty，它还会在远程shell中自动定义一个环境变量。该变量是SSH_TTY(SSH1、OpenSSH)或SSH2_TTY(SSH2)，其中包含了连接到pty的“从(slave)”方(也就是模拟实际tty的那一方)上的字符设备文件名。我们可以

注12：在SSH1和OpenSSH中，*authorized_keys*中的no-pty选项可以覆盖这里对tty的请求。[8.2.9]

使用几个简单的命令来看一下这样设置的效果。用户可以试着显示远程机器上 `SSH_TTY` 变量的值，如果没有分配 `tty`，那么其结果就为空：

```
$ ssh1 server.example.com 'echo $SSH_TTY'  
无输出
```

如果用户强制分配 `tty`，那么结果就是这个 `tty` 的名字：

```
$ ssh1 -t server.example.com 'echo $SSH_TTY'  
/dev/pts/1
```

有了这个变量的帮助，就可以运行远程主机上使用这个信息的 shell 脚本了。例如，下面这个脚本就会在有可用终端时运行缺省的编辑器：

```
#!/bin/sh  
if [ -n $SSH_TTY -o -n $SSH2_TTY ]; then  
    echo 'Success!'  
    exec $EDITOR  
else  
    echo "Sorry, interactive commands require a tty"  
fi
```

把这个脚本放到远程账号中，将其命名为 `myscript`（或者任何名字都可以），然后运行：

```
$ ssh server.example.com myscript  
Sorry, interactive commands require a tty  
$ ssh -t server.example.com myscript  
Success!  
...Emacs 运行...
```

7.4.5.6 后台运行远程命令

如果想在后台运行一个 SSH 远程命令，那么你可能会被结果吓一跳。在远程命令执行完之后，客户端在显示输出结果之前就会自动挂起：

```
$ ssh server.example.com ls &  
[1] 11910  
$  
... 时间流逝 ...  
[1] + Stopped (SIGTTIN) ssh server.example.com ls &
```

之所以会发生这种情况，是因为 `ssh` 在后台运行时正试图从标准输入读取信息，这样就导致 shell 挂起 `ssh`。要查看 `ssh` 执行的结果，必须把 `ssh` 调到前台：

```
$ fg  
README  
myfile  
myfile2
```

ssh 提供了一个 *-n* 命令行选项来解决这个问题。它把标准输入重定向到 */dev/null*，这样就可以防止 *ssh* 由于输入问题而挂起。现在当远程命令结束时，结果就会立即显示出来：

```
# SSH1, SSH2, OpenSSH  
$ ssh -n server.example.com ls &  
[1] 11912  
$  
... 时间流逝 ...  
README  
myfile  
myfile2
```

SSH2 有一个关键字 *DontReadStdin* 可以实现和 *-n* 选项相同的功能，该值可以是 *yes* 或 *no*（缺省值）：

```
# 仅对 SSH2  
DontReadStdin yes
```

7.4.5.7 后台运行远程命令，第二部分

上一节假设用户不需要输入密码或口令，例如用户正在运行 SSH 代理。如果用户使用 *-n* 或 *DontReadStdin*，而 SSH 客户端需要从这里读取密码或口令，此时会发生什么情况呢？

```
$ ssh -n server.example.com ls &  
$  
Enter passphrase for RSA key 'smith@client':
```

警告：别着急！不要输入口令！因为这个命令是使用 *-n* 选项在后台运行的，提示信息也会在后台显示。如果输入了口令，所输入的内容就到了 shell 中，而不是 *ssh* 提示符中，因此输入的任何内容都会显示出来。

需要这样一种解决方案：它不但可以不用输入，还会把该进程转入后台执行，而且还允许 *ssh* 向用户提示信息。这正是 *-f* 命令行选项的目的所在，它可以通知 *ssh* 按照以下顺序执行操作：

1. 进行认证，包括所有的提示输入密码 / 口令的信息。
2. 让该进程从 `/dev/null` 读取输入信息，就像 `-n` 选项一样。
3. 把进程转入后台执行：不需要输入“&”。

下面是一个这种例子：

```
$ ssh -f server.example.com ls
Enter passphrase for RSA key 'smith@client': *****
$ ...
... 时间流逝 ...
README
myfile
myfile2
```

SSH2 有一个关键字 `GoBackground` 可以实现相同的功能，该值可以为 `yes` 或 `no`（缺省值）：

```
# 仅对 SSH2
GoBackground yes
```

如果在命令行中指定了端口转发，`GoBackground` 和 `-f` 还会对端口转发进行 [9.2.6] 设置。设置过程发生的时机是在认证之后、转入后台执行之前。

7.4.5.8 RSH 的问题

假设远程主机现在没有运行 SSH 服务器，但你正好要使用 SSH 登录。现在会出现什么情况呢？根据客户端配置的设置不同，可能产生三种情况：

情况 1

`ssh` 试图建立 SSH 连接，但是没有成功，然后试图建立不安全的 `rsh` 连接（注 13）。这是缺省操作，大部分用户几乎都会这样做。（“噢，使用 SSH 不能连接服务器。我再试一下 `rsh` 好了。”）连接尝试会显示：

```
$ ssh no-ssh-server.com
Secure connection to no-ssh-server.com on port 22 refused; reverting to
insecure method.
Using rsh. WARNING: Connection will not be encrypted.
```

注 13：只有 `ssh` 编译时用标志 `--with-rsh` 支持了 `rsh` 才行。[4.1.5.12] 否则只可能发生情况 2 中失败后停止的现象。

情况 2

ssh 试图建立 SSH 连接，但是没有成功，然后就不再继续尝试并退出。这非常适合安全性要求很高而不能使用 *rsh* 的环境。

```
$ ssh no-ssh-server.com  
Secure connection to no-ssh-server.com on port 22 refused.
```

情况 3

ssh 根本就不试图建立 SSH 连接，而是直接建立不安全的 *rsh* 连接（注 14）。这非常适合用户早就知道特定的机器没有运行 SSH 服务器但却提供了 *rsh* 的情况。

```
$ ssh no-ssh-server.com  
Using rsh.  WARNING: Connection will not be encrypted.
```

有两个配置关键字可以选择用户想要的行为。（记住，可以为每个想访问的远程主机分别进行配置。）*FallBackToRsh* 负责控制在 SSH 连接失败时的操作：应该尝试 *rsh* 连接还是退出？*FallBackToRsh* 的值可以是 yes（缺省值，尝试 *rsh*）或 no（不尝试 *rsh*）：

```
# SSH1, SSH2, OpenSSH  
FallBackToRsh no
```

关键字 *UseRsh* 用来通知 *ssh* 立即使用 *rsh*，而不用尝试建立 SSH 连接。该值可以为 yes（使用 *rsh*）或 no（缺省值，使用 *ssh*）：

```
# SSH1, SSH2, OpenSSH  
UseRsh yes
```

下面介绍如何构造这三种情况：

情况 1

首先尝试使用 *ssh*，然后使用 *rsh*：

```
# SSH1, SSH2, OpenSSH  
FallBackToRsh yes  
UseRsh no
```

情况 2

只使用 *ssh*：

注 14： 还是只能在 *ssh* 用 --with-rsh 编译过的情况下适用。

```
# SSH1, SSH2, OpenSSH
FallBackToRsh no
UseRsh no
```

情况3

只使用 *rsh*:

```
# SSH1, SSH2, OpenSSH
UseRsh yes
```

使用 *UseRsh* 关键字时一定要慎重，确保在配置文件中把它的影响限制在单个远程主机中，而不会影响到所有的主机。下表给出了一一个例子，它为所有的 SSH 连接都禁用了加密特性：

SSH1, OpenSSH	SSH2
# 不要这样做！会危及安全！	# 不要这样做！会危及安全！
Host *	*
UseRsh yes	UseRsh yes

7.4.5.9 转义字符序列

回想一下 *ssh* 客户端有一个转义字符序列的特性。[2.3.2] 通过输入一个特殊字符（通常是“~”），并在其后紧跟一个上换行或回车字符，就可以向 *ssh* 发送特殊命令：中断连接、挂起连接等等。但是有时候缺省的转义字符也会引起问题。

假设使用 *ssh* 从主机 A 连到主机 B，然后从主机 B 连到主机 C，最后从主机 C 连到主机 D，从而构成一个 SSH 链。（我们分别把各个主机的 shell 提示符表示为 A\$、B\$、C\$ 和 D\$。）

```
A$ ssh B
...
B$ ssh C
...
C$ ssh D
...
D$
```

当登录到主机 D 上时，按下回车键，然后输入 ~^Z (~ 后面跟上 Control-Z) 暂时挂起连接。但是现在已经有三个活动的 *ssh* 连接，应该挂起哪一个呢？实际上会挂起第一个 *ssh*，转义字符会把你带回主机 A 的提示符。那如果想转回到主机 B 或主机 C 的环境中又该如何呢？有两种方法，一种要提前进行设置，而另一种可以立即使用。

如果想提前配置，可以使用配置关键字 `EscapeChar`，并在后面跟上一个字符，从而为每个连接都设置一个转义字符：

```
# SSH1, SSH2, OpenSSH
EscapeChar %
```

也可以使用 `-e` 命令行选项，后面跟上想使用的字符（如果要防止 shell 对其进行扩展时出现错误，可以使用引号把这个字符括起来）：

```
# SSH1, SSH2, OpenSSH
$ ssh -e '%' server.example.com
```

我们再回到刚才那个主机 A 到主机 D 的例子上来，现在想让这个连接链的每个部分都使用不同的转义字符，例如：

```
# SSH1, SSH2, OpenSSH
A$ ssh B
...
B$ ssh -e '$' C
...
C$ ssh -e '%' D
...
D$
```

如果你现在已经登录到主机 D 上，那么 ~ 转义字符仍然可以把你带回主机 A，而 \$ 符号可以把你带回主机 B，% 符号可以把你带回主机 C。使用 `EscapeChar` 关键字也可以达到相同的效果，但这需要在三个主机的配置文件中都提前设置更多内容，如下表所示。

SSH1, OpenSSH	SSH2
# 主机 A 配置文件	# 主机 A 配置文件
Host B	B:
EscapeChar ~	EscapeChar ~
# 主机 B 配置文件	# 主机 B 配置文件
Host C	C:
EscapeChar ^	EscapeChar ^
# 主机 C 配置文件	# 主机 C 配置文件
Host D	D:
EscapeChar %	EscapeChar %

虽然用户通常都不会使用SSH构成一个连接链，但也可能希望要修改转义字符。例如，由于某些原因你的工作可能需要输入很多~字符，那么偶尔就可能会输入一个转义字符序列（例如~.），这样就会中断会话。噢，这可真糟！

第二种方法不需要提前进行设置。回想一下，输入两次转义字符实际上就是要通过SSH连接发送这个字符。^[2.3.2]因此，可以输入两个转义字符来挂起第二个SSH连接，输入三个转义字符就可以挂起第三个SSH连接，依此类推。记住，必须在输入转义字符之前按下回车键。在已经登录到主机D上之后，可以这样返回主机B：按下回车键，然后输入两个~，再按下Control-Z。

7.4.6 代理和SOCKS

SOCKS是各种SSH实现都支持的一种应用层网络代理系统。代理通常都会在应用层提供一种连接两个网络的方法，它不允许在这两个网络之间直接建立网络层的连接。图7-3给出了一个典型的SOCKS安装情况。

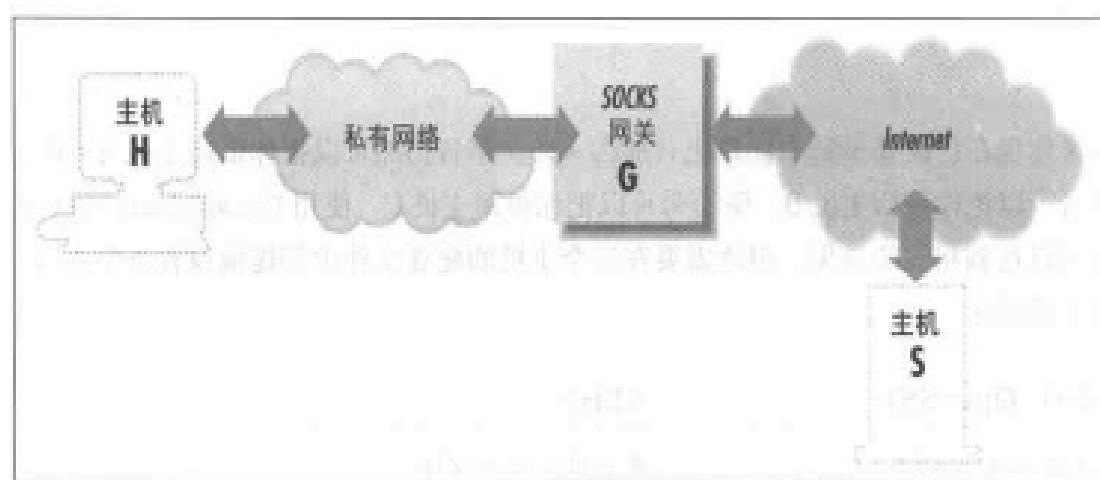


图7-3：典型的SOCKS安装情况

该图中给出了一个专用网络和Internet。网关主机同时要连接到这两个网络上，但是它并不提供路由器的功能；在这两个网络之间不会直接建立IP连接。如果一个在H上运行的程序想要和S上的服务器建立一条TCP连接，它需要首先连接到在G上运行的SOCKS服务器上。H使用SOCKS协议请求建立一条到S的连接。SOCKS服务器根据H的请求建立一条从G到S的连接，然后在H和S之间反复传递数据。

通常这种应用层代理都缺乏透明性：只有那些在编写时就支持这种特殊代理模式的程序才具有网络访问权。但是，SOCKS 并不是专门用于诸如 HTTP 或 SMTP 之类的高层协议。它提供了非常通用的服务：建立 TCP 连接、执行 ping 主机、traceroute 命令等等。很多这种服务都可以使用现在处于应用程序和网络服务边缘的库。由于现代计算机系统都使用动态链接库技术，因此我们可以把这种对套接字不透明的程序改进成对套接字透明的，这可以通过替换适当的库来实现，SSH 就是如此。

SOCKS 有两个版本，SOCKS4 和 SOCKS5。二者之间主要的区别是 SOCKS5 要执行用户认证，而 SOCKS4 不会。使用 SOCKS5，就可以要求客户端在访问网络服务之前提供用户名和密码（或其他认证模式）。

7.4.6.1 SSH1 中的 SOCKS

以下的介绍假设用户在安装 SSH 时就使用 NEC 的 socks5 包启用了 SOCKS 的支持。[\[4.1.5.8\]](#)如果你使用了其他包，那么 SOCKS 专用的配置细节就可能和我们介绍的稍有不同。

顺便说一下，socks5 这个名字易引起误解。虽然 NEC 软件名字为“sock5”，但是它实际上既实现了 SOCKS4 协议，又实现了 SOCKS5 协议。我们在此处使用小写的“socks5”来代表 NEC 的实现产品。

一旦安装好对 SOCKS 敏感的 *ssh*，就可以使用环境变量对和 SOCKS 有关的行为进行控制。缺省情况下，*ssh* 根本不会使用 SOCKS。如果把 SOCKS_SERVER 设置成“*socks.shoes.com*”，那么 *ssh* 就使用 *socks.shoes.com* 的 SOCKS 网关连接到本地主机子网（本地主机子网是在相应的网络接口中使用网络掩码进行设置的）之外的 SSH 服务器上。如果想让 *ssh* 在所有的连接中（包括本地子网中）都使用 SOCKS，就可以设置 SOCKS5_NONETMASKCHECK 变量。如果 SOCKS 网关需要用户名/密码认证，那么就需要把 SOCKS5_USER 和 SOCKS5_PASSWD 变量设置成自己的用户名和密码。适当设置环境变量，我们也可以使用 SOCKS 特有的调试输出：

```
#!/bin/csh
setenv SOCKS5_DEBUG 3
setenv SOCKS5_LOG_STDERR
```

文档中说明调试等级最大可以是 3，但实际上源代码中使用的调试等级更高，这在用来理解一些问题时非常重要。如果你觉得现在输出的信息太少，就可以尝试修改这个值。

7.4.6.2 SSH2 中的 SOCKS

SSH2 只能支持 SOCKS4，而且这种支持是集成在 SSH2 源代码中的，因此无需另外安装一个 SOCKS 包，也不需要在编译时专门启用 SOCKS 特性；它在程序中总是可用的。

SSH2 SOCKS 特性由一个参数进行控制，它是在 `SocksServer` 配置关键字或 `SSH_SOCKS_SERVER` 服务器变量中设置的。如果二者同时出现，那么配置关键字的设置会覆盖环境变量中的设置。

`SocksServer` 关键字是一个字符串，格式如下：

```
socks://[user]@gateway[:port]/[net1/mask1,net2/mask2,...]
```

此处，`gateway` 是运行 SOCKS 服务器的机器，`user` 是用户向 SOCKS 提供的用来标识自己的用户名，`port` 是 SOCKS 服务器使用的 TCP 端口（缺省值是 1080）。`net/mask` 说明被当作本地子网的网络；也就是说，`ssh2` 只会在连接到给定的网络范围之外时才使用 SOCKS。`mask` 是以位数的形式给出的，而没有使用显式的掩码，也就是说，它不会使用 192.168.10.0/255.255.255.0 的形式，而是使用 192.168.10.0/24 的形式。

方括号中的内容是可选的。因此 `SSH_SOCKS_SERVER` 的值的最简单形式可以为：

```
socks://laces.shoes.net
```

设置了这个值，`ssh2` 就会为所有的连接都使用 SOCKS。这些连接都会链接到 `laces.shoes.net` 端口 1080 的 SOCKS 服务器上，且不需要提供用户名。你可能奇怪为什么此处有用户名却没有密码字段呢？回想一下 SOCKS4 并不支持用户认证。用户名只是参考性质的；SOCKS 服务器无法对你所宣称的身份进行验证。

可能你永远都不会使用如此简单的 `SSH_SOCKS_SERVER` 设置：所有的 `ssh2` 连接都使用 SOCKS 服务器，即使这些连接最终回到本机或同一网络中的机器上也是如此。

此。更好的设置是只为那些连到你所在的网关之外的主机上的连接使用SOCKS。这里有一个更复杂的例子：

```
socks://dan@laces.shoes.net:4321/127.0.0.0/8,192.168.10.0/24
```

设置了该值，*ssh2*连接就可以通过回环地址（127.0.0.1）直接连接自己上面，或连到C类网络192.168.10.0上。其他连接都使用SOCKS，并指定用户名是“dan”，而且是在端口4321上查找SOCKS服务器。

7.4.6.3 OpenSSH 中的 SOCKS

OpenSSH并没有显式支持SOCKS。但是，我们已经发现，OpenSSH可以和NEC的SOCKS5包中提供的*runsocks*程序很好地一起工作。*runsocks*是一个封装程序，它对动态链接的次序重新进行组织，这样诸如bind、connect之类的套接字程序在运行时就可以使用SOCKS化了的程序来代替。在Linux系统中，我们发现如果像前面介绍的方法正确配置*socks5*环境变量，然后运行：

```
* runsocks ssh ...
```

就可以让OpenSSH完美无暇地使用我们的SOCKS服务器。但是要注意：要想正常工作，OpenSSH客户端程序一定不能setuid。由于很显然的安全原因，如果OpenSSH客户端可执行程序设置了setuid，那么共享库装载程序会忽略*runsocks*的把戏。记住setuid最好只用于可信主机认证。[3.4.2.3]

曾经有一段时间，在OpenSSH源代码中包含了对SOCKS的支持。但是后来将其删除了，并建议使用ProxyCommand特性来替代。其旨在让一个小程序从命令行中接收主机名和端口号，通过SOCKS连接到这个套接字上，然后将其用作一个管道，并在TCP连接的双方及其标准输入输出之间反复传递数据。如果这个程序名为*ssh-proxy*，就可以在OpenSSH中这样使用：

```
* ssh -o 'ProxyCommand ssh-proxy %h %p' ...
```

这在使用SSH1 RhostsRSA认证时仍然不能工作，除非*ssh-proxy*被setuid成root，而且其实现可以使用特权源端口。它自己并不会和SSH2基于主机的认证冲突，但是还有另外一个问题。[7.4.6.4]

我们觉得其他地方一定有这种 SOCKS 代理构件，但现在还没有找到。不能使用 SOCKS 化了的 telnet 来使用 socks5，因为它是不透明的；二进制 SSH 协议数据流中的字节都会被解释成 Telnet 的转义字符序列。作者并没有证实这种概念，但却给出了一个 netcat 的拷贝 (<http://www.I0phT.com/~w31d/netcat/>)，并通过将其连接到 socks5 库上而对其进行 SOCKS 化。netcat 的可执行程序名为 nc；使用我们修改过的版本就可以通过我们的 SOCKS 网关来发送 SSH 连接：

```
% ssh -o 'ProxyCommand nc %h %p' ...
```

可能 OpenSSH 的开发人员以后会在 OpenSSH 中加入这种工具。

7.4.6.4 其他 SOCKS 问题

记住，通过 SOCKS 的连接都是来自 SOCKS 网关，而不是来自原来的客户端主机。这样就导致使用可信主机进行认证时出现了一个问题。sshd 使用连接的源 IP 地址来查找客户端的主机密钥，因此 RhostsRSA 就会失效：它希望接收的是网关的主机密钥，而不是实际客户端的主机密钥。只能这样解决这个问题：通过为所有的客户端都分配相同的主机密钥，并将其在 SSH 服务器的已知名数据库中和网关关联在一起。这可不是什么好主意，因为一旦这个密钥被偷窃，那么偷窃者就可以伪装成任何客户端上的任何用户，而不仅仅是一个用户了。但是这在某些情况下是可以接受的。

在 SSH2 中就不会出现这种问题；SSH-2 协议进行基于主机的认证是和客户端的主机地址无关的。但是，SSH2 在实现这个问题时所使用的方法太过陈旧，它依然不能使用 SOCKS。用户可能会想到自己可以在源代码中禁用地址/主机名的检测。千万不要这样做！否则问题就更大了。[3.5.1.6]

7.4.7 转发

端口转发和 X 转发的内容将在第九章中介绍，代理转发的内容在第六章中已介绍。我们在这里提及这些内容只是为了完整性的需要，因为转发也可以在客户端的配置文件和命令行中进行控制。

7.4.8 加密算法

在建立一个连接时，SSH客户端和服务器要对加密算法进行协商。服务器说：“客户端您好，这是我能支持的加密算法。”客户端回答到：“服务器您好，我选择使用这个加密算法，谢谢。”通常，它们会达成一致，连接继续进行。如果它们不能对加密算法达成一致，那么连接就失败了。

大部分用户都会让客户端和服务器自行解决这些问题。但是如果你喜欢，也可以要求客户端在和服务器进行协商时请求特定的加密算法。在SSH1和OpenSSH中，这是通过使用Cipher关键字并在后面跟上你选择的加密算法来实现的：

```
# SSH1, OpenSSH
Cipher blowfish
```

也可以使用-c命令行选项：

```
# SSH1, SSH2, OpenSSH
$ ssh -c blowfish server.example.com
$ scp -c blowfish myfile server.example.com:
```

SSH2几乎完全相同，惟一不同的是关键字变成了Ciphers（注意最后有个“s”），后面可以跟上一个或多个加密算法，中间使用逗号分隔开，说明这些算法都可以使用：

```
# SSH2, OpenSSH/2
Ciphers blowfish,3des
```

SSH2也支持上面的-c命令行选项，而且可以出现多次，让你指定多个可以使用的算法：

```
# 仅对SSH2
$ ssh2 -c blowfish -c 3des -c idea server.example.com
$ scp2 -c blowfish -c 3des -c idea myfile server.example.com:
```

OpenSSH/2允许使用一个-c命令行选项来指定多个算法，中间使用逗号分隔开，从而达到相同的结果：

```
# 仅对OpenSSH/2
$ ssh -c 3des-cbc,blowfish-cbc,arcfour server.example.com
```

我们可以为客户端指定服务器可以使用的所有加密算法。[5.4.5] 检查最新的SSH文档就可以确定现在究竟支持哪些算法。

7.4.8.1 MAC 算法

`-m` 命令行选项可以让用户选择 `ssh2` 使用的完整性检测算法，称为 MAC (Message Authentication Code，消息认证代码): [3.9.3]

```
# 仅对 SSH2
$ ssh2 -m hmac-sha1 server.example.com
```

可以在命令行中指定多个算法，每个算法前面都使用一个 `-m` 选项:

```
# 仅对 SSH2
$ ssh2 -m hmac-sha1 -m another-one server.example.com
```

然后 SSH2 服务器就会从其中选择一个来使用。

7.4.9 重新生成会话密钥

`RekeyIntervalSeconds` 指定了 SSH2 客户端多久和服务器执行一次密钥交换，从而更新会话数据加密和完整性检测所使用的密钥。缺省值是 3600 秒 (一小时)，该值为 0 表示禁止重新生成密钥 (注 15):

```
# 仅对 SSH2
RekeyIntervalSeconds 7200
```

7.4.10 认证

在一个典型的 SSH 设置中，客户端都试图首先使用最严格的方法进行认证。如果一种特定的方法失效，或者没有建立起连接，就继续尝试下一种方法，依此类推。这种缺省的行为在大部分情况下都可以很好地工作。

然而，如果需要，客户端也可以请求使用特定的认证方法。例如，用户可能会想只使用公钥认证，如果失败，就不应该再尝试其他方法了。

注 15：注意，到本书（英文版）出版时，如果你打算用 SSH2 的客户端在 OpenSSH 服务器上使用这一选项，就必须关闭重新生成会话密钥的机制，因为后者尚不支持重新生成会话密钥。一旦生成密钥的周期一到，连接就会出错，然后死掉。不过这个功能很快就会实现了。

7.4.10.1 请求认证技术

SSH1 和 OpenSSH 客户端都可以使用关键字来指定特定的认证方法。语法和 */etc/sshd_config* 中服务器的语法相同。[\[5.5.1\]](#) 可以使用的关键字有：

```
PasswordAuthentication  
RhostsAuthentication  
RhostsRSAAuthentication  
RSAAuthentication  
TISAuthentication  
KerberosAuthentication
```

(最后两个关键字分别要求在编译时包含 TIS 或 Kerberos 支持。) 这些关键字的值都可以是 yes 或 no。

对 SSH2 来说，*AllowedAuthentications* 关键字可以选择一种或多种认证技术。同样，该关键字和在 SSH2 服务器上的用法相同。[\[5.5.1\]](#)

OpenSSH 可以使用 SSH1 中除 *TISAuthentication* 之外的其他关键字，它还增加了一个关键字 *SkeyAuthentication*，用于一次性密码认证。[\[5.5.1.10\]](#)

7.4.10.2 服务器决定使用的认证方法

当客户端指定认证方法时，这仅仅是一个请求，而未必一定会使用这种方法进行认证。例如，配置：

```
PasswordAuthentication yes
```

通知服务器，你（也就是客户端）同意使用密码认证。这并不是说一定要使用密码进行认证，而是说如果服务器同意，就可以使用密码进行认证而已。使用哪种方法进行认证最终由服务器决定，它可以选择使用其他方法进行认证。

如果客户端想要强制使用一种认证方法，那么它必须告诉服务器自己可以使用这种方法，而且只能使用这种方法。要实现这种功能，客户端就不能指定其他认证方法。例如，要在 SSH1 或 OpenSSH 中强制使用密码认证：

```
# SSH1, OpenSSH
```

```
# 如果服务器支持，一定会使用密码认证
PasswordAuthentication yes
RSAAuthentication no
RhostsRSAAuthentication no
RhostsAuthentication no
KerberosAuthentication no
# …… 用值“no”添加其他认证方法
```

但是，如果服务器不支持密码认证，那么这次连接尝试就会失败。

SSH2的实现更容易：它可以使用 AllowedAuthentications关键字，其语法和意义都与服务器中同名的关键字相同：

```
# 仅对 SSH2
AllowedAuthentications password
```

7.4.10.3 认证成功的检测

SSH2提供了两个关键字用来报告认证是否成功：AuthenticationSuccessMsg和AuthenticationNotify。这两个关键字都可以让SSH2客户端在尝试认证之后打印一条消息。

AuthenticationSuccessMsg控制在认证之后是否要在标准错误设备上显示“Authentication successful”。该值可以是yes（缺省值，显示成功消息）或no：

```
$ ssh2 server.example.com
Authentication successful.
Last login: Sat Jun 24 2000 14:53:28 -0400
...
$ ssh2 -p221 -o 'AuthenticationSuccessMsg no' server.example.com
Last login: Sat Jun 24 2000 14:53:28 -0400
...
```

AuthenticationNotify在文档中没有给出说明，它可以让ssh2打印一条不同的消息，这次是在标准输出设备上显示。如果认证成功，那么显示的消息就是“AUTHENTICATED YES”，否则就是“AUTHENTICATED NO”。该值可以是yes（显示消息）或no（缺省值）：

```
$ ssh2 -q -o 'AuthenticationNotify yes' server.example.com
AUTHENTICATED YES
Last login: Sat Jun 24 2000 14:53:35 -0400
...
```

这两个关键字行为的差异如下：

- `AuthenticationSuccessMsg` 把消息写往 `stderr`; 而 `AuthenticationNotify` 把消息写往 `stdout`。
- `-q` 命令行选项[7.4.15] 可以强制 `AuthenticationSuccessMsg` 的设置无效, 禁止输出信息; 而对 `AuthenticationNotify` 无效。这使得 `AuthenticationNotify` 更适合在脚本中使用 (例如, 要确定认证是否成功)。注意 `exit` 被当作一个远程命令执行, 因此 `shell` 就会立即结束:

```
#!/bin/csh
# Get the AUTHENTICATION line
set line = `ssh2 -q -o 'AuthenticationNotify yes' server.example.com
exit`
# 获取第二个词
set result = `echo $line | awk '{print $2}'` 
if ( $result == "YES" ) then
    ...

```

实际上, 当 `scp2` 和 `sftp` 在后台运行 `ssh2` 连接到远程主机并进行文件传输时, 就是这样使用 `AuthenticationNotify` 的。等到出现 “**AUTHENTICATED YES**” 消息时它们就知道连接已经成功, 现在可以和 `sftp-server` 进行对话了。

`AuthenticationSuccessMsg` 提供了另外一种安全特性: 它可以确保已经执行了认证。假设你调用 `ssh2` 并被提示输入口令:

```
$ ssh2 server.example.com
Passphrase for key "mykey": *****
```

然后大大出乎意料, 又出现了第二个口令提示符:

```
Passphrase for key "mykey":
```

此时就可以判断出第一次输入的口令有误, 现在再次输入口令。但是如果第二个提示符不是来自 `ssh2` 客户端, 而是来自已经被攻击的服务器, 那该怎么办呢? 你的口令刚才就被偷走了! 为了防止这种可能出现的口令窃取的情况, `ssh2` 在认证之后会打印 “**Authentication successful**”, 因此前面一个例子的情况就变成:

```
$ ssh2 server.example.com
Passphrase for key "mykey": *****
Authentication successful.
Passphrase for key "mykey":
```

现在第二个口令提示很明显就是在诈骗口令。

7.4.11 数据压缩

SSH连接可以进行压缩。也就是说，通过SSH连接发送的数据在加密和发送之前可以自动进行压缩，在接收和解密时可以自动解压缩。如果使用现代快速处理器运行SSH，那么压缩功能通常都会给用户带来很大的帮助。我们进行了一个非正式的测试，在一个由10台工作站组成的以太网上的两台SUN SPARC之间，分别使用压缩方式和非压缩方式在客户端和服务器之间传递12MB的数据。启用了压缩（后面会对其进行解释）时传输时间会减半。

要为一个连接启用压缩功能，可以使用命令行选项来指定。不幸的是，各种实现软件的语法都不相同。对于SSH1和OpenSSH来说，缺省情况下是禁用压缩功能的，我们可以使用-C命令行选项来启用压缩功能：

```
# SSH1, OpenSSH: 启用压缩  
$ ssh1 -C server.example.com  
$ scp1 -C myfile server.example.com:
```

但是对SSH2来说，-C选项的意思刚好相反，是要禁用压缩功能：

```
# 仅对SSH2: 禁用压缩  
$ ssh2 -C server.example.com
```

它使用+C选项来启用压缩功能：

```
# 仅对SSH2: 启用压缩  
$ ssh2 +C server.example.com
```

(scp2没有压缩选项。)要对所有的会话都启用或禁用压缩功能，可以使用Compression关键字来设置，该值可以是yes或no(缺省值)：

```
# SSH1, SSH2, OpenSSH  
Compression yes
```

SSH1和OpenSSH还可以发送一个整数压缩级别来指定数据应该压缩的程度。该值越大意味着压缩比越高，但是性能越差。压缩级别可以是从0~9(包括0和9)的值，缺省值是6(注16)。CompressionLevel关键字可以修改压缩级别的设置：

注16：SSH的压缩功能来自GNU Zip，也称为gzip，这是Unix世界中很流行的压缩工具。这里的9个压缩级别对应了gzip支持的9种压缩方法。

```
# SSH1, OpenSSH
CompressionLevel 2
```

修改 CompressionLevel 设置会对系统的性能造成巨大的影响。我们前面对 12MB 数据进行测试使用的是缺省压缩级别 6，需时 42 秒。使用各种压缩级别，所需要的时间从 25 秒到 2 分钟不等（请参看表 7-2）。如果用户所使用的处理器和网络连接都非常快，那么 CompressionLevel 1 的性能可能最好。你可以做个实验，看一下该值究竟设置成多少才可以达到最好的性能。

表 7-2：压缩和 CompressionLevel 的影响

压缩级别	发送的字节	耗时 (秒)	数据减小的百分比 (%)	时间减少的百分比 (%)
无	12112880	55	0	0
1	2116435	25	82.5	55
2	2091292	25	82.5	55
3	2079467	27	82.8	51
4	1881366	33	84.4	40
5	1833850	36	84.8	35
6	1824180	42	84.9	24
7	1785725	48	85.2	13
8	1756048	102	85.5	-46
9	1755636	118	85.5	-53

7.4.12 程序的位置

辅助程序 *ssh-signer2* 通常位于 SSH2 的安装目录中，其中还有其他一些 SSH2 的二进制程序。^[3.5.2.3] 用户可以使用关键字 *SshSignerPath*（文档中并没有对其进行说明）来修改这个位置：

```
# 仅对 SSH2
SshSignerPath /usr/alternative/bin/ssh-signer2
```

如果要使用这个关键字，就一定要将其设置成该程序的完整路径。如果在这里使用相对路径，那么就只有那些把 *ssh-signer2* 放到自己的搜索路径中的用户才可以正常使用基于主机的认证，如果路径中找不到 *ssh-signer2*，那么 *cron* 任务也会失败。

7.4.13 子系统

子系统是SSH2服务器支持的一种预定义的命令。^[5.7] 每个安装好的服务器都可以实现不同的子系统，因此要使用系统管理员的身份才可以检查服务器机器上有多少子系统（注17）。

*ssh2*的`-s`选项在发布时也没有文档对其进行介绍，它可以调用远程机器上的一个子系统。例如，如果在*server.example.com*上运行的SSH2服务器定义了一个“backups”子系统，那么就可以这样运行：

```
$ ssh2 -s backups server.example.com
```

7.4.14 SSH1/SSH2 兼容性

SSH2有很多与SSH1的兼容性有关的关键字。如果启用了兼容性，那么在客户端请求*ssh2*连接到SSH-1服务器上时，它就会调用*ssh1*（假设系统中可以使用*ssh1*）。

关键字*Ssh1Compatibility*可以启用SSH1兼容性，该值可以为*yes*或*no*。如果在编译时就启用了SSH1兼容性的支持，那么缺省值是*yes*；否则就是*no*：

```
# 仅对SSH2
Ssh1Compatibility yes
```

关键字*Ssh1Path*负责设置*ssh1*可执行文件的路径，其缺省值是在编译时配置中设置的：

```
# 仅对SSH2
Ssh1Path /usr/local/bin/ssh1
```

如果想让SSH2代理存储并读取SSH1密钥，就可以使用关键字*Ssh1AgentCompatibility*启用代理兼容性：^[6.3.2.4]

```
# 仅对SSH2
Ssh1AgentCompatibility yes
```

最后，如果使用了`-l`命令行选项，那么*scp2*就会调用*scp1*：

注17：或者自己检查一下服务器配置文件*/etc/ssh2/sshd2_config*中以*subsystem*开头的行。

```
# 仅对 SSH2  
scp2 -l myfile server.example.com:
```

在这种情况下，*scp2 -l* 简单地调用 *scp1*，并向其传递所有的参数（当然不包括 *-l* 选项）。我们可能不会经常使用这个选项：如果系统中可以使用 *scp1*，那为什么不直接使用 *scp1* 呢？但是如果需要，这个选项还是可以使用的。

7.4.15 日志和调试

在本章一开始，我们就介绍了 *-v* 命令行选项，它可以让客户端打印调试消息。*ssh* 和 *scp* 都可以使用详细模式：

```
# SSH1, OpenSSH  
$ ssh -v server.example.com  
SSH Version 1.2.27 [sparc-sun-solaris2.5.1], protocol version 1.5.  
client: Connecting to server.example.com [128.9.176.249] port 22.  
client: Connection established.  
...
```

在 SSH2 中，也可以使用 *VerboseMode* 关键字来启用详细模式（真奇怪！）：

```
# 仅对 SSH2  
VerboseMode yes
```

如果碰到 SSH 出现问题或一些奇怪的现象，那么就应该首先使用详细模式来看一下哪里出了问题。

SSH2 的调试消息有很多个级别；详细模式对应级别 2。可以使用 *-d* 命令行选项来指定其他级别，后面可以跟上一个 0~99 的整数：

\$ ssh2 -d0	无调试消息
\$ ssh2 -d1	有一点调试消息
\$ ssh2 -d2	同 -v
\$ ssh2 -d3	稍详细一点
\$ ssh2 -d#	等等……

OpenSSH 中类似的特性是由 *LogLevel* 关键字来指定的，它可以使用六种级别作为参数：QUIET、FATAL、ERROR、INFO、VERBOSE 和 DEBUG（按照详细程度递增的顺序）。因此：

```
# OpenSSH
$ ssh -o LogLevel=DEBUG
```

就相当于 `ssh -v`。

`-d` 选项还可以在服务器调试中使用相同的基于模块的语法: [5.8.2.2]

```
$ ssh2 -d Ssh2AuthPasswdServer=2 server.example.com
```

`scp2` 也支持这个调试级别, 但是它使用 `-D` 选项, 而不是 `-d` 选项, 因为 `-d` 早已被用来表示其他意思了:

```
$ scp2 -D Ssh2AuthPasswdServer=2 myfile server.example.com
```

要禁止输出所有的调试消息, 可以使用 `-q` 命令行选项:

```
# SSH1, SSH2, OpenSSH
$ ssh -q server.example.com

# 仅对 SSH2
$ scp2 -q myfile server.example.com:
```

也可以使用 `QuietMode` 关键字:

```
# 仅对 SSH2
QuietMode yes
```

最后, 要显示程序的版本信息, 可以使用 `-V`:

```
# SSH1, SSH2, OpenSSH
$ ssh -V

# 仅对 SSH2
$ scp2 -V
```

7.4.16 随机数种子

SSH2 可以让用户修改随机数种子文件的位置, 缺省情况下该值是 `~/.ssh2/random_seed`: [5.4.1.2]

```
# 仅对 SSH2
RandomSeedFile /u/smith/.ssh2/new_seed
```

7.5 使用 scp 安全拷贝文件

安全拷贝程序 *scp* 所使用的客户端配置文件中的关键字和 *ssh* 相同。另外，*scp* 还提供了其他一些特性和选项，本节将介绍这些内容。

7.5.1 完整的语法

到目前为止，我们只介绍了 *scp* 的基本语法：[2.2.1]

```
scp name-of-source name-of-destination
```

命令行中的这两个名字（或路径）使用以下规则来表示文件或路径（这和 Unix 的 *cp* 或 *rcp* 完全一致）：

- 如果 *name-of-source* 是一个文件，那么 *name-of-destination* 可以是一个文件（可以是已经存在的文件，也可以是不存在的文件）或者一个目录（目录必须存在）。换而言之，一个文件可以被拷贝成另一个文件，也可以被拷贝到一个目录中。
- 如果 *name-of-source* 是两个或多个文件，或者是一个或多个目录，或者既包含文件又包含目录，那么 *name-of-destination* 必须是已经存在的一个目录，用来存放所拷贝的内容（注 18）。换而言之，多个文件和目录可以被拷贝到一个目录中。

name-of-source 和 *name-of-destination* 从左到右格式如下：

1. 包含文件或目录的账号的用户名，后面跟上一个 @。这部分内容是可选的，如果该值被省略了，系统就默认用户名是调用 *scp* 的用户的用户名。
2. 包含文件或目录的主机的主机名，后面跟上一个冒号 (:)。如果给出了路径而没有给出用户名，那么这部分内容就是可选的；如果该值被省略了，那么缺省值就是 *localhost*。SSH2 允许 SSH 连接使用其他 TCP 端口号，可以这样表示：在主机名和冒号之间插入一个 # 号，然后跟上一个端口号。

注 18： 我们说“必须”，但从技术上讲有些情况下可以指定一个文件作为目的。不过这样做的结果可能不是你想要的。因为把多个文件拷贝到一个单一的文件上，其中的每一个都会被后一个覆盖。

3. 文件或目录所在的路径。(如果给出了主机名,那么这部分内容就是可选的。)

相对路径都假设是相对缺省目录的路径:对于本地机器上的路径来说是当前目录,对于远程机器上的路径来说是用户的主目录。如果这部分内容被省略了,那么系统就假设使用缺省路径。

虽然这三部分内容都是可选的,但是不能同时全都省略而使用空字符串。主机名(2)或目录路径(3)至少要有一个必须出现。例如:

MyFile

localhost 上的 *./MyFile* 文件

MyDirectory

localhost 上的 *./MyDirectory* 目录

. *localhost* 上的当前路径

server.example.com:

server.example.com 上的用户主目录

server.example.com

名为 “*server.example.com*” 的本地文件 (呀,是不是忘记加上冒号了?后面加冒号是一个常见的错误)

server.example.com:MyFile

server.example.com 上远程用户主目录下的 *MyFile* 文件

bob@server.example.com:

server.example.com 上的 *~bob* 目录

bob@server.example.com

名为 “*bob@server.example.com*” 的本地文件 (呀,又忘记冒号了)

bob@server.example.com:MyFile

server.example.com 上的 *~bob/MyFile* 文件

server.example.com:dir/MyFile

server.example.com 上远程用户主目录下的 *dir/MyFile* 文件

server.example.com:/dir/MyFile

server.example.com 上的 */dir/MyFile* 文件 (注意是绝对路径)

bob@server.example.com:dir/MyFile

server.example.com 上的 *~bob/dir/MyFile* 文件

bob@server.example.com:/dir/MyFile

server.example.com 上的 */dir/MyFile* 文件（虽然认证用户使用的是 *bob*, 但是路径是绝对路径）

server.example.com#2000:

server.example.com 上的远程用户主目录，通过 TCP 端口 2000 进行连接（只能用于 SSH2）

下面是几个完整的例子：

\$ scp myfile myfile2	本地拷贝，类似于 cp
\$ scp myfile bob@host1:	把 ./myfile 拷贝到 host1 的 ~bob 目录中
\$ scp bob@host1:myfile .	把 host1 上的 ~bob/myfile 拷贝成 ./myfile
\$ scp host1:file1 host2:file2	把 host1 上的 file1 拷贝成 host2 上的 file2
\$ scp bob@host1:file1 jen@host2:file2	和上例相同，但是要从 bob 的账号中拷贝到 jen 的账号中

表 7-3 对 *scp* 路径的语法进行了总结。

表 7-3: *scp* 路径定义

字段	其他语法	可选否？	对本地主机的缺省值	对远程主机的缺省值
用户名	后面跟上一个 @	可选	调用 <i>scp</i> 的用户的用户名	调用 <i>scp</i> 的用户的用户名
主机名	后面跟上一个 :	只有在省略了用户名且给出了路径时才是可选的	无，文件是在本地进行访问的	N/A
端口号 ^a	前面有一个 #	可选	22	22
路径	N/A	只有在给出了主机名时才是可选的	当前（调用 <i>scp</i> 时所在的）目录	远程主机上的用户主目录

a. 只能用于 SSH2。

7.5.2 通配符的处理

SSH1 和 OpenSSH 的 *scp* 没有为在文件名中使用通配符提供特殊的 support。用户可以简单地让 shell 对通配符进行扩展：

```
$ scp *.txt server.example.com:
```

要注意远程文件定义中所使用的通配符，因为这些通配符是在本地主机上进行扩展，而不是在远程主机上进行扩展。例如，这样拷贝文件就会失败：

```
$ scp1 server.example.com:*.txt .           糟糕的办法!
```

Unix shell 试图在调用 *scp1* 之前对通配符进行扩展，但是目录中没有可以和“server.example.com:*.txt”匹配的文件名。C shell 及由此而派生的 shell 都会报告“无匹配项”，不能执行这个 *scp1* 命令。而 Bourne 风格的 shell 发现在当前目录中没有匹配项之后，不会对这个通配符进行扩展，而是直接将其传递给 *scp1*，拷贝操作就会如我们所愿成功了。但是我们不应该依赖于这种巧合的结果，而应该使用转义字符，这样 shell 就可以忽略通配符，直接传递给 *scp1*：

```
$ scp1 server.example.com:\*.txt .
```

scp2 完成 shell 通配符扩展之后，还要执行自己的正则表达式匹配操作。SSH2 使用的 *sshregex* 手册页（请参看附录一）就介绍了所支持的操作符。即便如此，如果不希望让本地 shell 对某些通配符进行扩展，也应该在这些通配符之前加上转义字符。

7.5.3 递归拷贝目录

有时用户可能希望不仅仅拷贝一个文件，而是拷贝一个完整的目录。在这种情况下，可以使用 *-r* 选项，*-r* 表示递归。如果用户熟悉 *rcp*，就会知道 *rcp* 的 *-r* 选项的作用也是相同的。

例如，要把 */usr/local/bin* 目录和其中的文件以及子目录安全拷贝到另一台机器上：

```
# SSH1, SSH2, OpenSSH
$ scp -r /usr/local/bin server.example.com:
```

如果在拷贝目录时忘记了加上 *-r* 选项，那么 *scp* 就会抱怨：

```
$ scp /usr/local/bin server.example.com:  
/usr/local/bin: not a regular file
```

虽然 *scp* 可以拷贝目录，但这并不是最好的办法。如果目录中包含硬链接或软链接，这些内容就不能复制。链接是作为普通文件（链接目标）拷贝的，更糟糕的是，目录的嵌套导致了 *scp1* 的递归的不确定性。（*scp2* 检测到符号链接之后会拷贝链接目标。）其他特殊文件（例如命名管道）也不能正确拷贝（注 19）。一种较好的解决方法是使用 *tar* (*tar* 可以正确处理这些特殊文件) 对目录进行打包，并使用一个管道通过 SSH 发送给远程主机，并在远程主机解包：

```
$ tar cf - /usr/local/bin | ssh server.example.com tar xf -
```

7.5.4 保持权限

在使用 *scp* 拷贝文件时，目标文件都是以特定的文件属性创建的。缺省情况下，文件权限要遵从目标主机上 *umask* 的设置，文件的修改时间和最后访问时间就是拷贝文件的时间。当然，也可以告诉 *scp* 复制源文件的权限和时间戳。*-p* 选项就可以完成这个功能：

```
# SSH1, SSH2, OpenSSH  
$ scp -p myfile server.example.com:
```

例如，如果用户想把自己的整个主目录传送到一台远程主机上，可能会想保持源文件的文件属性：

```
$ scp -rp $HOME server.example.com:myhome/
```

7.5.5 自动删除源文件

如果需要，*scp2* 在拷贝文件完成之后可以删除源文件。*-u* 命令行选项就可以完成这个功能（注 20）：

```
# 仅对 SSH2  
$ scp2 myfile server.example.com:
```

注 19： 拷贝单个文件时也存在这些限制，但是至少可以很快看到出错信息。如果是目录的话，可能拷贝过去很多层之后还没有发现其实拷错了。

注 20： 在某些 SSH2 的早期版本中，该选项未生效。

```
$ ls myfile
myfile
$ scp2 -u myfile server.example.com:
$ ls myfile
myfile: No such file or directory
```

如果想在安全拷贝之外再加上一个“安全移动”的命令，那么可以通过定义一个`scp2 -u`的别名来实现：

```
$ alias smv='scp2 -u'
```

7.5.6 安全功能

`scp`有两个功能可以保护用户不会运行一些危险的命令。假设用户想把本地文件`myfile`拷贝到一个远程目录中，这样使用：

```
# SSH1, SSH2, OpenSSH
$ scp2 myfile server.example.com:mydir
$ rm myfile
```

然后登录到`server.example.com`上去看这个文件，糟糕的情况发生了：`mydir`并不是一个目录，而是一个文件，你刚才把它给覆盖了！`-d`命令行选项就可以防止这种惨剧的发生。如果目标不是目录，`scp`就会显示一些警告信息，并不拷贝文件，而是直接退出：

```
# SSH1, SSH2, OpenSSH
$ scp2 -d myfile server.example.com:mydir
warning: Destination file is not a directory.
warning: Exiting.
```

只有在仅拷贝一个文件时，这个选项才是必要的。如果要拷贝多个文件或一个目录，所有的`scp`实现都会缺省检查远程目标是否是一个目录（注 21）。

`scp2`的另外一个安全特性是`-n`选项，该选项告诉程序向客户端用户描述一下自己的操作，但不会执行任何拷贝工作。这对在执行一个有可能会发生危险的`scp2`命令之前对其进行验证非常有用。

注 21：有一种情况不会检查。即如果在单独一台机器上拷贝，如，`scp *.c mydir`，`scp`客户端就不必检查`mydir`是否是一个目录。

```
# 仅对SSH2
$ scp2 -n myfile server.example.com:
Not transferring myfile -> server.example.com:./myfile (1k)
```

7.5.7 统计信息的显示

在 *scp* 拷贝文件的同时，它会显示进度的统计信息。

7.5.7.1 *scp1* 的统计信息

对 *scp1* 来说，统计信息显示的内容可以使用命令行选项和环境变量进行配置（注 22）：

```
$ scp1 myfile* server.example.com:
myfile1      |      50 KB | 50.0 kB/s | ETA: 00:00:00 | 100%
myfile2      |      31 KB | 31.3 kB/s | ETA: 00:00:00 | 100%
myfile3      |       3 KB | 3.8 kB/s | ETA: 00:00:00 | 100%
```

scp1 会显示每个文件的文件名、大小、传输速率以及两个传输进度。“ETA”（估计所需时间）是传输文件估计所需要的时间，最后一个数字是到现在为止文件传输的进度（百分比）。随着文件的不断传输，ETA 值会逐渐减小到 0，传输百分比会逐渐增加至 100，但是在上面这个例子的最后结果中，用户是看不出来这种变化过程的。

虽然这些统计信息可以给用户提供相当丰富的信息，但是有时用户也可能会想对其进行修改或将其禁用。例如，在一个批处理任务时使用 *scp1*，用户就可能希望关闭统计信息，这样就不会在屏幕上产生输出。

有几种方法可以对这些统计信息的显示进行配置：包括命令行选项和环境变量（请参看表 7-4，注意命令行选项的优先级高于环境变量）。

表 7-4：*scp1* 中统计信息的控制

希望显示的统计信息	使用选项	设置环境变量
不显示统计信息*	<i>scp1 -q</i>	<code>SSH_NO SCP_STATS</code>

注 22： 对初学者而言，编译 *scp1* 时必须带 `--with-scp-stats` 选项，不然就看不到统计信息。[4.1.5.11]

表 7-4: *scp1* 中统计信息的控制 (续)

希望显示的统计信息	使用选项	设置环境变量
显示统计信息，但不是逐个文件显示	<i>scp1 -Q -A</i>	SSH_NO_ALL SCP_STATS SSH SCP_STATS
为每个文件都显示统计信息	<i>scp1 -Q -a</i>	SSH_ALL SCP_STATS SSH SCP_STATS

a. 同样适用于 OpenSSH 的 *scp* 客户端。

首先，用户可以控制是否要显示统计信息。这可以使用命令行选项 *-q* 和 *-Q* 实现，也可以使用环境变量 *SSH SCP_STATS* 和 *SSH_NO SCP_STATS* 实现。要禁止显示统计信息，可以使用这两种设置：

```
# SSH1, OpenSSH
$ scp -q myfile server.example.com:

# 仅对 SSH1
$ setenv SSH_NO SCP_STATS 1
$ scp1 myfile server.example.com:
```

要显示统计信息，可以使用下面这两种形式：

```
# 仅对 SSH1
$ scp1 -Q myfile server.example.com:

# 仅对 SSH1
$ setenv SSH SCP_STATS 1
$ scp1 myfile server.example.com:
```

如果允许显示统计信息，还可以选择是否为每个文件都显示一个统计信息。这可以使用命令行选项 *-a* 和 *-A* 实现，也可以使用环境变量 *SSH_ALL SCP_STATS* 和 *SSH_NO_ALL SCP_STATS* 实现。要为每个文件都显示统计信息，可以使用下面这两种形式：

```
# 仅对 SSH1
$ scp1 -Q -a myfile server.example.com:

# 仅对 SSH1
$ setenv SSH_ALL SCP_STATS 1
$ scp1 myfile server.example.com:
```

要想只显示一个统计信息，其中累积了所有文件的统计信息，可以这样使用：

```
# 仅对 SSH1  
$ scp1 -Q -A myfile server.example.com:  
  
# 仅对 SSH1  
$ setenv SSH_NO_ALL_SCP_STATS 1  
$ scp1 myfile server.example.com:
```

7.5.7.2 scp2 统计信息

*scp2*统计信息的显示也是可以配置的，但是在SSH2 2.1.0的手册页中并没有给出这些内容。缺省情况下，统计信息的显示是启用的，也没有像SSH1的`--with-scp-stats`一样的编译时选项来禁用这个特性。*scp2*显示的内容和*scp1*的内容看起来不同：

```
$ scp2 myfile* server.example.com:  
Transferring myfile1 -> server.example.com:./myfile1 (50k)  
|.....|  
51200 bytes transferred in 1.00 seconds [50.0 kB/sec].  
Transferring myfile2 -> server.example.com:./myfile2 (30k)  
|.....|  
31744 bytes transferred in 1.03 seconds [31.3 kB/sec].  
Transferring myfile3 -> server.example.com:./myfile3 (3k)  
|.....|  
3068 bytes transferred in 0.79 seconds [3.8 kB/sec].
```

进程进度指示器（一个个点形成的一条线）会随着文件的传输不断变化，但是坦白地讲，这样并不直观。要禁止显示统计信息，可以使用`-Q`命令行选项（它和SSH1的`-Q`选项的意思正好相反）：

```
$ scp2 -Q myfile server.example.com:
```

7.5.8 定位 ssh 可执行程序

要安全地拷贝文件，*scp*就需要在内部调用*ssh*。因此，*scp*需要知道*ssh*可执行程序在磁盘上的什么地方。通常，*ssh*的路径都是在编译时通知*scp*的（使用编译时标志`--prefix`），但如果喜欢，也可以手工指定这个路径。[4.1.5.2] 例如，可以使用一个老版本的*scp*来测试新版本的*ssh*。此时用户可以使用`-S`命令行选项来指定路径：

```
# SSH1, SSH2  
$ scp -S /usr/alternative/bin/ssh myfile server.example.com:
```

7.5.9 内部保留用法

SSH1和OpenSSH的`scp`有两个内部保留选项`-t`和`-f`，这两个选项都没有文档进行说明。我们基本上不会用到这两个选项。它们告诉`scp`拷贝文件的方向：从本地机器拷贝到远程机器，还是从远程机器拷贝到本地机器。`-t`选项的意思是把文件从本地机器拷贝到远程机器，`-f`选项的意思是把文件从远程机器拷贝到本地机器。

无论何时调用`scp`，它都会秘密地在远程机器上运行第二个`scp`进程，其中就在命令行中使用了`-t`或`-f`选项。如果可以使用详细模式运行`scp`，就可以看到这一点。如果是从本地把文件拷贝到远程机器上，就可以看到：

```
$ scp -v myfile server.example.com:  
Executing: host server.example.com, ..., command scp -v -t :  
...
```

反之，如果是把文件从远程机器拷贝到本地，那么就可以看到：

```
$ scp -v server.example.com:myfile .  
Executing: host server.example.com, ..., command scp -v -f :  
...
```

同样，用户几乎永远都不会使用这些选项，但是在想详细分析`scp`输出结果时它们是非常有用的。而且，`scp`的手册页中也提到了这些选项，因此应该理解这些选项是什么。

7.6 小结

SSH客户端可以使用环境变量、命令行选项以及配置文件中的关键字进行配置。命令行选项的优先级最高，其次是本地客户端的配置文件，优先级最低的是全局客户端配置文件。

客户端配置文件由很多段组成，几个段可以共同用于一次调用。如果同一个关键字被设置了多次，那么最先（SSH1、OpenSSH）或最后（SSH2）出现的值优先级最高。

用户在试验客户端配置时，请记得使用详细模式。如果要试验SSH的特殊行为，那么应该在再次运行客户端时本能地加上`-v`选项，以观察调试输出。

本章内容：

- 密钥的局限
- 基于公钥的配置
- 可信主机访问控制
- 用户认证文件
- 小结

第八章

每账号

服务器配置

到现在为止我们已经介绍了两种在全局上控制SSH服务器行为的方法：编译时配置（第四章）和服务器范围的配置（第五章）。这两种技术对到达给定服务器的所有SSH连接都会产生影响。现在我们开始介绍第三种服务器控制方法，也是更细致的一种方法：每账号配置。

顾名思义，每账号配置让SSH服务器可以区分每个服务器上的各个用户。例如，用户sandy可以接收从Internet上任何机器上发来的SSH连接，而rick只允许接收来自`verysafe.com`域的SSH连接，fraidycat拒绝所有基于密钥的连接。每个用户都可以使用图8-1中高亮显示的机制来配置自己的账号，这不需要特殊的权限，也不需要求助于系统管理员。

我们已经看过一种简单的每账号配置。用户可以把公钥放入自己的认证文件中，由此来通知SSH服务器使用公钥认证登录到自己的账号中。但是每账号配置的功能远不止于此，它可以成为访问控制的一种功能强大的工具，可以让用户的账号使用一些有趣的技巧，而控制是否可以使用特定密钥或主机进行连接只不过是牛刀小试而已。例如，用户可以让到达的SSH连接运行自己选定的程序，而不运行客户端选定的程序。这称为强制命令（forced command），后文中我们会介绍很多有趣的应用程序。

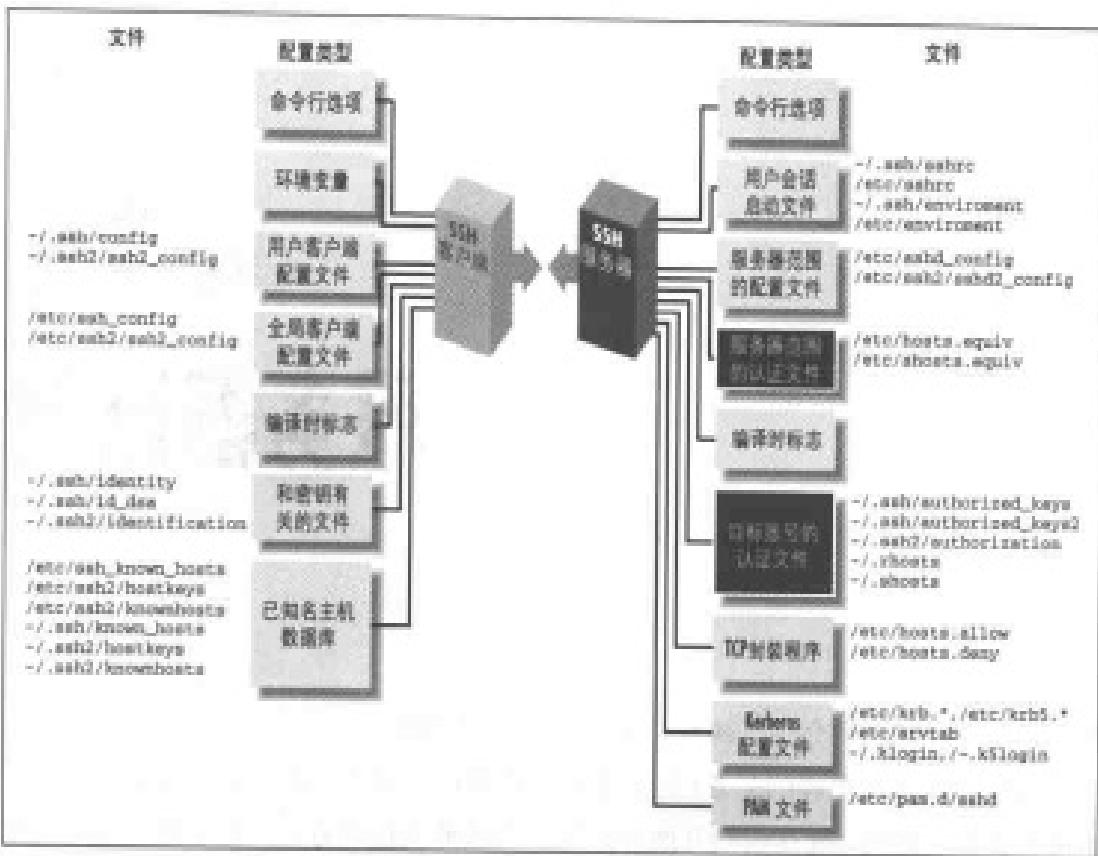


图 8-1：每账号配置（高亮显示部分）

每账号配置只能控制那些到达自己账号的SSH连接。如果对运行SSH客户端配置外发的SSH连接感兴趣，请参考第七章。

8.1 该技术的局限

每账号配置可以处理很多有趣的事情，但是它也有一些局限（我们会陆续讨论）：

- 每账号配置不能覆盖使用编译时配置和服务器范围配置所采用的安全措施。（感谢上帝！）
- 如果使用公钥认证，那么每账号配置就十分灵活和安全。可信主机和密码认证提供的选择范围要更小。

8.1.1 覆盖服务器范围的设置

用户账号的SSH设置只能对到达连接的认证进行限制。这些设置不能启用那些已经

在更大范围内禁用的SSH特性，而且不能允许已禁用的用户或主机进行认证。例如，如果SSH服务器拒绝接收所有来自 *evil.org* 域的连接，那么就不能使用每账号配置在自己的账号中覆盖此限制（注1）。

这样限制是很有道理的，因为终端用户不能违反整个服务器的安全策略。但是，我们应该允许（而且的确允许）终端用户限制到达其账号的连接。

有些服务器范围的特性是可以被每账号配置覆盖的。最明显的一个特性是服务器的空闲超时时间（idle timeout）。这个特性可以超出服务器范围设置的限制，但是也不能强制服务器接受已经在全局上配置为禁用的连接。

如果是一个终端用户，而且每账号配置并没有提供足够的灵活性，那么就可以运行自己的SSH服务器实例，这样就可以将其配置得让人满意。但是要小心，因为这通常都不“合法”。你正试图绕过的限制是系统管理员为机器定制的安全策略的一个部分，你不应该仅仅因为自己可以避开这种限制就对这种策略满不在乎。如果现在机器是由你控制的，那么就可以随心所欲地配置主SSH服务器。否则，安装并运行自己的`sshd` 可能违反你的使用许可协议，并且 / 或者肯定会干扰系统管理员的工作，这在什么时候都不是明智之举。

8.1.2 认证问题

要充分利用每账号配置，就得使用公钥认证。密码认证太有限了，因为控制访问权限的惟一方法是密码本身。可信主机认证有一定的灵活性，但还是远远不如公钥认证。

如果你现在仍然对密码认证感到困惑，就把这当成采用公钥的另一个理由好了。虽然密码和公钥口令这两个术语看起来可能是类似的（都是输入一串密文，然后就登录进来了），但是对于允许或者拒绝访问你的账号这个目的来说，公钥口令比密码灵活得多。请耐心往下看，你就会知道原因了。

注1： 本条规则有一个例外：可信主机认证。用户的`~/.shosts`文件可以覆盖系统管理员在`/etc/shosts.equiv`文件中所做的限制。请参看可信主机访问控制部分。

8.2 基于公钥的配置

要在SSH服务器上自己的账号中设置使用公钥认证，得创建一个认证文件，通常是 *authorized_keys* (SSH1, OpenSSH/1), *authorized_keys2* (OpenSSH/2) 或者 *authorization* (SSH2)，并在其中加入能让你访问你的账号的密钥。你的认证文件可能不只包含密钥，还包含其他的关键字，以及一些高级SSH服务器控制选项。(在后文中) 我们会讨论：

- 认证文件的完全格式。
- 用于限制客户端可以在服务器上调用的程序的强制命令。
- 限制来自特定主机的连接。
- 为远程程序设置环境变量。
- 设置空闲超时时间，这样如果客户端用户不再发送数据就强制将其断连。
- 对到达的SSH连接禁用某些特性，例如端口转发和tty分配。

在我们介绍如何修改自己的认证文件的过程中，别忘记SSH服务器只有在认证时才会参考该文件。因此，如果你对认证文件进行了修改，那么只有(此后)新的连接可以使用这些新信息。现存的所有连接都已经认证过了，不会受到影响。

还要记住，如果SSH服务器由于其他原因拒绝了到达的连接请求(例如不满足服务器范围配置的要求)，那么这些连接请求就不会访问认证文件。因此如果对认证文件的修改看来无效，就要确认这些修改是否和(更高级别的)服务器范围配置中的设置冲突。

8.2.1 SSH1 认证文件

SSH1的*authorized_keys*文件通常都是`~/.ssh/authorized_keys`，它是使用SSH-1协议通往你的账号的一道安全关口。该文件的每一行都包含一个公钥，其意义如下：“我授权SSH-1客户端以一种特殊的方法(使用本密钥进行认证)来访问我的账号。”(注意此处只可以使用一种特殊的方法进行访问，而不是所有的方法都可以。) 到现在为止，公钥为账号提供的访问权限是无限制的。现在我们继续看其他的内容。

*authorized_keys*文件的每一行都依次包含三项内容，有些是可选的，有些是必需的：

- 一些选项 (option, 可选的。选项放在这里可真奇怪)。
- 公钥 (public key, 必需的)。公钥分为三个部分：
 - 密钥的位数，通常是一个小整数，例如 1024
 - 密钥的指数 (exponent)：整数
 - 密钥的模数 (modulus)：一个很大的整数，通常有几百个数字长
- 注释(descriptive comment, 可选的)。注释可以是任何文本，例如“Bob's public key” 或 “My home PC using SecureCRT 3.1”。

公钥和注释都是由 *ssh-keygen* 生成的，存储在 *.pub* 文件中，用户可以在该文件中重新获得公钥和注释，通常都可以使用拷贝粘贴的方法将其插入 *authorized_keys* 文件中。然而，选项通常是使用文本编辑器输入 *authorized_keys* 中的（注 2）。

每个选项都可以有两种格式。可以是一个关键字，例如：

```
# SSH1, OpenSSH: 禁止端口转发  
no-port-forwarding
```

也可以是一个关键字，之后紧跟一个等号和一个值，例如：

```
# SSH1, OpenSSH: 设置空闲超时时间为 5 分钟  
idle-timeout=5m
```

多个选项可以放在一行中，彼此之间使用逗号分隔开，在选项之间不要使用空格：

```
* SSH1, OpenSSH  
no-port-forwarding,idle-timeout=5m
```

如果错误地输入了空格：

注 2： 在编辑 *authorized_keys* 文件时，要确保使用可以处理长行的文本编辑器。密钥的模数可能会长达几百个字符。有些文本编辑器不能显示长行，因此也不能正确地编辑，它会自动插入行结束符，或者在本来好好的公钥上面加入乱七八糟的东西。（咳，我们就别讨论那些伤脑筋的文本编辑器了。）使用最新的编辑器，并关掉自动换行。这里我们用的是 GNU Emacs。

```
# THIS IS ILLEGAL: 选项中的空格
no-port-forwarding, idle-timeout=5m
```

那么使用该密钥的连接就不能正常工作。如果在连接的时候启用了调试特性 (*ssh -v*)，就会从 SSH 服务器上看到一个“syntax error (语法错误)”的消息。

很多 SSH 用户并不知道这些选项，或者忽视了这些选项的使用。这真是遗憾，因为这些选项提供了额外的安全控制和便利。用户对访问自己账号的客户端了解得越多，可用的访问控制选项也就越多。

8.2.2 SSH2 认证文件

SSH2 认证文件，通常都是 *~/.ssh2/authorization* (注 3)，其格式和 SSH1 认证文件有所不同。SSH2 认证文件不再包含公钥，而是包含关键字和值，它更类似于我们前面见过的其他 SSH 配置文件。该文件的每一行都包含一个关键字，其后紧跟着其值。最通用的关键字是 Key 和 Command。

公钥是使用 Key 关键字来指明的。Key 之后紧跟一个空格，然后是包含公钥的文件名。这些文件是指 *~/.ssh2* 目录中的文件。例如：

```
# 仅对 SSH2
Key myself.pub
```

就意味着 SSH-2 公钥包含在 *~/.ssh2/myself.pub* 文件中。要使用公钥认证，认证文件必须至少包含一个 Key 行。

每个 Key 行之后可以紧跟一个 Command 关键字和其值，这是可选的。Command 指定一个强制命令 (forced command)，强制命令是无论何时使用前面的密钥进行访问时都会执行的命令。稍后我们会详细讨论强制命令。(请参看强制命令部分。) 现在所要了解的是：强制命令是使用关键字 Command 开始的，之后紧跟一个空格，最终以一个 shell 命令行结束。例如：

注 3：这个名字可以在服务器范围配置文件中使用 *AuthorizationFile* 关键字进行修改。
ssh2 手册页中说明 *AuthorizationFile* 也可以在客户端配置文件中设置，但是在 SSH2 2.2.0 中这种设置是无效的。这点也并没有什么可奇怪的，因为 *sshd2* 并不读取客户端的配置文件。

```
# 仅对 SSH2  
Key somekey.pub  
Command "/bin/echo All logins are disabled"
```

记住，只有一个 Command 行是不对的。下面的例子都是非法的：

```
# 这是非法的：无 Key 行  
Command "/bin/echo This line is bad."  
# 这是非法的：在第二个 Command 前面无 Key 行  
Key somekey.pub  
Command "/bin/echo All logins are disabled"  
Command "/bin/echo This line is bad."
```

8.2.2.1 SSH2 PGP 密钥认证

2.0.13 版本的 SSH2 引入了对 PGP 认证的支持。认证文件还可以包括 PgpPublicKeyFile、PgpKeyName、PgpKey Fingerprint 和 PgpKeyId 行。就像 Key 行一样，Command 行也可以跟在 PgpKeyName、PgpKey Fingerprint 或者 PgpKeyId 行之后：

```
# 仅对 SSH2  
PgpKeyName my-key  
Command "/bin/echo PGP authentication was detected"
```

8.2.3 OpenSSH 认证文件

对于 SSH-1 协议的连接来说，OpenSSH/1 使用和 SSH1 相同的 *authorized_keys* 文件。SSH1 可用的所有配置在 OpenSSH/1 中都可以使用。

对于 SSH-2 协议的连接来说，OpenSSH/2 采用的方法和 SSH2 的方法不同：它使用了一个新的认证文件：*~/.ssh/authorized_keys2*，该文件的格式和 *authorized_keys* 类似。该文件的每一行可以包括以下内容：

- 密钥认证选项（可选的）。
- 字符串“*ssh-dss*”（必需的）。
- DSA 公钥，用一个长字符串来表示（必需的）。
- 注释（可选的）。

这里有一个例子，其中我们对长公钥字符串进行了缩减：

host-192.168.10.1 ssh-dss AAAQABJQzaC1kc3Ma... My OpenSSH key

8.2.4 强制命令

通常，SSH连接会调用客户机选定的远程命令：

```
# 调用远程登录 shell  
$ ssh server.example.com  
# 调用远程目录列表  
$ ssh server.example.com /bin/ls
```

强制命令把这种控制权从客户端转交给了服务器。(使用强制命令,)不是由客户端来规定要运行哪个命令,而是由服务器账号的属主来规定。在图 8-2 中,客户端请求执行命令 /bin/ls,而服务器端的强制命令却运行 /bin/who。

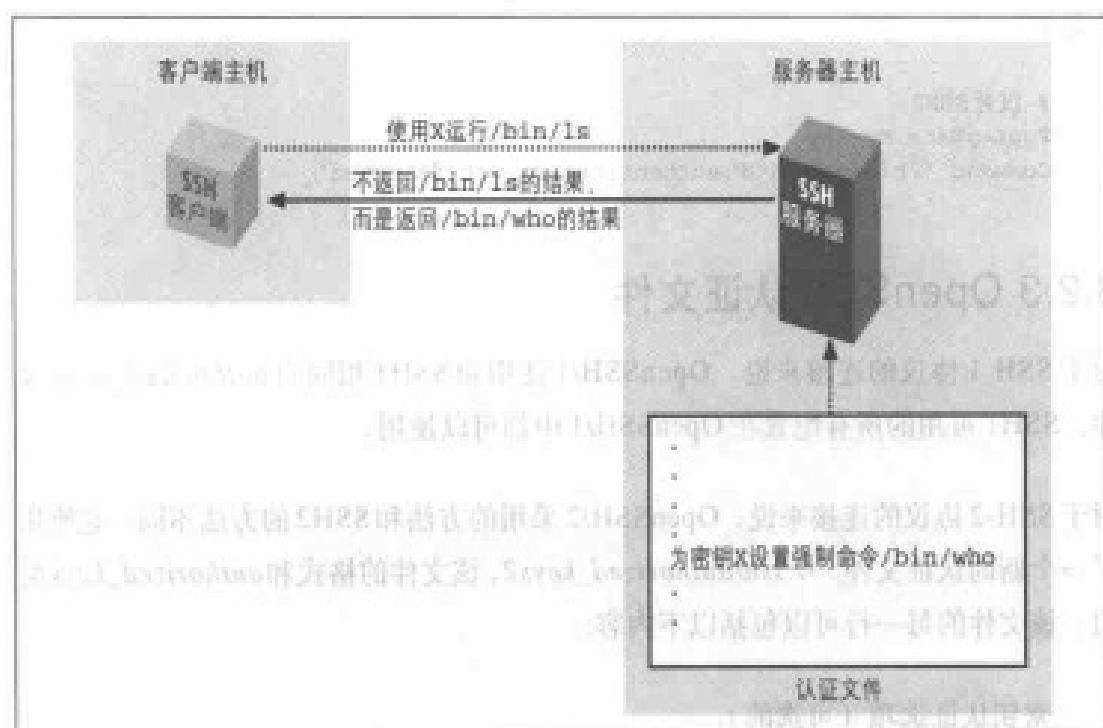


图 8-2：强制命令使用 /bin/who 取代 /bin/ls

强制命令十分有用。假设想给你的助手授权，让他可以访问你的账号，但是只能读取你的信件，那么你就可以给你的助手的 SSH 密钥上关联一个强制命令，从而让他只能运行你的 email 程序，而不能执行其他操作。

在 SSH1 和 OpenSSH 中，我们可以在 *authorized_keys* 文件中指定的密钥之前使用“Command”选项指定强制命令。例如，要让你的助手每次连接上来都运行 email 程序 *pine*，可以这样：

```
# SSH1, OpenSSH
command="/usr/local/bin/pine" ...secretary's public key...
```

在 SSH2 中，强制命令紧跟在指定的 Key 之后，(开头) 要使用 Command 关键字。前面的例子可以这样表示：

```
# 仅对 SSH2
Key secretary.pub
Command "/usr/local/bin/pine"
```

用户最多可以给某个特定的密钥关联一个强制命令。要给一个密钥关联多个强制命令，就得把这些命令放入远程主机的一个脚本中，并将该脚本作为强制命令运行。(我们会在“显示命令菜单”部分中介绍这方面的内容。)

8.2.4.1 安全问题

在开始深入介绍强制命令的例子之前，让我们先来讨论一下安全问题。乍一看，强制命令似乎是对调用 shell 的“普通” SSH 连接进行安全化处理。这是因为 shell 可以调用任何程序，而强制命令只能调用一个程序，也就是强制命令本身。如果强制命令是 */usr/local/bin/pine*，那么用户只能调用 */usr/local/bin/pine*。

然而，有一点需要注意。强制命令如果使用不当，可能会让你有一种错觉，使你处于一种“伪安全 (false security)”状态，以为自己已经限制了客户端的能力，实际上却没有。如果强制命令无意中开放了一个 shell 出口 (即允许在强制命令中调用 shell) 就会发生这种情况。如果存在 shell 出口，那么 shell 可以调用的任何程序在客户端中都可以调用。很多 Unix 程序都有 shell 出口，例如文本编辑器 (*vi*, *Emacs*)、分页程序 (*more*, *less*)、调用分页程序的程序 (*man*)、新闻阅读程序 (*rn*)、邮件阅读程序 (例如前面例子中的 *pine*) 以及调试程序 (*adb*)。交互程序通常最容易出问题，但即使是非交互程序也可以运行 shell 命令 (*find*, *xargs* 等等)。

在定义强制命令时，你可能不想让其密钥用于任意的 shell 命令。因此，我们建议使用以下的安全规则来判断一个程序是否适合用作强制命令。

- 避免使用具有 shell 出口的程序。仔细阅读程序文档。如果仍然不能确认（该程序是否具有 shell 出口），就向其他人求助。
- 避免使用编译器、解释器或其他可以让用户生成并运行任意可执行代码的程序。
- 慎重对待任何可以在用户指定位置创建或删除文件的程序。这不仅仅包括应用程序（字处理软件、图形程序等等），而且包括可以移动或拷贝文件的命令行工具（*cp*、*mv*、*rm*、*scp*、*ftp* 等等）。
- 避免使用 setuid 或 setgid 被置位了的程序，特别是那些 setuid 为 root 的程序。
- 如果使用脚本作为强制命令，就要遵循编写安全脚本的传统规则。在一个脚本之内，要限制使用相对路径作为搜索路径，应该使用绝对路径来调用所有的程序，不要盲目地把用户提供的字符串作为命令来执行；不要让该脚本执行任何 setuid 的工作（注 4）。再次强调，不要调用具有 shell 出口的程序。
- 考虑使用受限的 shell 来限制到达的客户端连接可以执行的操作。例如，受限的 shell /usr/lib/rsh（不要同名字为“rsh”的 r-command 混淆）可以限制客户端可以输入的远程目录。
- 为一个单独的、专用的 SSH 密钥（不是你用来登录的那个密钥）关联一个强制命令，这样不会影响你的登录能力就可以方便地禁用该密钥。
- 使用我们后面介绍的一些选项来禁用不必要的 SSH 特性。在 SSH1 中，你可以使用 no-port-forwarding 选项禁用端口转发，使用 no-agent-forwarding 选项禁用代理转发，使用 no-pty 禁用 tty 分配。

任何程序都可以用作强制命令，但是有些程序用作强制命令会比较危险。在下面这个例子中，我们就会介绍几个常见的问题。

8.2.4.2 使用定制消息拒绝连接

假设之前你允许一个朋友使用 SSH 来访问自己的账号，但是现在你决定禁止他访问了。你可以简单地将他的密钥从你的认证文件中删除，但还可以把这件事情做得

注 4：出于安全原因，现代的 Unix 通常会忽略脚本的 setuid 位。

更精细一些。你可以定义一个强制命令给你的朋友打印一条定制消息，告诉他的访问权限已经被禁止了。例如：

```
# SSH1, OpenSSH
command="/bin/echo Sorry, buddy, but you've been terminated!" ...key...

# 仅对 SSH2
Key friend.pub
Command "/bin/echo Sorry, buddy, but you've been terminated!"
```

任何到达的SSH连接使用该密钥成功认证之后，都会在标准输出设备上显示以下消息：

```
Sorry, buddy, but you've been terminated!
```

然后就关闭连接。如果想打印一条更长的消息，但将其写入认证文件很不方便，你就可以将其存储在一个单独的文件中（比如`~/go.away`），并使用适当的程序（例如`cat`）显示该消息的内容：

```
# SSH1, OpenSSH
command="/bin/cat $HOME/go.away" ...key...

# 仅对 SSH2
Key friend.pub
Command "/bin/cat $HOME/go.away"
```

由于这条消息很长，你可能会想用`more`或`less`之类的分页程序一次显示一屏。千万不要这样做！

```
# SSH1:不要这样做
command="/bin/more $HOME/go.away" ...key...
```

这个强制命令给你的账号打开了一个漏洞：`more`程序和大部分Unix分页程序一样，都有一个shell出口。该强制命令并没有限制访问权限，而是允许无限制地访问系统。

8.2.4.3 显示命令菜单

假设要限制账号的访问权限，令到达的SSH客户端只能调用几个特定的程序，就可以使用强制命令。例如，用户可以为允许执行的已知程序组编写一个shell脚本，并将该脚本作为强制命令运行。在例子8-1中给出的样例脚本只允许从一个菜单的三个程序中选择。

例 8-1：菜单脚本

```
#!/bin/sh
/bin/echo "Welcome!
Your choices are:

1      See today's date
2      See who's logged in
3      See current processes
q      Quit"

/bin/echo "Your choice: \c"
read ans
while [ "$ans" != "q" ]
do
    case "$ans" in
        1)
            /bin/date
            ;;
        2)
            /bin/who
            ;;
        3)
            /usr/ucb/w
            ;;
        q)
            /bin/echo "Goodbye"
            exit 0
            ;;
        *)
            /bin/echo "Invalid choice '$ans': please try again"
            ;;
    esac
    /bin/echo "Your choice: \c"
    read ans
done
exit 0
```

当有人使用公钥访问你的账号，并调用这个强制命令时，该脚本会显示：

```
Welcome!
Your choices are:
1      See today's date
2      See who's logged in
3      See current processes
q      Quit

Your choice:
```

然后用户可以输入 1、2、3 或者 q 来运行相应的程序。其他的输入都会被忽略，因此不能执行其他程序。

这种脚本必须仔细编写，以防引起安全漏洞。特别是所有允许执行的程序都不能有 shell 出口，否则用户就可以在你的账号中执行所有的命令了。

8.2.4.4 检查客户端的原始命令

正如我们已经看到的一样，强制命令取代了 SSH 客户端可能发送的其他命令。如果 SSH 客户端试图调用 *ps* 程序：

```
$ ssh1 server.example.com ps
```

但是强制命令设置为要执行 “/bin/who”：

```
# SSH1, OpenSSH
command="/bin/who" ...key...
```

那么 *ps* 就被忽略，取而代之的是运行 */bin/who*。不过，SSH 服务器还是要读取客户端发送的原始命令字符串，并将其存储在环境变量中。对于 SSH1 和 OpenSSH（注 5）来说，该环境变量是 **SSH_ORIGINAL_COMMAND**；在 SSH2 中是 **SSH2_ORIGINAL_COMMAND**。因此在本例中，**SSH_ORIGINAL_COMMAND** 的值是 *ps*。

一种快速查看这些变量的方法是用强制命令打印这些变量的值。对于 SSH1，可以创建这样的强制命令：

```
# 仅对 SSH1
command="/bin/echo You tried to invoke $SSH_ORIGINAL_COMMAND" ...key...
```

然后使用 SSH-1 的客户端连接到 SSH1 服务器上，同时提供一个远程命令（该命令并不会被执行），例如：

```
$ ssh1 server.example.com cat /etc/passwd
```

这样，SSH1 服务器并不会执行 *cat*，而是简单的打印以下内容：

```
You tried to invoke cat /etc/passwd
```

然后结束。同理，对于 SSH2 服务器来说，可以定义以下强制命令：

注 5：OpenSSH 的早期版本不会设置 **SSH_ORIGINAL_COMMAND**。

```
# 仅对 SSH2
Key mykey.pub
Command "/bin/echo You tried to invoke $SSH2_ORIGINAL_COMMAND"
```

接着执行这样的客户端命令：

```
$ ssh2 server.example.com cat /etc/passwd
```

会产生如下输出：

```
You tried to invoke cat /etc/passwd
```

8.2.4.5 限制客户端的原始命令

让我们试试 SSH_ORIGINAL_COMMAND 环境变量稍微复杂一些的用法。我们要创建一个强制命令，让其检查环境变量并根据请求命令的不同进一步选择不同的操作。例如，假设你想允许一个朋友在你的账号中调用除 rm（删除文件）之外的远程命令。换而言之，这样的命令：

```
$ ssh server.example.com rm myfile
```

会被拒绝。下面这个脚本会检查命令字符串中是否有 rm，如果有，就拒绝执行该命令：

```
#!/bin/sh
# 仅对于 SSH1；对于 SSH2，要使用$SSH2_ORIGINAL_COMMAND
#
case "$SSH_ORIGINAL_COMMAND" in
  *rm*)
    echo "Sorry, rejected"
    ;;
  *)
    $SSH_ORIGINAL_COMMAND
    ;;
esac
```

将该脚本存储为 *~/rm-checker*，并定义一个强制命令来调用这个脚本：

```
* 仅对 SSH1
command="$HOME/rm-checker" ... key...
```

我们的脚本只是一个例子：这个例子并不安全。用一个聪明的命令序列就能轻易绕开该脚本的限制来删除文件：

```
$ ssh server.example.com '/bin/ln -s /bin/r? ./killer && ./killer myfile'
```

该命令会使用一个不同的名字 (*killer*) 创建一个到 */bin/rm* 的连接, 然后执行删除文件操作。然而, 我们介绍的这种概念依然是正确的: 你可以检查 *SSH_ORIGINAL_COMMAND* 来选择另外一个命令来执行。

8.2.4.6 把客户端的原始命令记录在日志中

“原始命令”环境变量的另外一个很好的用法是可以把使用特定密钥运行的命令记录在日志中。例如:

```
* 仅对 SSH1  
command="log-and-run" ... key...
```

其中的 *log-and-run* 是下面的脚本。它把一行追加到一个日志文件末尾, 其中包含一个时间戳和试图执行的命令:

```
#!/bin/sh  
if [ -n "$SSH_ORIGINAL_COMMAND" ]  
then  
    echo `'/bin/date': $SSH_ORIGINAL_COMMAND` >> $HOME/ssh-command-log  
    exec $SSH_ORIGINAL_COMMAND  
fi
```

8.2.4.7 强制命令和安全拷贝 (scp)

(到现在) 我们已经看到在密钥关联了强制命令时运行 *ssh* 的情况。但是 *scp* 的情况又是如何呢? 是运行强制命令, 还是执行拷贝操作?

在这种情况下, 最终要执行强制命令, 而本来的文件拷贝操作则被忽略了。在不同情况下, 这种结果可能好, 也可能坏。通常, 我们并不推荐将 *scp* 与关联强制命令的密钥一起使用。相反, 我们推荐使用两个密钥, 其中一个用于普通登录和文件拷贝, 另外一个用于强制命令。

现在我们已经详细介绍了强制命令, 接下来让我们开始介绍每账号配置的其他特性。

8.2.5 基于主机或域的访问控制

公钥认证需要两方面信息: (与公钥) 对应的私钥及其口令 (如果有的话)。缺少任

任何一部分内容，认证都不能成功。每账号配置可以加入第三种必须的信息，从而（给系统）添加了额外的安全性：即对于客户端主机名或IP地址的限制。这是由 `from` 选项完成的。例如：

```
# SSH1, OpenSSH
from="client.example.com" ...key...
```

会强制SSH-1连接必须来自 `client.example.com`，否则就会被拒绝。因此，即使你的私钥文件被盗窃了，而且你的口令也被破解了，只要攻击者不能从可信的客户端机器登录，那么他也会被阻隔在系统之外。

如果你觉得“`from`”的概念听起来很熟悉，那么说明你的记忆力很不错：它和服务器范围配置中 `AllowUsers` 关键字所提供的访问控制是相同的。但是，`authorized_keys` 选项是在你的账号中设置的，只能用于一个密钥，而 `AllowUsers` 是由系统管理员指定的，可以用于登录到一个账号的所有连接。这里有一个例子来解释这种区别。假设你允许来自 `remote.org` 的连接登录到 `benjamin` 账号中，作为系统管理员，可以在 `/etc/sshd_config` 中这样配置：

```
# SSH1, OpenSSH
AllowUsers benjamin@remote.org
```

使用每账号配置，用户 `benjamin` 可以在 `authorized_keys` 文件中配置同样的设置，但是只用于一个特定的密钥：

```
# SSH1, OpenSSH
# File -benjamin/.ssh/authorized_keys
from="remote.org" ...key...
```

当然，服务器范围设置优先级较高。如果系统管理员已经使用 `DenyUsers` 关键字拒绝了这种访问：

```
# SSH1, OpenSSH
DenyUsers benjamin@remote.org
```

那么用户 `benjamin` 就不能再在 `authorized_keys` 中使用 `from` 选项来覆盖这种限制。

和 `AllowUsers` 类似，`from` 选项也可以用通配符 * 匹配任意字符串，用? 来匹配任意一个字符：

`from="*.someplace.org"`

可以匹配 `someplace.org` 域中的所有主机

from="som?pla?e.org"	可以匹配 somXplaYe.org, 但是不能匹配 foo, someXplaYe.org 或者 foo.somplace.org
----------------------	---

它还可以匹配客户端的 IP 地址，在 IP 地址中可以用通配符，也可以不用（这在手册中并没有介绍）：

```
from="192.220.18.5"  
from="192.2???.18.*"
```

也可以（在一个 from 选项中加入）多种模式，中间用逗号分隔（AllowUsers 采用空格）；其中不允许有空格。用户也可以使用一个惊叹号（!）前缀，这样就是否定模式。确切的匹配规则是：该列表中的每个模式都和客户端的规范主机名或其 IP 地址进行比较。如果该模式只包含数字、点和通配符，那么它就和 IP 地址匹配，否则就和主机名匹配。客户端当且仅当可以至少匹配一个肯定模式，并且不能匹配否定模式时，才能接受连接。例如，下面的规则会拒绝来自 *saruman.ring.org* 的连接，但是允许来自 *ring.org* 域中的其他主机的连接，并拒绝来自其他任何域中的连接：

```
from="!saruman.ring.org,*.ring.org"
```

而下面这条规则会拒绝来自 *saruman.ring.org* 的连接，但是允许来自其他域中所有客户端的连接：

```
from="!saruman.ring.org,*"
```

不幸的是，SSH1 不能让你使用地址和掩码来指定任意的 IP 网络，也不能让你使用地址 / 位数的形式。但是 *libwrap* 可以指定任意的 IP 网络，不过其限制是对所有的连接生效，而不是遵循每密钥的规则。

记住，使用基于主机名的访问控制可能是有问题的，这些问题是由域名解析和安全性的原因所致。幸运的是，from 选项只是 SSH-1 公钥认证的一个附加特性，不使用它也可以实现完整的基于主机的解决方案，它是在此基础上又提供了一层更高的安全性。

8.2.5.1 在 SSH2 中模拟 “from” 选项

虽然 SSH2 不能支持 from 选项，但用户也可以在 SSH2 中使用强制命令来创建自己的基于主机的访问控制。技巧是检查环境变量 \$SSH2_CLIENT，并创建一个脚本来执行以下步骤：

- 从\$SSH2_CLIENT中提取出到达的客户端的IP地址，它是字符串中的第一个值。
- 根据该IP地址或者任何你喜欢的逻辑规则接受或拒绝连接。

例如，假设你想允许来自IP地址24.128.97.204的连接，并拒绝来自128.220.85.3的连接。将下面的脚本作为一个强制命令调用便可以实现这种技巧：

```
#!/bin/sh
IP=`echo $SSH2_CLIENT | /bin/awk '{print $1}'`
case "$IP" in
  24.128.97.204)
    exec $SHELL
    ;;
  128.220.85.3)
    echo "Rejected"
    exit 1
    ;;
esac
```

把该脚本命名为（比如）`~/ssh2from`，并将其作为一个SSH2强制命令，（任务）就完成了：

```
# 仅对SSH2
Key mykey.pub
Command "$HOME/ssh2from"
```

这种技术只能用于IP地址，不能用于主机名。但是，如果你信任名字服务，当然可以把\$SSH2_CLIENT中得到的IP地址转换成主机名。在Linux中用`/usr/bin/host`就可以达到目的，比如说只接受来自*client.example.com*的连接或来自*niceguy.org*域的连接：

```
#!/bin/sh
IP=`echo $SSH2_CLIENT | /bin/awk '{print $1}'`
HOSTNAME=`/usr/bin/host $IP | /bin/awk '{print $5}'`
case "$HOSTNAME" in
  client.example.com)
    exec $SHELL
    ;;
  *.niceguy.org)
    exec $SHELL
    ;;
  *)
    echo "Rejected"
    exit 1
    ;;
```

```
esac
```

8.2.6 设置环境变量

`environment` 选项可以指示 SSH1 服务器在客户端使用给定的密钥连接时设置一个环境变量。例如，*authorized_keys* 文件的一行如下：

```
# SSH1, OpenSSH
environment="EDITOR=emacs" ...key...
```

它把环境变量 `EDITOR` 设置成 `emacs`，从而可以设置客户端用于登录会话的缺省编辑器。`environment=` 之后的语法是一个使用引号括起的字符串，其中包括一个变量、一个等号和一个值。引号之间的所有字符都是有意义的，也就是说，其值可以包含空格：

```
# SSH1, OpenSSH
environment="MYVARIABLE=this value has whitespace in it" ...key...
```

甚至可以包含双引号，但要用转义字符斜线 (\) 对双引号进行标注：

```
# SSH1, OpenSSH
environment="MYVARIABLE=I have a quote\" in my middle" ...key...
```

此外，*authorized_keys* 文件中一行可以设置多个环境变量：

```
# SSH1, OpenSSH
environment="EDITOR=emacs",environment="MYVARIABLE=26" ...key...
```

为什么要为密钥设置环境变量呢？这可以让用户对自己的账号进行适当的定制，以便根据使用的密钥的不同而进行不同的响应。例如，假设用户创建了两个密钥，每个密钥都为一个环境变量（如，`SPECIAL`）设置了不同的值：

```
# SSH1, OpenSSH
environment="SPECIAL=1" ...key...
environment="SPECIAL=2" ...key...
```

现在，用户可以在自己账号的 shell 配置文件中检查 `$SPECIAL`，并触发每个密钥的特定操作：

```
* 在 .login 文件中
switch ($SPECIAL)
  case 1:
```

```

echo 'Hello Bob!'
set prompt = 'bob> '
breaksw
case 2:
    echo 'Hello Jane!'
    set prompt = jane> '
    source ~/.janerc
    breaksw
endsw

```

在此，我们给每一个密钥的用户打印一条定制的欢迎语。如果登陆的是 Jane，就调用定制的初始化脚本，`~/.janerc`。这样，environment 选项就在 `authorized_keys` 和远程 shell 之间提供了一种方便的通信手段。

8.2.6.1 例子：CVS 和 \$LOGNAME

现在给出一个 environment 选项更高级的例子。假设在 Internet 上，一组开放源码的软件开发人员正在开发一个计算机程序。他们决定实施良好的软件工程并使用 CVS (Concurrent Versions System，并发版本系统) 版本控制工具来存储源代码。由于缺乏资金建立自己的服务器，因此他们要将 CVS 仓库 (repository) 放到该组成员之一 benjamin 的账号中，因为他有很大的可用磁盘空间。benjamin 的账号在 SSH 服务器 `cvs.repo.com` 上。

其他开发人员在 `cvs.repo.com` 上并没有账号，因此 benjamin 要把大家的公钥放入 `authorized_keys` 文件，这样其他开发人员就可以执行导入 (check-in) 的工作。但是现在有一个问题，当软件开发人员对文件进行了修改并将该文件的新版本导入仓库时，CVS 会生成一个日志项，说明修改文件的作者。但是每个开发人员都是使用 benjamin 账号登录到服务器上的，因此不管到底是谁导入了修改过的文件，CVS 总是会把作者定义成 “benjamin”。从软件工程的标准来看，这可不是件好事；每个改动的作者都应该被清晰地标识出来（注 6）。

用户可以通过修改 benjamin 的 (认证) 文件来解决这个问题。在每个开发人员的密钥前面加上一个 environment 选项，CVS 会检查 LOGNAME 环境变量来获取作者名，这样就可以为每个开发人员的密钥设置不同的 LOGNAME：

注 6： 在工业开发环境中，每个开发人员在 CVS 仓库主机上都有一个账号，因此不可能出现这种问题。

```
# SSH1, OpenSSH
environment="LOGNAME=dan" ...key...
environment="LOGNAME=richard" ...key...
...
```

现在再在CVS导入操作中使用一个给定密钥时，CVS就会识别出执行相应操作的作者，它是由LOGNAME唯一确定的。这样问题就解决了（注7）。

8.2.7 设置空闲超时时间

idle-timeout选项通知SSH1服务器将已经空闲时间超过一定界限的会话断连。这和服务器范围配置的IdleTimeout关键字很相似，不同之处在于idle-timeout是在你的账号中设置的，而不是由系统管理员设置的。

假设你要让你的朋友Jamie使用SSH-1协议访问自己的账号。但是Jamie的工作环境并不充分可信，你会担心他可能在连接到自己的账号的时候离开计算机，这样其他人可能正好路过，就可以使用他的会话进行操作。减少这种风险的一种方法是为Jamie的密钥设置一个空闲超时时间，这样超过给定的空闲超时时间之后，SSH-1会话就会自动断连。如果客户端有一段时间没有输出数据，那么Jamie就可能已经离开了，这样就可以结束他的会话。

超时时间是在idle-timeout选项中设置的。例如，要把空闲超时时间设置成60秒：

```
# SSH1, OpenSSH
idle-timeout=60s ...key...
```

idle-timeout使用的时间符号和IdleTimeout关键字相同：一个整数，后面跟上一个字母（可选）说明时间单位。例如，60s是60秒钟，15m是15分钟，2h是2小时。如果数字后面没有字母，缺省的单位是秒。

idle-timeout选项会覆盖服务器范围配置中IdleTimeout关键字值的设置。例如，如果服务器范围配置把空闲超时时间设置成5分钟：

```
# SSH1, OpenSSH
IdleTimeout 5m
```

注7：顺便说一下，在本书的合作过程中作者全部使用这种技术。

而你的文件中将自己的账号的空闲超时时间设置成了10分钟：

```
# SSH1, OpenSSH  
idle-timeout=10m ...key...
```

那么不管服务器范围的设置如何，使用该密钥的所有连接的空闲超时时间都是10分钟。

这种特性除了能断连离开用户的连接之外，还可以有其他用途。假设你正在使用一个SSH-1密钥进行自动化处理，比如备份。如果该进程由于某个错误挂起超过了空闲超时时间，系统就会把该进程自动删掉。

8.2.8 禁用转发

即使你允许使用SSH-1协议访问自己的账号，但也可能并不想让自己的账号使用端口转发，充当到其他机器的跳板。要禁用这个特性，就要为该密钥开启 no-port-forwarding 选项：

```
# SSH1, OpenSSH  
no-port-forwarding ...key...
```

同理，如果你不想远程用户使用特定的密钥经由你的账号通往其他计算机，就可以禁用代理转发特性。这是使用 no-agent-forwarding 选项来完成的：

```
# SSH1, OpenSSH  
no-agent-forwarding ...key...
```

警告：（使用这些选项）并没有严格的限制。只要你允许 shell 接入，在这一连接上就可以进行任何操作。通过此连接，用户只需要使用一对普通的客户端程序就可以进行对话，还可以执行诸如端口转发、代理转发以及其他一些你原以为不应该执行的操作。在服务器端配置这些选项，如强制命令或者限制目标账号的 shell 接入时必须小心地限制访问权限。这绝不是危言耸听。

8.2.9 禁用 TTY 分配

通常在通过 SSH-1 协议登录时，服务器都要为登录会话分配一个伪终端（tty）：

```
# 为该客户分配一个tty  
$ ssh1 server.example.com
```

服务器环境会设置一个环境变量 `SSH_TTY`，并将其值设置成分配的 `tty` 名。例如：

```
# 通过SSH-1登录后  
$ echo $SSH_TTY  
/dev/pts/1
```

但是，当运行非交互命令时，SSH 服务器并不会分配一个 `tty` 来设置 `SSH_TTY`：

```
# 不分配tty  
$ ssh1 server.example.com /bin/ls
```

假设你想授权给某个人，让他可以使用 SSH-1 调用非交互式的命令，但是并不运行交互式的登录会话。我们已经看到强制命令是如何限制对特定程序的访问的，但是作为一个附加的安全预防措施，也可以使用 `no-pty` 选项来禁用 `tty` 分配：

```
# SSH1, OpenSSH  
no-pty ...key...
```

现在非交互式命令可以正常工作了，但是交互式会话请求都会被 SSH1 服务器拒绝。如果试图建立一个交互式会话，那么客户端就会打印一条警告消息，例如：

```
Warning: Remote host failed or refused to allocate a pseudo-tty.  
SSH_SMSG_FAILURE: invalid SSH state
```

或者客户端就会一直挂起，或者登录失败。

仅仅是出于兴趣的原因，让我们通过一个简单的实验来观察一下 `no-pty` 对 `SSH_TTY` 环境变量的影响。设置一个公钥并在该公钥之前使用下面的强制命令：

```
# SSH1, OpenSSH  
command="echo SSH_TTY is [$SSH_TTY]" ...key...
```

现在试着使用该密钥进行交互连接和非交互连接，并观察输出结果。交互命令会给 `SSH_TTY` 赋一个值，而非交互命令则不会：

```
$ ssh1 server.example.com  
SSH_TTY is [/dev/pts/2]  
  
$ ssh1 server.example.com anything  
SSH_TTY is []
```

接下来，在该文件中增加 no-pty 选项：

```
# SSH1, OpenSSH
no-pty,command="echo SSH_TTY is [$SSH_TTY]" ...key...
```

并试着进行交互连接。连接（肯定）会失败，SSH_TTY 环境变量也不会被设置：

```
$ ssh1 server.example.com
Warning: Remote host failed or refused to allocate a pseudo-tty.
SSH_TTY is []
Connection to server.example.com closed.
```

即使客户端明确地（用 `ssh -t`）请求分配 tty，no-pty 选项也会禁止分配 tty。

```
# SSH1, OpenSSH
$ ssh -t server.example.com emacs
Warning: Remote host failed or refused to allocate a pseudo-tty.
emacs: standard input is not a tty
Connection to server.example.com closed.
```

8.3 可信主机访问控制

如果使用可信主机认证而不使用公钥认证，那么就可以使用一种有限的每账号配置。特别是，用户可以允许 SSH 根据从系统文件 `/etc/shosts.equiv` 和 `/etc/hosts.equiv` 以及个人文件 `~/.rhosts` 和 `~/.shosts` 得出的客户端的用户名和主机名来访问自己的账号。像这样一行：

```
+client.example.com jones
```

允许用户 `jones@client.example.com` 使用可信主机的 SSH 访问。我们已经介绍过这四个文件的详细内容，在本章中就不再重复介绍了。

对于 `authorized_keys` 文件中的 `from` 选项，使用可信主机认证的每账号配置和使用公钥认证类似，它们都可以拒绝来自特定主机的 SSH 连接，二者之间的区别如下表所示。

特性	可信主机	公钥 from
按照主机认证	是	是
按照 IP 地址认证	是	是
按照远程用户名认证	是	否

特性	可信主机	公钥 from
在主机名和IP地址中可以使用通配符	否	是
登录时必须输入口令	否	是
使用其他公钥特性	否	是
安全性	少	多

若要使用可信主机认证进行访问控制，则必须完全满足以下条件：

- 在编译时和服务器范围的设置文件中同时启用可信主机认证。
- 在服务器范围配置中没有排除想用的客户端主机，也就是没有使用 All-
wHosts 和 DenyHosts。
- 对于 SSH1 来说，*ssh1* 在安装时 setuid 成 root。

不管实际功能如何，可信主机认证要比大家预期的复杂得多。例如，如果你仔细设计 *.shosts* 文件拒绝 *sandy@trusted.example.com* 访问：

```
# ~/.shosts
-trusted.example.com sandy
```

而无意中 *.rhosts* 文件又允许他访问：

```
# ~/.rhosts
+trusted.example.com
```

那么 *sandy* 就可以使用 SSH 访问你的账号。更糟糕的是，即使你没有 *~/.rhosts* 文件，系统文件 */etc/hosts.equiv* 和 */etc/shosts.equiv* 也可以在你的账号中留下安全漏洞，这可非你所愿。不幸的是，使用每账号配置无法避免这个问题。只有编译时配置或服务器范围配置中才可以禁用可信主机认证。

由于这些问题和其他一些严重的固有缺点，我们不推荐使用这种脆弱的可信主机认证（Rhosts 认证）作为每账号配置的一种形式。（缺省情况下这已经被禁用了，我们十分赞同这种处理。）如果需要可信主机认证的特性，建议你使用一种更严格的形式，称为 RhostsRSAAuthentication（SSH1, OpenSSH）或 hostbased（SSH2），它可以通过对主机密钥增加密码验证。[3.4.2.3]

8.4 用户 rc 文件

SSH 服务器会在每个 SSH 连接到达时调用 shell 脚本 `/etc/sshrc`。用户可以在自己的账号中定义一个类似的脚本，`~/.ssh/rc` (SSH1, OpenSSH) 或 `~/.ssh2/rc` (SSH2)，对登录到账号的每个 SSH 连接都调用该脚本。如果你自己定义的脚本文件存在，`/etc/sshrc` 就不会执行。

SSH `rc` 文件十分类似于 shell 的启动文件（例如，`~/.profile` 或者 `~/.cshrc`），但是该文件只有在使用 SSH 访问你的账号时才会执行。交互式登录和远程命令都会运行该文件。用户可以把在使用 SSH 而不是普通登录访问你的账号时想要执行的任何命令都放入该脚本中。例如，可以在该文件中运行并加载自己的 `ssh-agent`:

```
# ~/.ssh/rc, 假定登录shell是C shell
if ( ! $SSH_AUTH_SOCK ) then
    eval `ssh-agent`
    /usr/bin/tty | grep 'not a tty' > /dev/null
    if ( ! $status ) then
        ssh-add
    endif
endif
```

和 `/etc/sshrc` 类似的是，你个人的 `rc` 文件在到达的连接请求执行 shell 或者远程命令之前执行。不同之处在于，`/etc/sshrc` 总是由 Bourne shell (`/bin/sh`) 处理，而你的 `rc` 文件是由你的账号登录 shell 处理的。

8.5 小结

每账号配置让你能够通知 SSH 服务器对你的账号进行区别对待。使用公钥认证，可以根据客户端密钥、主机名或者 IP 地址来允许或限制连接。使用强制命令，可以限制在你的账号中客户端能执行的程序集。还可以禁用不想要的 SSH 特性，例如，端口转发、代理转发以及 tty 分配。

使用可信主机认证，可以允许或禁止特定的主机或远程用户访问自己的账号。这种方法使用了 `~/.shosts` 或（优先级稍微低一点的）`~/.rhosts` 两个文件。但是，这种机制的安全性和灵活性都不如公钥认证。

本章内容：

- 转发是什么？
- 端口转发
- X 转发
- 转发安全性：TCP-wrappers 及 libwrap
- 小结

第九章

端口转发与 X 转发

SSH 的一个主要优点就是透明性。终端会话建立起之后就一直受到 SSH 的保护，运作方式就与普通的不安全（比如由 *telnet* 或 *rsh* 创建的）会话完全相同。不过，这全靠 SSH 在幕后保护会话过程，为其提供强有力的认证、加密及完整性检测。

然而在下面的情况下，透明性很难保证。网络防火墙可能会挡在中间，阻碍你进行某些必要的网络传输。公司的安全策略可能不允许你把 SSH 密钥存储在某些主机上。或者，你也可能需要在原本安全的环境中运行一些不安全的网络应用程序。

本章将讨论 SSH 的一个重要功能，称为转发（forwarding）或者隧道传输（tunneling），以解决与透明性有关的几个问题：

保护其他 TCP/IP 应用程序

SSH 能为另外一个应用程序的数据流提供透明的加密措施。这称为端口转发（port forwarding）。

保护 X Window 应用程序

通过 SSH 可以调用远程主机上的 X 程序，令其在本地显示设备上安全地显示（通常 X 中的这项功能是不安全的）。这称为 X 转发（X forwarding），是端口转发的特例，SSH 为其提供额外支持。

SSH 转发并不是完全透明的，因为其发生在应用程序层，而不是网络层。应用程序必须经过设置才能参与转发，还有一些协议在转发过程中会产生问题（明显的例子

是 FTP 的数据通道)。不过对大多数一般的情况来说,一旦安全隧道建立起来,参与的应用程序对用户提供的还是通常的操作方式。如果要在应用程序层实现完全透明,必须在网络层采取一些措施,如,IPSEC[1.6.4]或专用的 VPN(虚拟专用网,Virtual Private Network),这既可以是主机上的软件,也可以是专门的路由器,提供这类产品的商家很多。虽然 VPN 是更加完整的解决方案,但其实施过程与 SSH 转发相比,付出的劳动与代价也大得多。

所以,本章中的“透明”都是指“只要完成一些设置,就对应用程序透明”。

警告: 本章会讨论用 SSH 转发技术实现原本被防火墙阻隔的网络传输。如果运行得当,这就是一项完全合法而且完全够用的安全措施:因为防火墙阻断的只是未经授权的传输,而 SSH 转发则帮助授权用户绕过这些限制。不过不要忘记,你所绕过的限制是有其存在的理由的。请遵循我们提出的指导方针,这样才能确保 SSH 转发是安全的。还要留心使用转发是否违反公司政策,能这样做并不一定意味着这样做是对的。若有疑问,请向系统管理员咨询。

9.1 转发是什么?

转发是一种与其他网络应用程序交互的方式,如图 9-1 所示。在 SSH 连接的一端,SSH 截取其他程序的服务请求,将其通过加密的 SSH 连接发送,传给另一端的适当服务接收者。多数情况下,这一过程对连接的两端而言都是透明的:每端都相信它在与对方直接对话,丝毫意识不到转发的存在。SSH 转发还有更强大的功能,可以实现几种原来无法实现的通信。

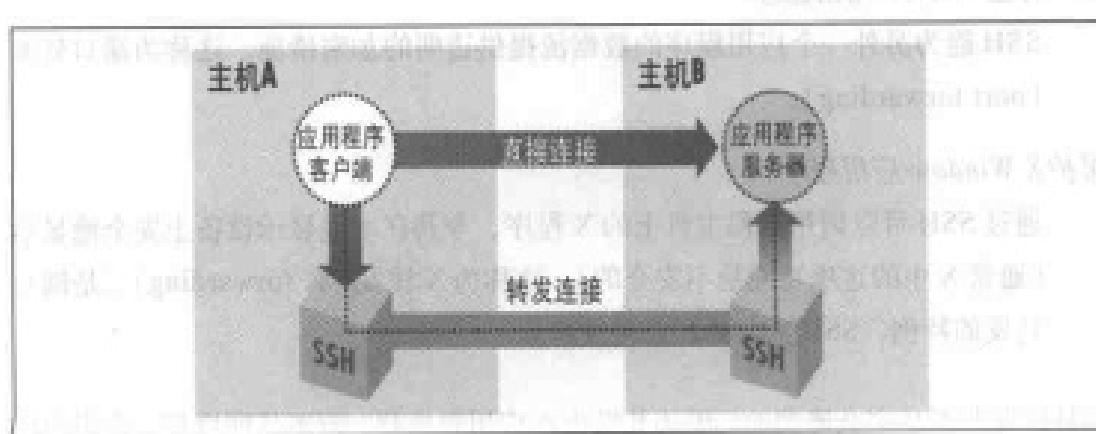


图 9-1: SSH 转发

转发不是一种新的概念。网络上的终端连接（比如说用 *telnet*）的基本操作也是一种转发。在 *telnet* 连接中，你在一端，远程 shell 在另外一端，双方就像是直接由串行电缆连接起来一样。不过，*telnet* 客户端和服务器在中间相互配合，完成数据来回转发的工作。*SSH* 转发基本与此相同，此外还运用了一些增强数据安全性的技术。

我们已经见过另一种类型的 *SSH* 转发，代理转发。[\[6.3.5\]](#) 使用公钥认证时，它令我们能从一台机器，经过第二台机器，建立到第三台机器的 *SSH* 连接，而不用把私钥安装在第二台机器上。为此，*SSH* 服务器要扮演 *SSH* 代理的角色，与另一个远程代理之间实现数据的透明转发。这一过程在 *TCP* 端口转发和 *X* 转发过程中同样适用，因为 *SSH* 服务器得装扮成另一个网络应用程序，同时保持其透明性。

9.2 端口转发

SSH 使用的传输机制是 *TCP/IP*，通常使用的都是服务器的 *TCP* 端口 22，并对经过连接传输的数据进行加密/解密操作。下面介绍的功能非常好，它能用 *SSH* 对其他应用程序在别的 *TCP* 端口上建立的 *TCP/IP* 传输进行加密和解密。这一过程称为端口转发，其绝大部分操作是透明的，功能非常强大。*Telnet*、*SMTP*、*NNTP*、*IMAP* 和其他一些基于 *TCP* 的不安全协议都可以变得安全，只要将其连接通过 *SSH* 转发即可。端口转发有时又叫做隧道传输，这是因为 *SSH* 连接提供了一条安全的“隧道”，其他 *TCP/IP* 连接可由此经过。

假设你家里有一台电脑 H，上面运行了某个支持 *IMAP* 的 email 阅读器，你想连接机器 S 上的 *IMAP* 服务器读取并发送邮件。通常，这个连接是不安全的，你的邮件账号密码在邮件程序和服务器之间都是以明文形式传送的。运用 *SSH* 端口转发技术，你可以透明地将 *IMAP* 连接（建立于服务器 S 的 143 端口）重新路由，令其经过 *SSH*，并安全地对连接上的数据进行加密（注 1）。*IMAP* 服务器主机必须运行 *SSH* 服务器，这样才能真正使用端口转发保护连接。

注 1： 我们给出的端口转发例子保护的是 *IMAP* 连接，但并不真正保护 email 消息。在到达 *IMAP* 服务器之前，这些消息还经过其他 *IMAP* 服务器，在传输过程中可能泄密。要保障端到端邮件传输的安全性，你与收信人之间应该用 PGP 或 S/MIME 这类的工具对消息本身进行签名和/或加密。

简而言之，只需对程序进行最少的设置修改，就可以把任意的TCP/IP连接重定向到SSH会话中，从而使用SSH端口转发对其进行保护。如果设置得当，SSH甚至可以让连接安全通过防火墙。只要一旦开始使用端口转发对通信进行保护，你就会觉得奇怪，为什么以前竟然能忍受没有端口转发的日子。现在你可以实现以下功能：

- 越过阻止直接连接的防火墙，访问多种TCP服务器（如，SMTP、IMAP、POP、LDAP等等）。
- 同时保护这些TCP服务器的会话，防止密码泄密或被修改，保护其他原本在会话中明文传输的内容。
- 将FTP会话的控制连接打入隧道中，对用户名、密码和命令进行加密。（不过，通常不可能保护载有文件内容的数据通道。[11.2]）
- 即使你从ISP之外的网络连接进来，而ISP禁止你从当前位置传送邮件，你也可以访问到SMTP服务器，并发送邮件。

注意：一般意义上讲，SSH端口转发是TCP使用的一种通用代理机制，而且只能用于TCP/IP协议。（要了解TCP的基本概念，参看“TCP连接”。）如果协议不是基于TCP的，比如基于UDP的DNS、DHCP、NFS和NetBIOS（注2），或者非IP类协议，如AppleTalk或Novell的SPX/IPX，就不能使用端口转发机制。

9.2.1 本地转发

在前一个例子中，我们在主机上运行了一个IMAP服务器，在家里的主机H上运行了一个email阅读程序。现在我们想使用SSH保护IMAP连接。现在我们深入探讨一下这个例子。

注 2：此处并非十分严密。DHCP是完全基于UDP的，所以SSH端口转发对其不会有任何作用，不过其他几个或者根据使用目的的不同，同时使用了TCP和UDP，或者通常情况下使用UDP，但有时还可以设置基于TCP运行。无论如何，就最一般的情况而言，SSH无法对这些协议实施转发。

TCP 连接

要理解端口转发，很重要的一点是知道一些TCP(传输控制协议)的细节。TCP是Internet的一个基础组成部分。它构建在IP协议之上，是很多应用层Internet协议使用的传输机制，例如FTP、Telnet、HTTP、SMTP、POP、IMAP以及SSH本身。

TCP为传输提供强有力的保障。TCP连接是一条建立在两个通信对象之间的虚拟全双工线路，看起来就像一条双向管道。任何时候每一端都可以向管道中写入任意字节的数据，这些数据保证能按顺序原封不动地到达另一端。不过，设计这些保障机制，是为了解决网络中的传输问题，比如路由时绕过失效链路，再比如噪声污染数据或由于网络临时拥塞而导致丢包时的重传。但它对于攻击者蓄意窃取连接或在传输过程中偷换数据则无能为力。在这方面，SSH给TCP提供了保护。

如果应用程序不需要这样强有力的保障，或者不想经常在这上面付出代价，那么用另一个协议UDP(用户数据报协议)通常也能满足要求。UDP是一种面向报文(package)的协议，而不是面向连接的协议，它不保证传输一定成功到达，也不能保障报文的顺序。下面这些协议都运行于UDP协议之上：NFS、DNS、DHCP、NetBIOS、TFTP、Kerberos、SYSLOG和NTP。

当程序向建立一条到某个服务的TCP连接时，它需要两部分信息：目的主机的IP地址，标识所请求服务的方法。TCP(和UDP)使用一个正整数来标识一项服务，这个正整数称为端口号(port number)。例如，SSH使用22端口，telnet用23端口，IMAP用143端口。端口号可以让同一个IP提供多个服务。

一个IP地址和一个端口号合称为一个套接字。例如，你运行telnet连接到IP地址为128.220.91.4的机器的23端口上，这个套接字就记为“(128.220.91.4, 23)”。简单说来，TCP连接的目的地就是一个套接字。在连接源(客户端程序)处也有一个套接字，这个连接是一个整体，使用这对源与目的套接字就可以完全定义这个连接。

— 待续 —

成功连接某一套接字的前提是这个套接字必须正在“监听”。也就是说，运行于目的主机上的某个程序已经通知TCP接受到该端口的连接请求，并将其传递给这个程序。如果你曾经试着建立TCP连接，收到的响应是“connection refused”，这就意味着远程主机已经启动，正在工作，但目标套接字上没有监听。

客户端程序怎样才能知道服务器在哪个目的端口号上监听呢？许多协议的端口号都是标准的，由互联网号码分配机构（IANA，Internet Assigned Numbers Authority）负责分配。（IANA 的完整端口号列表见：<http://www.isi.edu/in-notes/iana/assignments/port-numbers>。）比如，给NNTP（Usenet新闻组）协议分配的TCP端口号是119。因此，新闻服务器都在119端口监听，新闻阅读器（newsreader，客户端）通过119端口连接服务器。更具体地说，如果某个新闻阅读器访问的新闻服务器的IP地址是10.1.2.3，那么它就应该请求建立一条到套接字（10.1.2.3, 119）的TCP连接。

端口号不一定非要在程序里硬性指定。许多操作系统定义了TCP协议名及端口号的对照表，因此应用程序可以不通过端口号，而是通过协议名访问协议。程序就可以使用协议名查找端口号。在Unix中，此表通常保存在/etc/services文件中，或NIS服务映射表（NIS Services Map）中，用库函数getservbyname()和getservbyport()及相关函数可查询此表。有些环境允许服务器通过某种名字服务动态注册其监听的端口，如AppleTalk的名字绑定协议（Name Binding Protocol）或DNS中的WKS和SRV记录。

到目前为止，我们已经讨论了TCP客户端程序建立连接时，对应的TCP服务器使用的端口号。这称为目的端口号。客户端也用了一个端口号，称为源端口号，这样服务器才能向客户端传递数据。客户端IP地址和源端口号合称为客户端套接字。

源端口号并非标准的，这一点与目的端口号不同。实际上在大多数情况下，客户端和服务器都不关心客户端使用的源端口号到底是多少。通常，客户端会让TCP选择一个尚未使用的端口作为其源端口。（不过Berkeley的r命令是与源端口号有关的。[3.4.2.3]）如果用netstat -a或lsof -i tcp检查机器中现有的

TCP连接，你将看到，连至通用服务的知名端口号的连接，其另一端的源端口号都是一些很大的、像是随机选择的数字。这些源端口是从建立连接的机器上尚未分配的TCP端口中选择出来的。

TCP连接一旦建立，就可以由源套接字与目的套接字的组合完全确定。因此，多个TCP客户端可以连接到同一个目的套接字。如果连接来自于不同主机，那么其源套接字的IP地址部分就不相同，由此就可以区分这些连接。如果连接来自于同一台主机上的不同应用程序，那么该主机上的TCP协议就可以保证连接具有不同的源端口号。

IMAP使用TCP的143端口；这意味着IMAP服务器会在主机服务器的143端口上监听连接。要使IMAP连接通过SSH隧道，就得在主机H上选择一个本地端口（1024至65535），将其转发至远程套接字（S. 143）。假设随机选择本地端口2001，下面是创建隧道的命令（注3）：

```
$ ssh -L2001:localhost:143 S
```

-L表明是本地转发，此时TCP客户端与SSH客户端同在本地主机上。后面接着三个值，由冒号分开，分别表示：需要监听的本地端口（2001）、远程主机名或IP地址（S）及远程的转发目标端口号（143）。

上面这个命令可完成登录到S的功能，如果仅输入ssh S也能达到同样的效果。然而，现在的这个SSH会话同时将H的2001端口转发到S的143端口；在退出会话之前转发一直有效。使用该隧道的最后一个步骤是，告知email阅读器使用被转发的端口。通常email程序连接服务器的143端口，即套接字（S, 143）。现在要令其连接本地主机H自己的2001端口，也就是套接字（localhost, 2001）。所以现在，数据传输是这样进行的：

1. 家中的主机H上的email阅读器向本地端口2001发送数据。
2. H上的本地SSH客户端读2001端口，加密数据，将其通过SSH连接传至S上的SSH服务器。

注3：也可以把上面的localhost换成“S”，执行命令的是ssh -L2001:S:143 S。稍后将讨论为什么尽可能用localhost比用S好。

3. S 上的 SSH 服务器给数据解密，将其传给在 S 的 143 端口上监听的 IMAP 服务器。
4. 使用相反的步骤把数据从 IMAP 服务器把数据传回主机 H。

只有建立 SSH 时才能定义端口转发。据我们所知，没有任何 SSH 版本可以给一个已经存在的 SSH 连接增添转发，不过 SSH 协议本身并没有与此相抵触之处，也许将来有一天这会成为 SSH 中一项有用的功能。创建本地转发时也可以不用 -L 选项，而在客户端配置文件中用 LocalForward 关键字。

```
# SSH1, OpenSSH
LocalForward 2001 localhost:143
# 仅对 SSH2
LocalForward "2001:localhost:143"
```

请留心语法结构上的细微差别。在 SSH1 与 OpenSSH 中，参数有两个：本地端口号和远程套接字，后者表示为 *host:port*。对 SSH2 而言，整个参数表达式与命令行中的一样，但必须由双引号引起来。如果忘了用双引号，*ssh2* 虽然不会出错，但是也不会转发端口。

从家中的主机 H 向 IMAP 服务器 S 建立连接的过程举例如下：

```
# SSH1, OpenSSH
Host local-forwarding-example
HostName S
LocalForward 2001 localhost:143
# 运行于家中的主机 H 上
$ ssh local-forwarding-example
```

9.2.1.1 本地转发与 GatewayPorts

在 SSH1 和 OpenSSH 中，缺省情况下只有运行 SSH 客户端的那台主机才能连接本地的转发端口。这是由于对转发端口而言，*ssh* 仅仅在本机的回环地址上监听；也就是说，绑定的是 (*localhost*, 2001) 也可以写成 (127.0.0.1, 2001)，而不是 (H, 2001)。所以上例中，只有主机 H 可以使用该转发；如果从其他机器上连接 (H, 2001) 只会得到 “connection refused.” 错误。不过，SSH1 和 OpenSSH 的 *ssh* 命令还有一个 -g 选项，它可以取消这种限制，让所有主机都能连接到本地转发端口。

```
# SSH1, OpenSSH
$ ssh1 -g -L<localport>:<remotehost>:<remoteport> hostname
```

客户端的设置关键字 `GatewayPorts` 也对此进行控制；某缺省值为 “no”，`GatewayPorts=yes` 与 `-g` 开关的作用相同：

```
# SSH1, OpenSSH  
GatewayPorts yes
```

之所以缺省状态会关掉 `GatewayPorts` 和 `-g` 是有其原因的：打开它将对安全构成威胁。[9.2.4.2]

9.2.1.2 远程转发

远程转发端口与本地转发几乎完全相同，只是方向相反。此时 TCP 客户端在远程，服务器在本地，转发连接由远程主机发起。

下面我们继续介绍前面的例子，假设你现在已经登录进服务器 S，IMAP 服务器已在其上运行。现在可以创建一条从远程终端到 IMAP 服务器 143 端口的安全隧道。再次随机选择一个要转发的端口号（比方还是 2001），并创建隧道：

```
$ ssh -R2001:localhost:143 H
```

`-R` 表示远程转发。后面有三个参数，同样由冒号分开，不过意思稍有不同。第一个参数现在是要转发的远程端口（2001），接下来是机器名或 IP 地址（`localhost`），然后是端口号（143）。现在 SSH 就可以把连接从（H, 2001）转发到（`localhost`, 143）。

此命令执行之后，一条从远程主机 H 的 2001 端口到服务器 S 的 143 端口的安全隧道就建立起来了。现在，H 上的任何程序都可以通过连接（`localhost, 2001`）来使用此安全隧道。与前面的情况类似，这条命令同时也像 `ssh H` 那样，在远程主机 H 上建立了 SSH 终端会话。

也可以像本地转发那样，在客户端配置文件中用一个关键字来建立远程转发。`RemoteForward` 关键字与 `LocalForward` 用法相同，在 SSH1 与 SSH2 之间的语法差别也一样。

```
# SSH1, OpenSSH  
RemoteForward 2001 S:143  
  
# 仅对 SSH2  
RemoteForward "2001:S:143"
```

下面以 SSH2 格式的配置文件为例，说明定义转发的步骤：

```
# 仅对 SSH2
remote-forwarding-example:
Host H
RemoteForward "2001:$:143"

$ ssh2 remote-forwarding-example
```

注意：读者也许会以为上一节讨论的**GatewayPorts**与远程端口转发作用相同吧。作为功能本身应该是这样，不过实际情况并非如此。客户端原本应该能用某种方式与服务器交换某个给定转发的这项参数，不过 SSH1 协议尚未包含此功能。SSH1 协议缺乏提示这一不同点的能力；SSH2 可以，不过目前的客户端通常只会请求监听所有的地址。在 SSH1 与 SSH2 中，远程转发端口总在所有网络接口上监听，也总是接受来自任何地方的连接请求（例外情况请参看本章后面的 **TCP-wrappers**）。OpenSSH 服务器能识别 **GatewayPorts** 选项，并将其运用在该服务器建立的所有远程转发之上。

9.2.2 多连接引发的问题

如果在配置文件中使用了**LocalForward**或**RemoteForward**，就可能引发一些不可思议的问题。假设在配置文件中有一段语句，其设置把连接从本地端口 2001 转发至 IMAP 服务器：

```
# 用于说明的 SSH1 语法
Host server.example.com
LocalForward 2001 server.example.com:143
```

如果连接一次，这样配置是可以正常工作的：

```
$ ssh server.example.com
```

但如果想同时建立到服务器 *server.example.com* 的第二条 *ssh* 连接（这可能是在桌面的另一个窗口中运行了不同的程序）就会失败：

```
$ ssh server.example.com
Local: bind: Address already in use
```

发生这种现象的原因是什么？是由于配置文件企图再次转发 2001 端口，但却发现该端口已被第一个 *ssh* 实例占用了（“绑定”监听）。我们需要一种既可以建立连接，又不受端口转发影响的办法。

SSH1（但不包括 OpenSSH）提供了一种解决方法，即使用客户端配置关键字 ClearAllForwardings。单看名字，你可能会以为它把所有已经存在的转发都终止了，其实不然。这个参数更像是取消当前 *ssh* 命令中指定的所有转发功能。前例中，可以这样不使用转发连接 *server.example.com*:

```
# 仅对 SSH1
$ ssh1 -o ClearAllForwardings=yes server.example.com
```

原先第一条命令建立的隧道会一直存在，但 ClearAllForwardings 阻止了第二条命令建立隧道的企图。下面这条自相矛盾的命令深入说明了这一点：

```
$ ssh1 -L2001:localhost:143 -o ClearAllForwardings=yes mymachine
```

-L 选项试图建立转发，但 ClearAllForwardings 将其终止。此命令与下面的命令在功能上完全相同：

```
$ ssh1 mymachine
```

当然，ClearAllForwardings 也可以放置在配置文件中。不过好像在命令行中用处更大些，因为这样可以随时应用，而不必编辑文件。

9.2.3 本地转发与远程转发的比较

本地转发与远程转发之间的区别是很微妙的。要想知道在什么情况下用哪一种转发有时候会令人困惑。一个快速的办法是查看一下 TCP 客户端程序。

注意：如果（将转发其连接的）TCP 客户端程序运行于本地的 SSH 客户端机器上，就用本地转发。否则，客户端程序就是与远程 SSH 服务器运行于同一台机器上，应该用远程转发。

本节致力于分析转发的详细过程，力图让读者理解这一规则的来由。

9.2.3.1 基本原理

可能是术语太多的缘故，本地转发与远程转发会令人迷惑。在给定的某种端口转发环境中，有两个客户端和两个服务器。SSH 客户端及其服务程序（即 *ssh* 与 *sshd*），还有 TCP 应用程序客户端及服务器，要用端口转发来保护的是后者的连接。

SSH会话有其创建方向。就是说，在某一台机器上运行SSH客户端，用其发起一个会话，与另一台机器上的SSH建立连接。与此类似，被转发的连接创建时也是有方向的：在某台机器上运行客户端程序，与另一台机器上的服务器建立连接。这两个方向可能不一致。这就是本地转发和远程转发的区别。我们将介绍一些术语，并会用图表辅助理解。

首先，两台主机A和B分别运行应用程序客户端和服务器（见图9-2）。

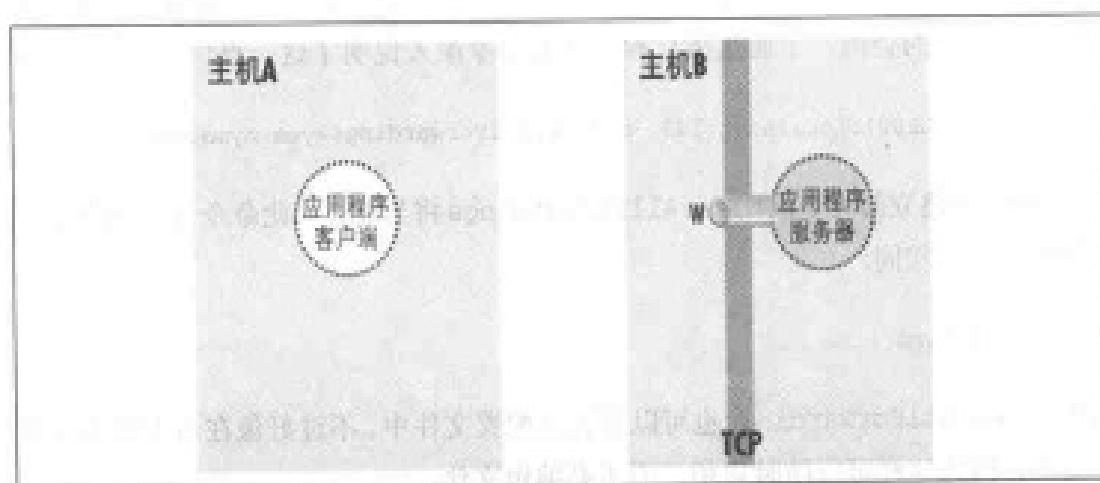


图 9-2：应用程序客户端及服务器

应用程序服务器在某个已知端口W上监听到达的客户端连接。没有SSH时，我们可以告诉应用程序客户端其服务程序在主机B的W端口上。客户端会直接连接服务器，所有的应用程序协议数据都将在网络上以明文传输（见图9-3）。

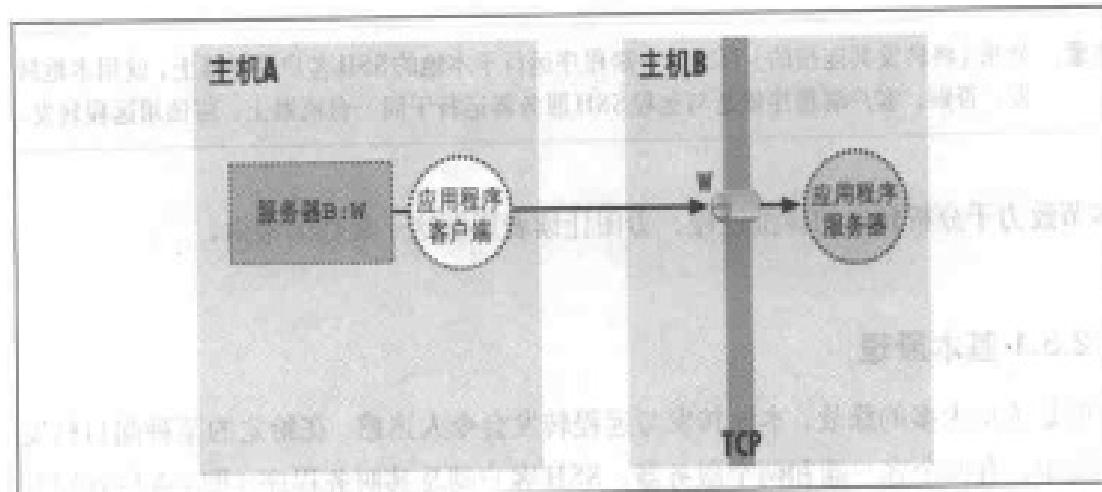


图 9-3：客户端直接连接服务器（无转发）

我们在这两台主机之间创建一个 SSH 会话，以便用转发来保护应用程序协议数据。建立 SSH 会话时，在应用程序客户端这边（主机 A）选择一个未用的端口号 P，并请求使用从套接字（A, P）至（B, W）的 SSH 端口转发。会话一旦建立，A 上的 SSH 进程就会在端口 P 监听到达的 TCP 连接请求了。此时告诉客户端服务器在（A, P），而不是（B, W），端口转发的准备工作就完成了。

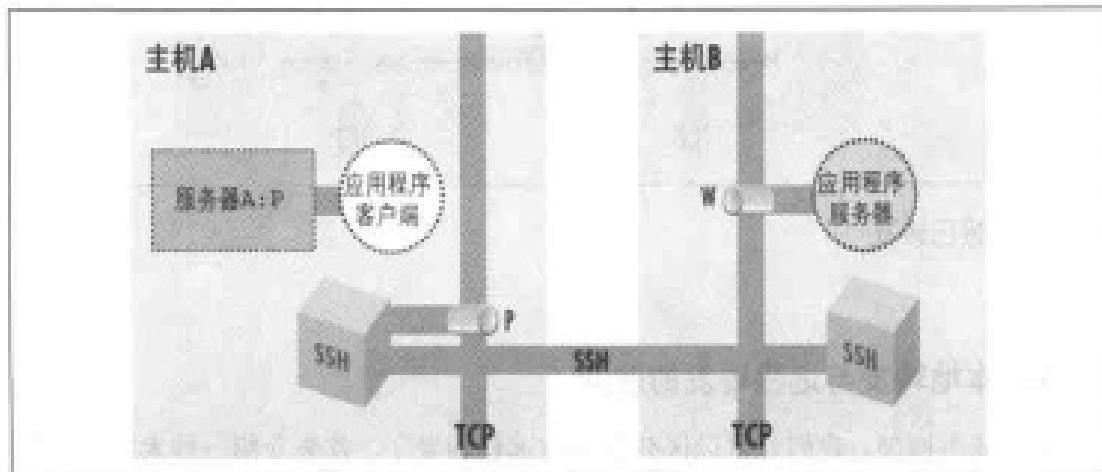


图 9-4：端口已转发

目前存在两个相互合作的 SSH 进程，两者之间已建立起加密的 SSH 会话；但是我们依然没有区别 SSH 客户端和服务器。SSH 在此会话中创建多个通道或者逻辑数据流，以加载数据。在通过 SSH 进行的交互登录或远程命令执行过程中，通道负责携带并区分输入、输出与错误信息流；与此类似，每次使用某个端口转发，也都要创建新通道，以便在受保护的 SSH 会话中加载转发数据。

图 9-5 表明，目前如果应用程序客户端要连接其服务器，它连接的是 SSH 的监听进程（1）。SSH 的监听注意到此连接请求，并接受它。然后通知对方 SSH 进程，该端口转发将开始新的实例，两者合作建立起为此转发实例携带数据的新通道（2）。最后，对方 SSH 进程创建到转发目标的 TCP 连接；应用程序服务器在（B, W）上监听（3）。一旦连接成功建立，端口转发的实例就到位了。两个 SSH 进程协作，在 SSH 会话中的通道上往返传送应用程序客户端及服务器发送的所有数据。这样，应用程序既可以相互通信，它们在网络上的活动也会受到保护（见图 9-5）。

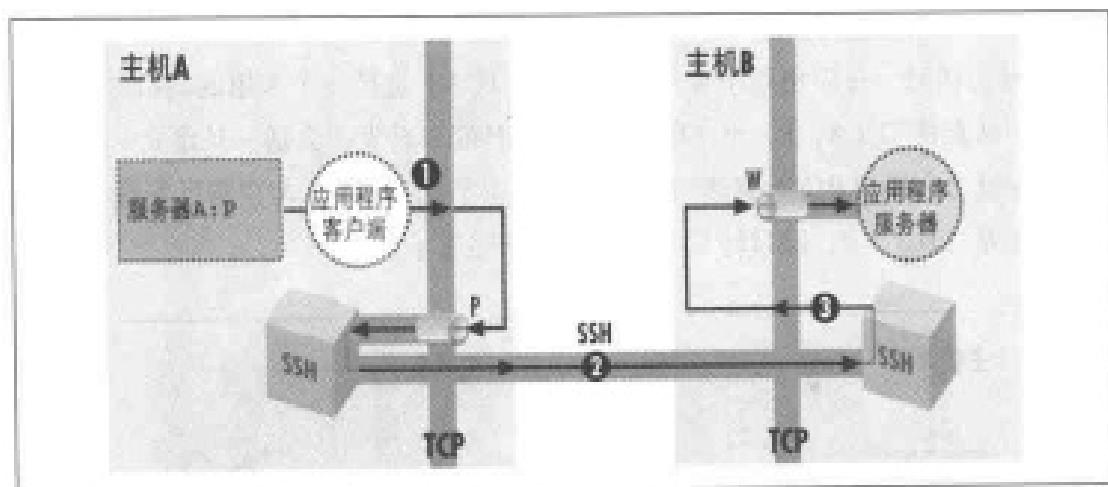


图 9-5：连接已转发

9.2.3.2 本地转发与远程转发的差异

有了这个基本框架，我们就可以区分本地与远程转发了。首先介绍一些术语。上节对端口转发过程概括地进行了介绍，我们已经知道有一个 SSH 进程负责监听连接，另一个随时准备对前一个接受的连接作出响应，发起连接，这样就完成了整个转发过程。我们称第一个进程为 SSH 会话针对该转发的监听端（listening side）；另一个为连接端（connecting side）。例如，图 9-4 中主机 A 在监听端，而 B 在连接端。请注意，这些术语并不是彼此排斥的。既然单个 SSH 会话中可以同时存在多个转发，那么会话的同一端可能在某些转发中是监听端，而同时在另外的转发中又是连接端。但对任何具体的转发而言，则非此即彼。

回忆一下在上一节中我们并未区分两个 SSH 进程中哪个是 SSH 客户端，哪个是 SSH 服务器，只是简单地称其为相互协作的 SSH 进程。现在将区分它们，并简单说明本地转发与远程转发的差异：

- 在本地转发中（见图 9-6），应用程序客户端与监听端同 SSH 客户端在一起。应用程序服务器与连接端同 SSH 服务器在一起。
- 远程转发的情况恰好相反：应用程序客户端与监听端同 SSH 服务器在一起，应用程序服务器与连接端同 SSH 客户端在一起。

因此，正如本节开始时所述：应用程序客户端位于 SSH 连接本地时使用本地转发，在远程时使用远程转发。

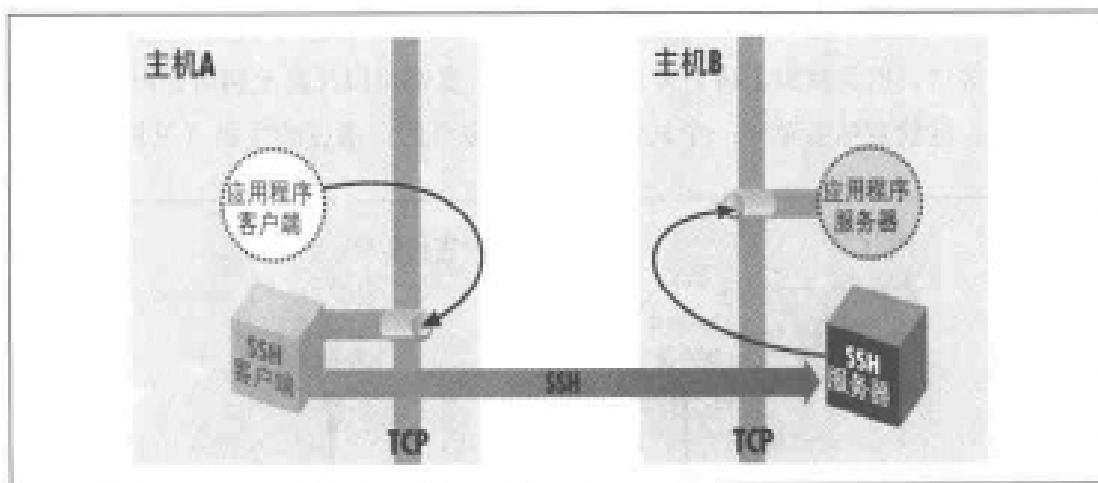


图 9-6：本地转发

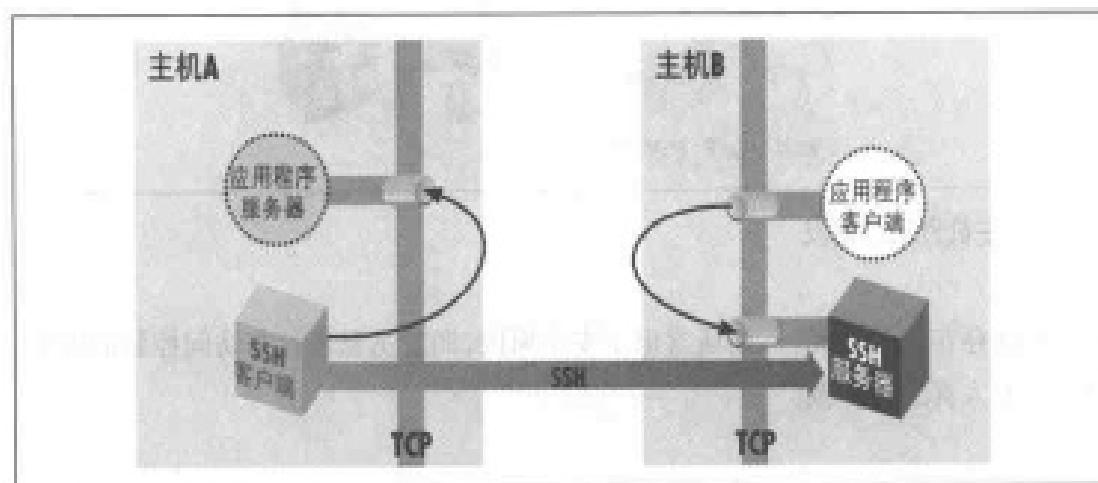


图 9-7：远程转发

对称于“本地”使用一对主机，我们通过将端点的连接替换成反向连接，从而实现对称。这样就将要连接到的本地端点替换成远端端点，从而在两个不同的主机上运行。

9.2.4 主机外转发

端口转发讨论到目前为止，应用程序客户端和服务器都位于运行 SSH 会话的机器上。这表现在我们都使用“localhost”来命名转发的目标套接字：

```
$ ssh -L2001:localhost:143 server.example.com
```

既然应用程序服务器与 SSH 端口转发的连接端在同一台主机上，那么目标主机就可以是“localhost”。不过，应用程序客户端与 SSH 监听端之间，还有应用程序服务器与 SSH 连接端之间，都是 TCP 连接。为方便起见，TCP 应用程序都允许在同一

主机上的两个套接字之间建立连接。数据只是在两个进程之间传输，并未真正通过任何网络接口。然而原则上讲，客户端或者服务器都可以（甚至两者同时）在不同的机器上，这就意味着单一一个转发可能涉及多至四台潜在的主机（见图 9-8）。

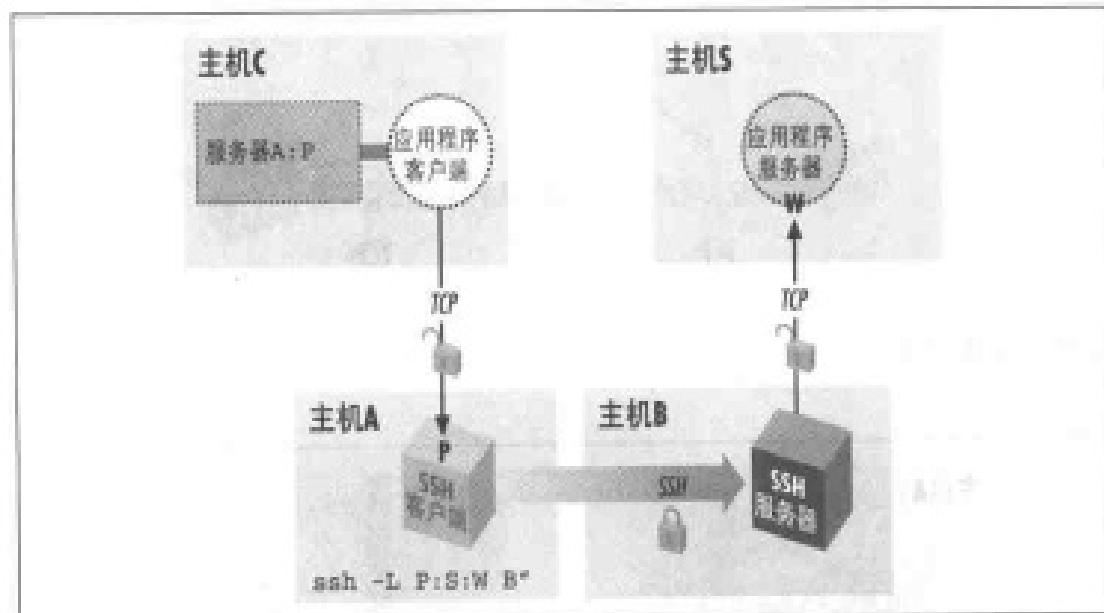


图 9-8：主机外端口转发

尽管可能存在这种情况，然而通常出于安全原因，即数据私密性与访问控制的需要，也不会有人真想这样做。

9.2.4.1 保护数据的私密性

如图9-8所示，转发数据经过的完整路径包括三个TCP连接。但是只有第二个连接，即两个SSH进程之间的那个，是作为SSH会话中的通道而受到保护的。另外两个只是简单的TCP连接。通常，每一个这样的连接都在同一主机上，因此不会在网络上受到窃听或干扰，所以，整个转发路径都是安全的。但如果两个连接中的任意一个处于不同主机之间，其数据传输过程将受到威胁。

9.2.4.2 访问控制与回环地址

主机外转发的另一个安全问题与监听端有关。简而言之，转发的监听端不存在访问控制机制，所以入侵者能得到其控制权。为了说明此问题，必须先讨论一下主机的回环地址。

一台运行IP协议的主机，除了其可能具有的所有物理网络接口之外，还有一个虚拟回环接口。这是一个软件概念，不与任何的网络硬件相对应。然而回环地址的样子和真的接口完全一样，也可以作出响应。Unix通常称之为`lo0`，用`ifconfig`命令可以看到：

```
$ ifconfig -a  
...  
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232  
      inet 127.0.0.1 netmask ff000000
```

回环接口通回主机本身。在回环地址上传递的报文立即在回环接口上形成到达的数据包，IP协议将其接收，并作为发往本地主机的数据处理之。

回环接口总是在同一个IP地址上：127.0.0.1，称为回环地址（注4），本地名字服务给这个地址起的名字是“`localhost`”。无论主机真正连接的IP地址是什么，也不管主机是否真的有实际的网络连接，这一机制都可以为进程间通过本机IP协议相互通信提供可靠途径——因为用知名回环地址总可以访问本地主机。

回环地址有意设计成主机专有的。一台机器不能访问别的机器的回环地址。回环地址127.0.0.1是所有IP主机的标准地址，只要连接127.0.0.1就能使主机与其自身对话（另外，Internet路由不处理回环地址）。

9.2.4.3 接口监听（绑定）

主机监听某TCP端口时，就建立起TCP连接的一个潜在端点。但TCP连接的端点是套接字，而套接字是(`address, port`)，不是(`host, port`)。监听必须建立在具体的套接字之上，这样就必须关联某个特定的地址，即与主机的特定网络接口相联系。这个过程称为接口绑定(binding)（注5）。除非特别指定，否则，如果请求监听某个端口，TCP就绑定所有的主机接口，并会接受向其中任何一个的连接请求。这对

注4：实际上整个网段127.0.0.0/8由一千六百万个地址构成，这些地址都是本地主机的保留地址。常用的只有127.0.0.1，不过也见过有些设备将另外的地址用于其他特殊目的，如，终端服务器或路由器上的“reject”接口。

注5：此名词来自Berkeley socket库的`bind`例程，该例程通常用于建立关联。

于服务器而言通常是正常的行为。因为它并不关心主机有多少网络接口：无论连接请求是发向哪个主机地址的，服务器都接受到其监听接口的连接。

不过，可以想像这对于SSH端口转发意味着什么。转发的监听端上根本不存在任何认证和访问控制机制；只是简单地接受任意连接，并将其转发。如果监听端为转发端口绑定所有的主机接口，这就意味着只要能访问正在监听的主机，任何人（甚至是整个Internet）都能够使用该转发。这显然不是什么好事。SSH解决此问题的办法是，转发监听端缺省情况下只绑定回环地址。这样，就只有同一台主机上的其他程序可以连接转发套接字。这对于PC或其他单用户机上的端口转发来说已经相当安全了，不过在多用户主机上仍然存在安全问题。例如，如果用户有足够的知识，就可以随意连接Unix主机上所有正在监听的套接字，并查看其内容。如果在Unix主机上使用端口转发，一定要牢记这一点！

如果转发端口允许进行主机外连接，那么可以用-g开关或GatewayPorts选项将监听绑定在所有接口上，如前例：[9.2.4]

```
$ ssh -g -L P:S:W B
```

但是别忘记这是一个安全隐患。在这种情况下，TCP-wrapper可以在端口转发之上实施更多的控制，本章稍后将讨论它。

9.2.5 绕过防火墙

现在我们介绍一个更复杂的端口转发的例子。图9-9的公司环境与图6-5中讨论代理转发时相同。[6.3.5] 家中的机器H通过堡垒主机B访问公用机W，你想从家里访问公司的email。W上运行IMAP服务器，H上装有能识别IMAP协议的email阅读器，但你无法直接将两者连通。家中的IMAP客户端试图直接用TCP连接W上的IMAP服务器，但防火墙将连接阻断了。既然主机B在防火墙内部，其上运行有SSH服务器，那就应该存在某种方式，将所有这些东西相互关联起来，以建立从H到W的IMAP连接。

用端口转发可以解决这个问题。和以前一样，IMAP服务器在143端口，选择一个随机的本地端口号，2001。不过，这一次创建转发的命令稍微有点不同：

```
# 在家里的主机H上执行  
$ ssh -L2001:W:143 B
```

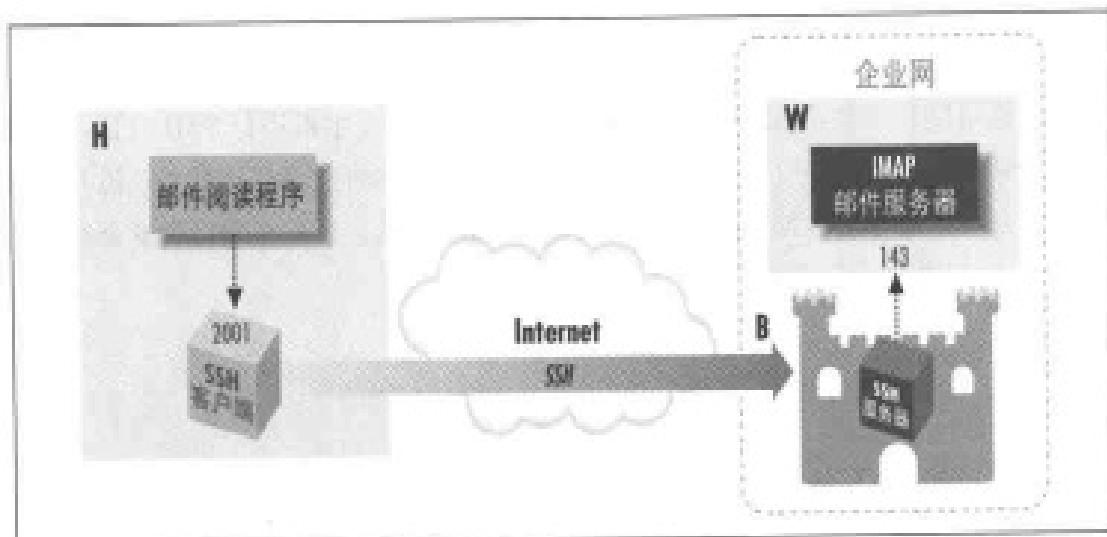


图 9-9：用端口转发通过防火墙

这样就在建立从家中主机 H 到堡垒主机 B 的 SSH 交互会话的同时，也建立起从 H 到 email 服务器 W 的一条 SSH 隧道。确切地说，作为对 2001 端口上的连接请求的响应，本地 SSH 客户端通知 B 上运行的 SSH 服务器，令其打开一个到 W 的 143 端口（即套接字 W:143）的连接。SSH 服务器之所以能够这样做，是因为 B 在防火墙之内。现在，如果像前面的转发那样让 email 阅读器连接本地端口 2001，那么通信的路径就是这样：

1. 家中主机 H 的 email 阅读器向本地 2001 端口发送数据。
2. 本地 SSH 客户端读取 2001 端口的输入，并对其进行加密，然后将其发送至隧道中。
3. 由于该隧道是防火墙可接受的 SSH 连接（端口为 22），因此能够通过防火墙。
4. B 的 SSH 服务器解密数据，将其发送至公用机 W 的 143 端口。这一过程未加加密，但是它在防火墙的保护范围之内，因此加密并不是必需的。（如果你不担心内部网络中有人窃听的话。）
5. 数据按同样步骤反向从 IMAP 服务器送回家中的 H 主机。

现在，你已经可以用 SSH 中的 IMAP 隧道绕过防火墙了。

9.2.6 无远程登录的端口转发

有时候你可能只希望用 SSH 转发端口，而不想登录远程主机进行 SSH 会话。比如说，如果你碰到我们一直说的 IMAP 转发那个例子，你可能只想读邮件，但并不想同时打开一个不必要的终端连接。在 SSH2 中这很简单，只需在端口转发命令中给 `ssh2` 加上 `-f` 选项：

```
# 仅对 SSH2
$ ssh2 -f -L2001:localhost:143 server.example.com
```

也可使用 `GoBackground` 关键字来达到同样目的：

```
# 仅对 SSH2
GoBackground yes
```

结果是，`ssh2` 自身在后台处理到转发端口 2001 的连接，仅此而已。标准输入、输出及出错信息中都不存在交互式终端会话。`-S` 选项也可以避免启动终端会话，不过与 `-f` 不同的是，它不把会话放在后台运行（换句话说，`-f` 包含 `-S`）：

```
# 仅对 SSH2
$ ssh2 -S -L2001:localhost:143 server.example.com
```

SSH1 与 OpenSSH 也支持 `-f`，但操作与 SSH2 不同。在这两者中，`-f` 的设计目标更像是执行不需要终端会话的远程命令，如那些使用 X 的图形程序。具体说来：

- 后台 `ssh` 把终端会话的本地端与 `/dev/null` 连接起来（即，`-f` 包含 `-n`）。
- 需要指定远程命令，最好不是从标准输入中读出的，这是由于后台 `ssh` 将会话通道的本地端连接到 `/dev/null` 上。

举个例子，如果打开 X 转发（这个我们稍后讨论），下面的命令就会转到后台运行，而在本地显示器上弹出一个图形化的时钟，这个时钟程序是在远程的 `zwei.uhr.org` 上运行的。

```
# SSH1, OpenSSH
$ ssh -f zwei.uhr.org xclock
```

这与下面的后台命令等效：

```
# SSH1, OpenSSH
$ ssh -n zwei.uhr.org xclock &
```

与之相反，SSH2 使用 *-f* 选项时不需要执行远程命令。也可以像上面那样加一个远程命令。此时 *ssh2* 与 SSH1 或 OpenSSH 的处理方式相同：

```
$ ssh2 -f zwei.uhr.org xclock
```

但远程命令不是必需的；你可以很方便地设置一个转发，并把 *ssh2* 放置在后台运行：

```
$ ssh2 -f -L2001:localhost:143 server.example.com
```

如果在 SSH1 或 OpenSSH 中执行这条命令，结果是：

```
# SSH1, OpenSSH  
$ ssh -f -L2001:localhost:143 server.example.com  
Cannot fork into background without a command to execute.
```

要避免新提供远程命令引起麻烦，可以用一个长期运行但不会执行任何内容的命令，如 *sleep*：

```
# SSH1, OpenSSH  
$ ssh -f -L2001:localhost:143 server.example.com sleep 1000000
```

9.2.6.1 一次性转发

当用 *-f* 或 *GoBackground* 调用 *ssh* 时，如果不使用 Unix 的 *kill* 命令明确地将其删除，它就会一直存在。（用 *ps* 命令可看到其 pid。）你可以换用一次性转发（one shot forwarding），这样，在连接结束时，客户端就会退出。具体地说，在第一次连接之前，客户端处于无限期等待状态。此后，当转发的连接数降为零时，客户端就会退出。

-fo 选项可在 SSH2 中很方便地实现一次性转发，这是在 *-f* 的基础上变化得来的。（“o”代表“one shot”）。

```
# 仅对 SSH2  
$ ssh2 -fo -L2001:localhost:143 server
```

SSH1 或 OpenSSH 不直接支持一次性转发，但用下面的方法可以达到同样的效果：

1. 用 *ssh -f* 建立转发，在远程命令中指定一个持续时间较短的 *sleep*：

```
$ ssh -f -L2001:localhost:143 server sleep 10
```

2. 在 *sleep* 过期之前使用转发连接:

```
$ ssh -p2001 localhost
```

一旦 *sleep* 命令执行完毕, 第一个 *ssh* 将尝试退出——但由于存在正在使用的转发连接, 退出请求就会被拒绝, 并打印一条(可以忽略的)警告:

```
Waiting for forwarded connections to terminate...
The following connections are open:
  port 2001, connection from localhost port 143
```

ssh 会一直等到连接结束之后才终止, 这样就实现一次性转发的功能了。

9.2.7 监听端口号

前面我们建议可以为转发的监听端选择任意一个尚未使用的端口。端口号为16位编码, 取值范围从1到65535(端口0保留)。在 Unix 这类多用户操作系统中, 1~1023 称为特权端口, 是为超级用户(用户 ID 为零)的进程所保留的。非特权进程执行绑定特权端口监听的操作都会失败, 出错信息类似于“insufficient permission(权限不足)”。

设置隧道的监听端时, 通常必须选择从 1024~65535(包含)的端口号, 这是因为你自己的用户 ID 下运行的 SSH 程序要负责监听端口, 而超级用户运行的程序与此无关。如果 SSH 报告说你选择的端口已经使用, 就另外选一个好了, 要找到一个可用的端口并不会太困难。

在隧道的目标端可以指定任意端口号, 它不论是否是特权端口都可以。你只是尝试连接该端口, 并不在其上监听。事实上, 多数情况下目标端会是一个特权端口, 因为大部分常见的 TCP 服务都使用特权端口。

如果你是某个运行 SSH 客户端的机器上的超级用户, 你就能用特权端口执行本地转发。同理, 如果你的远程账号有超级用户权限, 就可以转发远程的特权端口。

有一些 TCP 应用程序将服务器端口号写在程序里, 不允许更改。这样的应用程序在有特权端口限制的操作系统下无法使用端口转发。例如, 假设你的 FTP 客户端硬性规定连接标准的 FTP 服务器控制端口 21。为了实现端口转发, 你必须把本地端口 21 转发到远程端口 21 上。但由于 21 是特权端口, 只有超级用户才能在其上监听。幸

好 Unix 下的大多数 TCP 程序允许设置连接的目的端口号，而且 PC 和 Mac 机没有特权端口的限制。

9.2.8 选择转发的目标地址

假设想要从本地主机将连接转发至 *remote.host.net*，可以使用下面两条命令：

```
$ ssh -L2001:localhost:143 remote.host.net  
$ ssh -L2001:remote.host.net:143 remote.host.net
```

转发连接都在远程主机创建，或者连至回环地址，或者 *remote.host.net*，每种情况下连接都留在远程主机上，不会在网络上传输。然而这两个连接对于接收该转发连接的服务器而言具有明显的差异。这是因为连接的源套接字不同。到 *localhost* 的连接显示为从源地址 127.0.0.1 发出，而到 *remote.host.net* 的则是从与此名字对应的那个地址发出的。

多数时候这一区别都无关紧要，不过有时必须考虑这一点。应用服务器（例如，IMAP 守护进程）可能会根据源地址进行访问控制，其设置可能不接受回环地址。或者，服务器可能运行于具有多个网络接口的主机上，而且只绑定了其拥有的地址中的一部分，其中可能不包括回环地址。发生这种情况通常是出于疏忽，但你可能对此毫无办法。如果转发的连接端发出“connection refused”，但你能够确定服务器已经运行并能响应正常的客户端，那么就可能是这个问题。Unix 下的 *netstat -an* 能列出服务器运行的机器上的所有网络连接与监听。我们可以在相关端口上寻找监听及其绑定的地址。

如果服务器用源 IP 地址本身作为协议会话的一部分，那么问题会更加严重。在 SSH 上转发 FTP 就会引发这一问题。

一般说来，我们推荐尽可能使用“*localhost*”作为转发目标。这样就可以减少不小心把转发建立在主机之外的可能性。

9.2.9 终止

SSH 连接终止时转发会如何？端口只是不再进行转发而已；也就是说，SSH 不在其上继续监听，所有连接这些端口的请求都会返回“connection refused”。

如果一个SSH会话现在还有活动的转发连接，而你要终止这个会话，情况又会如何呢？SSH能够发觉这一点，它会一直等到所有转发都断开之后再停止会话。不同产品在这一点上会有所差异。

在SSH2中，如果你退出有活动转发连接的会话，那么该会话会依然保持打开状态，但却会自行转到后台运行：

```
remote$ logout
warning: ssh2[7021]: number of forwarded channels still open, forked to
background to wait for completion.
local$
```

这样，*ssh2*进程就一直在后台等待，直到转发连接终止后才退出。SSH1和OpenSSH与此相反，如果要断连还有活动转发的会话，会出现警告信息，但会话仍保持在前台运行。

```
remote$ logout
Waiting for forwarded connections to terminate...
The following connections are open:
    port 2002, connection from localhost port 1465
```

转义字符`return-tilde-ampersand`可使*ssh*转至后台运行，返回本地shell：[2.3.2]

```
~& [backgrounded]
local$
```

这样就和SSH2一样，在转发连接终止之后才能退出。小心这里不要用SSH转义字符`^Z`。不然*ssh*会在后台挂起，无法接收转发端口上的TCP连接。如果一不小心这样做了，就得使用shell的任务控制命令（例如，*fg*和*bg*）恢复进程。

9.2.9.1 TIME_WAIT问题

有时会发生很神秘的事情：在SSH会话停止之后，转发端口却已然存在。可能一条多次成功使用的命令突然出错了：

```
$ ssh1 -L2001:localhost:21 server.example.com
Local: bind: Address already in use
```

（这种情况通常发生在尝试用端口转发实现一些功能的时候。）我们知道2001端口上已经没有SSH的监听了，那到底是怎么回事呢？如果用*netstat*命令查看该端口上的其他监听，就会发现连接处于TIME_WAIT状态。

```
$ netstat -an | grep 2001  
tcp      0      0    127.0.0.1:2001    127.0.0.1:1472    TIME_WAIT
```

TIME_WAIT 状态是 TCP 协议的产物。在某些情况下，TCP 连接断连时，其一端的套接字在很短一段时间（通常只有几分钟）内变得不可用。所以，在断连过程结束之前就不能把该端口重用于 TCP 转发（或其他任何目的）。如果你没耐心等了，就选择别的端口，继续你的工作——或者等待一段时间，到端口能重新使用为止。

9.2.10 在服务器上设置端口转发

我们已经看到，有一些关键字和命令行选项可以在 SSH 客户端设置端口转发，如 *-L* 和 *-R*。除此之外，端口转发设置也可以在 SSH 服务器上进行。下面我们将介绍编译时配置、服务器范围配置以及每账号配置的内容。

9.2.10.1 编译时配置

编译时可用 *configure* 启用或禁用端口转发。[\[4.1.5.5\]](#)缺省值是启用转发。SSH1 的配置标志 *--disable-server-port-forwardings* 和 *--disable-client-port-forwardings* 分别禁用 *sshd1* 和 SSH1 客户端的端口转发功能。对于 SSH2 来说，只用一个标志 *--disable-tcp-port-forwarding* 就能将服务器与客户端的端口转发全部禁用。

9.2.10.2 服务器范围配置

在 *sshd* 中可以全局启用或禁用端口转发。完成这项功能的服务器范围配置关键字是 */etc/sshd_config* 中的 *AllowTcpForwarding*。该关键字的值可以是 *yes*（缺省值，表示启用转发）或 *no*（禁用转发）。

```
# SSH1, SSH2  
AllowTcpForwarding no
```

此外，SSH2 还有下列选项：

```
# SSH2  
AllowTcpForwardingForUsers  
AllowTcpForwardingForGroups
```

这些选项的语法格式与 *AllowUsers* 和 *AllowGroups* 相同。[\[5.5.2.1\]](#)它们列出允许

使用端口转发的用户或组的名字；服务器会拒绝其他任何人的端口转发请求。注意，这里指的是SSH会话的目标账号，而不是客户端上的用户名（通常这个账户名根本就是不知道的）。

F-Secure SSH1服务器还支持下列关键字：AllowForwardingPort、DenyForwardingPort、AllowForwardingTo和DenyForwardingTo，可使控制的粒度更细。两个“...Port”关键字用于控制给定TCP端口上的远程转发，可以使用通配符和数值范围进行设置。例如，下面的命令允许在3000、4000至4050（包括）及5000以上，以及任何结尾为7的端口号上进行转发：

```
# 仅对F-Secure SSH1
AllowForwardingPort 3000 4000..4050 >5000 *7
```

“...To”关键字与此类似，但控制的是到特定主机和端口的转发（即到特定套接字）。主机与端口号之间用冒号隔开，其中使用的元字符与“...Port”关键字相同：

```
# 仅对F-Secure SSH1
DenyForwardingTo server.example.com:80 other.net:* yoyodyne.com:<1024
```

可用元字符/通配符见下表：

元字符	含义	举例
*	任意数字	300*
<	小于	<200
>	大于	>200
..	值域（包含）	10..20

必须明确一点，只有关闭交互式登录，并限制可在远程运行的程序之后，这里的指令才能真正禁用端口转发。否则有经验的用户很容易就能在SSH会话上运行他们自己的端口转发了。在非技术人员的圈子里，这种设置是很有效的一种防护措施，不过对知道自己在干什么的人而言就没什么用了。

9.2.10.3 每账号配置

可以在账号中禁止客户端使用特定的密钥进行端口转发。方法是，将公钥置于*authorized_keys*文件中，并在前面加上no-port-forwarding选项：

```
# SSH1, OpenSSH  
no-port-forwarding ...key...
```

(SSH2 中目前无此项功能。) 这样任何使用该密钥认证的 SSH 客户端就都无法在你的 SSH 服务器上执行端口转发了。

我们前面介绍的服务器范围的转发配置同样也适用于此。除非你对该密钥能执行的操作进一步进行限制，否则这种限制根本就没什么实际意义。

9.3 X 转发

你已经了解了普通的 TCP 端口转发，现在我们开始介绍一个新的话题：转发 X 协议连接。X 是 Unix 工作站上很流行的窗口系统，其中一项重要功能是它的透明性。使用 X 转发，我们可以运行远程 X 应用程序，但将其显示在本地机器上（反之亦然，也可以把本地运行的程序显示在远程机器上）。但是机器间的通信不安全，它完全暴露在窥探器之下。不过还好，SSH 的 X 转发可通过 X 协议隧道保护通信。

X 转发也解决了防火墙带来的很多问题。假设你是系统管理员，而有些产品发布主机位于防火墙之外，容易受到攻击。你用 SSH 登录其中一台机器，并打算运行一个 X Window 系统下的图形化性能监视工具，如 Solaris 的 *perfmon*。但这是无法实现的，因为外部主机必须为此建立返回你登录的那台内部主机的 TCP 连接，而防火墙会将其封锁（这样做是必需的，因为 X 非常不安全）。X 转发能够通过 SSH 在隧道中安全地建立 X 协议连接，使其绕过防火墙，问题就解决了。

我们首先对 X 做一简短叙述，然后简要介绍 X 转发的细节。在描述如何使用 X 转发这一问题的同时，我们还揭示了 X 认证的本质、它与 SSH 的交互方式，以及其他技术问题。

9.3.1 X Window 系统

X Window 系统，简称 X，是 Unix 机器上应用最广泛的图形显示系统。X 与 SSH 相似，也有客户端和服务器。X 客户端是窗口化的应用程序，如，终端模拟器、绘图程序、图形化时钟程序等等。X 服务器是后台运行的显示引擎，负责处理 X 客户端的请求，它用某种网络协议通信，这种协议称为 X 协议。典型的情况是一台机器上运行单个 X 服务器，但可能有许多 X 客户端。

VNC 转发：X 转发的替代品

X 转发从安全的角度看是有问题的，这与 X 本身不安全的原因相同。我们将看到，X 的设计意味着远程程序必须创建独立的返回到用户的网络连接；这就需要另一个认证与授权过程，情况变得复杂化，也使得攻击者有机可乘。SSH 的 X 转发会尽可能地保护这一过程，不过在某些环境中，结果还是不能令人满意。

基于 SSH 的虚拟网络计算技术（Virtual Network Computing, VNC）可以代替 X 转发。VNC 是英国的 AT&T 实验室开发的免费软件，它提供 Unix 和 Windows 平台下的远程 GUI 访问机制。VNC 可以在运行 X 的本地 Unix 主机上打开一个窗口，而上面显示的是远程 Windows 主机的桌面，因此能对 Windows 窗口执行远程操作。VNC 仅涉及单独一条向外的连接，所以 VNC 的 SSH 隧道比 X 的更容易建立、也更安全。下面的网站上有更多的 VNC 信息（及软件下载）：

<http://www.uk.research.att.com/vnc/>

更重要的是，X 支持基于网络的复杂窗口管理。X 客户端不仅可以在其本地主机上打开窗口，还可以在网络中的其他计算机上打开窗口，不论这些计算机是在大厅的另一头，还是在地球的另一头。要实现这个功能，X 客户端需要建立到某个远程 X 服务器的网络连接，将会话内容加载于其中，并通过 X 协议绘制远程屏幕、接收远程键盘事件、获取远程鼠标位置等等。这显然需要某种类型的安全保证，我们很快就会讨论到。

X 的一个最重要的概念是显示区（display），这是 X 服务器管理下抽象意义上的显示屏。X 客户端启动时需要知道使用哪个显示区。显示区的名字由 *Host:n.v* 形式的字符串组成，其中：

- *Host* 是运行控制显示区的 X 服务器的主机名。
- *n* 是显示器编号，整型，通常为 0。X 允许用单个服务器控制多个显示器；其他的显示器编号为 1、2，依此类推。
- *v* 是可视区编号，也是整型。可视区是一个虚拟的显示器。X 支持在一个单独的物理显示器上建立多个虚拟显示区。如果只有一个虚拟显示区（这是最普遍的情况），就可以忽略 “.v”，其缺省值为 0。

例如，*server.example.com* 的显示区 0 可视区 1 可表示为字符串 “*server.example.com:0.1*”。

Unix 下多数 X 客户端程序允许用两种方式定义显示区字符串：命令行参数 *-d* 或 *-display*，或者环境变量 DISPLAY。例如，在工作站 *anacreon* 的惟一一个 X 显示区中运行 X 客户端程序 *xterm*，命令行方式为：

```
$ xterm -d anacreon:0 &
```

环境变量方式为：

```
$ setenv DISPLAY anacreon:0  
$ xterm &
```

X 是一种庞大深奥的软件产品，O'Reilly 在这方面出版了很多相关图书进行介绍。我们的解释仅仅涉及 X 的表面，但对于理解 X 转发而言已经够了。

9.3.2 X 转发

尽管 X 客户端可以与远程 X 服务器通信，但这种通信是不安全的。由于连接没有加密，网络窥探器很容易就能监听到 X 客户端与服务器之间的所有交互，如键盘事件及显示的文字。不仅如此，多数 X 环境用于连接远程显示区的认证方式都很原始。有经验的攻击者也能连接到你的显示区、监视你敲击键盘并控制其他正在运行的程序。

这次 SSH 又来帮忙了。可将 X 协议连接导入 SSH 连接，以保障其安全性，并提供更强的认证。此项功能称为 X 转发。

X 转发工作方式如下。当 SSH 客户端连接到 SSH 服务器时，它请求建立 X 转发（假设该客户端允许 X 转发）。如果服务器也允许此连接建立 X 转发，登录过程就正常进行，不过服务器会在后台执行一些特殊的操作。除了处理终端会话，它还将其自身设置成一个运行于该远程主机上的代理 X 服务器，并设置对应远程 shell 的 DISPLAY 环境变量设置，令其指向该代理 X 显示区：

```
syrinx$ ssh sys1  
Last login: Sat Nov 13 01:10:37 1999 from blackberry  
Sun Microsystems Inc. SunOS 5.6 Generic August 1997  
You have new mail.
```

```
sys1$ echo $DISPLAY
sys1:10.0
sys1$ xeyes
"xeyes" 的 X 客户端显示在本地屏幕上
```

看上去 DISPLAY 的值指向 sys1 的 X 显示区 #10，但实际上并不存在此显示区。（事实上，sys1 可能根本就没有真正的显示区。）DISPLAY 实际上指向 SSH 建立的 X 代理，即，SSH 在此伪装成 X 服务器。如果现在运行 X 客户端程序，就会连接到此代理。代理就像一个“真正的”X 服务器，接下来它也会指示 SSH 客户端，令其担当一个代理 X 客户端的角色，向你本机的 X 服务器发起连接。然后，SSH 客户端和服务器将协同工作，在两个 X 会话之间的 SSH 隧道上来回传递 X 协议信息，X 客户端程序就如同直接连接你的显示区那样显示在你的屏幕上，这就是 X 转发的基本思想。

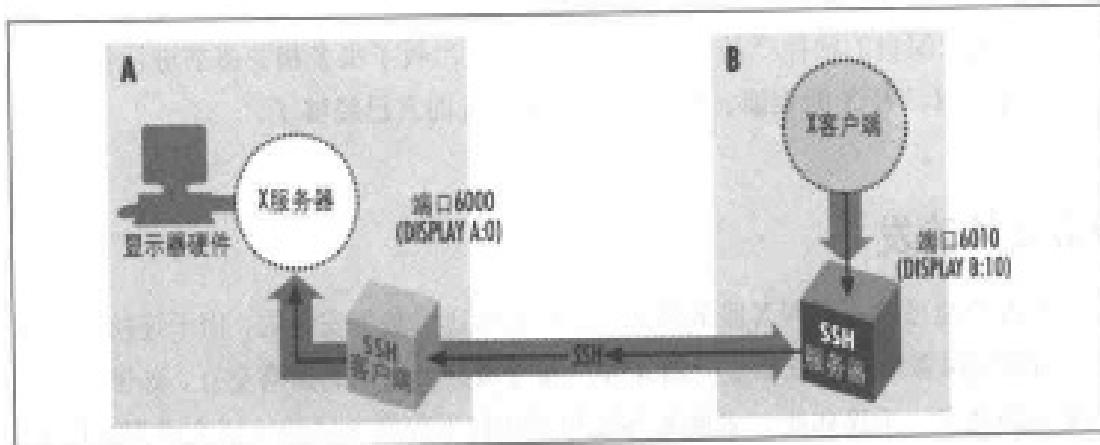


图 9-10：X 转发

X 转发甚至可以解决前面提到的防火墙问题，只要防火墙允许 SSH 通过就行了。如果在本机和远程主机之间有防火墙，并在远程主机上运行 X 客户端，X 转发就能通过隧道使 X 连接通过防火墙的 SSH 端口连至本地主机。因此，X 客户端的窗口可在本地显示区中打开。如果没有 X 转发，防火墙就会将连接阻断。

X 转发在某些方面听起来可能很像前面介绍过的端口转发。事实上，X 转发正是端口转发的一个特例，SSH 对此有特别的支持。

9.3.3 启用 X 转发

在 SSH1 和 SSH2 中，缺省情况下 X 转发都是启用的，但在 OpenSSH 中缺省是禁用的。下面是在客户端中启用/禁用 X 转发的方法。通常的端口转发与 TCP 端口号有

关，而 X 转发只有一个启用/禁用的状态。我们可以在 SSH 客户端设置文件中把关键字 ForwardX11 设置成“yes”（缺省值，表示启用）或“no”（表示禁用）。

```
# SSH1, SSH2, OpenSSH
ForwardX11 yes
```

在命令行方式中我们可以使用 -x 来禁用 X 转发：

```
# SSH1, SSH2, OpenSSH
$ ssh -x server.example.com
```

SSH2 和 OpenSSH 可以使用下面的参数启用 X 转发：

```
# 仅对 SSH2
$ ssh2 +x server.example.com

# 仅对 OpenSSH
$ ssh -X server.example.com
```

9.3.4 配置 X 转发

X 转发的运行方式可以使用编译时配置、服务器范围配置及每账号配置进行修改。

9.3.4.1 编译时配置

编译 SSH1 和 SSH2 时可以启用/禁用 X 的支持。对应的编译标志是 --with-x 与 --without-x。

```
# SSH1, SSH2
$ configure ... --without-x ...
```

此外，如果编译时我们选择支持 X，还可以设置 X 转发的缺省的行为。对于 SSH1，我们可以使用编译标志 --enable-client-x11-forwarding (或 --disable-client-x11-forwarding) 和 --enable-server-x11-forwarding (或 --disable-server-x11-forwarding) 在客户端和服务器分别设置缺省启用/禁用 X 转发。

```
# 仅对 SSH1
$ configure ... --disable-server-x11-forwarding ...
```

对于 SSH2 来说，我们可以使用 --enable-X11-forwarding 或 --disable-X11-forwarding 缺省启用或禁用所有的 X 转发。

```
# 仅对 SSH2
$ configure ... --enable-X11-forwarding ...
```

请记住，enable/disable 标志仅仅设置了缺省的行为，你还可以在服务器范围配置和每账号配置中覆盖这些缺省值。

9.3.4.2 服务器范围配置

服务器范围配置关键字 X11Forwarding (SSH1、SSH2、OpenSSH) 及其同义词 ForwardX11 (SSH2) 和 AllowX11Forwarding (SSH2) 将在 SSH 服务器端启用 / 禁用 X 转发。缺省状态为启用。

```
# SSH1, SSH2, OpenSSH
X11Forwarding no

# 仅对 SSH2: 二者都可生效
ForwardX11 no
AllowX11Forwarding no
```

关键字 X11DisplayOffset 可以保留一些 X11 显示区号，这样 sshd 就不能使用这些号码。这个关键字指定了 SSH 可以使用的最小显示区号，以免 sshd 与运行在较小显示区号上的真正的 X 服务器冲突。例如，如果你通常在显示区 0 和 1 上运行真正的 X 服务器，就设置为：

```
# SSH1, OpenSSH
X11DisplayOffset 2
```

关键字 XauthLocation 指定 xauth 程序的路径，此程序用于操纵 X 的授权记录。这个关键字等我们讨论完 xauth [9.3.6.4] 之后再说明。

```
# 仅对 SSH1
XAuthLocation /usr/local/bin/xauth
```

9.3.4.3 每账号配置

在 SSH1 或 OpenSSH 的 *authorized_keys* 文件中，我们可以禁止使用特定的密钥进行认证的 SSH 连接执行 X 转发。这是通过 no-X11-forwarding 选项实现的。[8.2.8]

```
# SSH1, OpenSSH
no-X11-forwarding ... 密钥部分 ...
```

9.3.5 X 认证

前面我们已经介绍过，当 X 客户端请求连接 X 服务器时，X 就会执行其自己的认证。现在我们将深入探索 X 认证的技术细节，指出其不安全的原因，并说明如何在其上建立安全的 SSH X 转发。

多数情况下 X 转发都非常简单，人们不必考虑 X 转发是如何工作的。以下内容可以有助于你理解这个问题，也使你（还有我们自己）在与别人谈论技术问题时能讲得深入一些。

9.3.5.1 X 认证如何工作

当 X 客户端向 X 服务器请求建立连接时，服务器就会对该客户端进行认证。也就是说，X 服务器确定客户端的身份标识，从而决定是否允许其建立到服务器显示区的连接。当前版本的 X Window 系统（X11R6）提供两种认证机制：基于主机的认证和基于密钥的认证。

基于主机的 X 认证

这是比较简单的方式。我们可以使用程序 *xhost* 给出允许连接到 X 显示区的主机列表。注意：这时连接仅仅使用主机名进行认证，而不会使用用户名。也就是说，列表上的主机中的任何用户都可以连接到显示区上。

基于密钥的 X 认证

xauth 程序为 X 客户端维护了一个 X 认证密钥表，或称为显示区密钥表。密钥保存在一个文件中，通常是 *~/.Xauthority*，这一文件同时也存储了与客户端试图访问的多个显示区的有关的信息。当 X 客户端连接到某个要求认证的服务器时，客户端就从 *xauth* 维护的信息中将那个特定显示区对应的认证信息提取出来，提交给服务器。如果认证成功，X 客户端随后就连接到该 X 服务器管理的显示区上。

显示区密钥从 X 服务器上获得，根据环境的不同，获取方式也多种多样。例如，如果直接在终端上用 *xinit* 或 *startx* 启动服务器，这些程序就会调用 X 服务器，并将服务器的密钥直接插入 *xauth* 数据中。如果向一台运行 X 显示管理器（X Display Manager，XDM）的远程主机发起连接，那么在建立 XDM 会话的同时，密钥将发送至你的远程账号上。

9.3.5.2 xauth 与 SSH 的 “rc” 文件

SSH有一些文件，我们可以对这些文件进行设置，并让其在客户端登录进来时在服务器端运行。这些文件包括用于服务器范围配置的`/etc/sshrc`和用于每账号配置的`~/.ssh/rc`。它们既可以是 shell 脚本，也可以是任何种类的可执行文件。

有件事情我们要注意：如果不运行某个 `rc` 程序，那么 `sshd` 在执行 `xauth` 时仅仅把代理显示区密钥加载进来。如果运行 `rc` 程序，就会用标准输入设备中的单独一行，将密钥类型及数据反馈至该程序，此时存储显示区密钥的就是这个 `rc` 程序了。如果某些环境中只运行 `xauth` 不能满足认证的要求，那么此功能提供了一种定制显示区密钥处理方式的方法。

9.3.5.3 X 认证的问题

如果已经用过 X，你就会认为认证过程可能是透明的，看上去也运行得挺不错。然而透过表面现象，其机制本身是不安全的。主要的问题是：

xhost 不安全

一旦授权某台远程主机连接你的显示区，那台主机上的任意用户就都能建立连接了。不仅如此，与 `r-` 命令一样，这种认证方式依赖于建立连接的主机的网络地址，而攻击者要伪装一个地址是很容易的。

密钥传输过程可能是手工完成的，因此不安全

有些远程登录协议（如 `telnet`）与 X 认证不兼容。如果显示区密钥不能在远程主机上获得，那么就必须自己将其传送过去，这可以手工进行，也可以在登录脚本中自动传输。这样不仅操作起来麻烦，而且很不安全，因为密钥是以明文方式在网络上传输的。

最常用的方法 MIT-MAGIC-COOKIE-1 也不安全

尽管采用随机位串（或 cookie）作为 `xauth` 显示区密钥，然而此密钥在每个连接建立的开始仍然是用明文传输的，这时就可能被截获并读取其内容。

远程主机可能不支持您选择的 X 认证方式

X11R6 支持其他一些更加安全的认证方式。例如，XDM-AUTHORIZATION-1 使用了 DES，SUN-DES-1 运用了 Sun 公司的安全 RPC 系统，MIT-KERBEROS-5

引入 Kerberos 用户到用户 (user-to-user) 的认证机制 (注 6)。不过在特定的 X 软件中，这些方法经常用不了。有时由于密码出口受到限制，这些功能没有编译到 X 安装版中；有时 X 版本可能太老，不支持这些更安全的方法。

如果远程主机不安全，那么显示区密钥也会受牵连

最好的情况是，X 服务器支持强认证，密钥也能安全地拷贝至远程主机，但你仍然得把关键的密钥存储在那里。如果那台机器不能信任，那么你的密钥就危险了。（SSH 不存在这个问题，因为存储在 SSH 服务器上的只有公钥。）

9.3.5.4 SSH 与认证欺骗

SSH 通过 X 转发为 X 会话提供透明、安全的认证与密钥传输机制，这是通过一种叫认证欺骗 (authentication spoofing) 的技术实现的，如图 9-11 所示。认证欺骗涉及一个伪装的显示区密钥，该密钥称为代理密钥 (proxy key)，用于在远程主机端认证访问 SSH X 代理服务器的访问权限。当传递包含密钥的 X 流量时，SSH 聪明地替换了真正的显示区密钥。下面解释它是如何工作的。

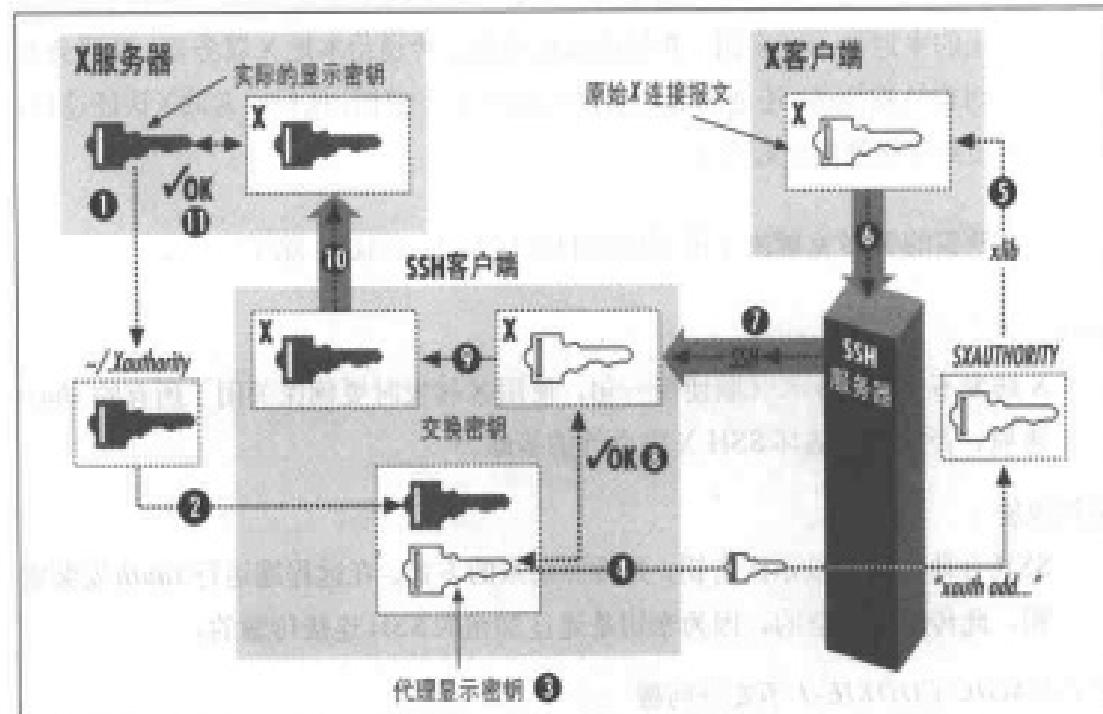


图 9-11：对转发的 X 连接进行认证

注 6：有关这些方法的内容请参看 X11R6 Xsecurity (1) 的手册页。还要记住，这只是认证，而不是加密。X 连接的内容仍是未加密的，在网上很容易被窃听和篡改。

游戏从这里开始。假设你已经登录具备本地显示区的一台本地主机。该主机上运行了 X 服务器和 SSH 客户端。在网络连接另一端的远程主机上运行着一个 SSH 服务器，X 客户端将在那里调用。目标是使用 SSH，让远程的 X 客户端出现在本地显示区中。

首先，运行一个本地的 SSH 客户端，令其建立 X 转发。该 SSH 客户端向远程的 SSH 服务器请求 X 转发，同时从 *.Xauthority* 文件中读取本地显示区密钥。

接下来，SSH 客户端生成一个代理密钥。这个密钥是一个与本地显示区密钥长度相同的随机信息串。SSH 客户端随之将此代理密钥及其类型（例如，MIT-MAGIC-COOKIE-1）发送到远程主机上。SSH 服务器根据你的请求执行 *xauth* 程序，将代理密钥与本地显示区关联起来。这一步完成了设置 X 转发的工作。

当启动远程 X 客户端时，本地的 SSH 客户端连接至本地的 X 显示区。随后，SSH 客户端开始监听从转发的连接上来的第一个 X 协议消息，并对其进行特殊处理。分析其语法结构，找出其中的 X 认证密钥，与代理密钥进行比较。如果密钥不相符，SSH 客户端就会将此消息丢弃，并关闭连接。否则，如果两个密钥相符，SSH 就会将其替换成真正的本地显示区密钥，并将修改过的消息传递给本地 X 服务器。X 服务器不知道密钥曾经被换过，还是满心欢喜地读取显示区密钥，执行正常的 X 认证过程。此时，X 转发连接就建立起来了。

基于认证欺骗的 X 转发解决了前面提到的所有 X 认证问题，除了：

xhost

X 转发不使用 *xhost*。（顺便提一句，使用 X 转发时要确保关闭了所有的 *xhost* 选项，否则就会破坏 SSH X 安全性的基础。）

密钥传输

SSH 自动传输 X 显示区密钥，并按照请求的方式，在远程端运行 *xauth* 安装密钥。此传输是安全的，因为密钥是通过加密的 SSH 连接传输的。

MIT-MAGIC-COOKIE-1 不安全问题

现在，每一个 X 会话开始时传送的密钥及会话本身都在 SSH 会话中加密。这从操作手段上极大地增强了普通的 X 认证机制的安全性。

不可信的远程主机

在认证欺骗机制中，传送到远程主机的只有代理密钥，而不再是真正的显示区

密钥。代理密钥只在通过 SSH 连接显示区时才会用到，不能用其直接连接显示区。SSH 会话一结束，代理密钥马上就没用了。SSH 会话是不断变化的，而有些人常常几天都一直打开 X 会话（用同一个密钥），所以说 X 转发是个极大的改进。

9.3.5.5 改进的认证欺骗

X 转发剩下的问题就是 X 认证机制不被支持的可能性。本地端可能会使用更加复杂的认证方式，而远程主机也许并不支持。

理论上讲，SSH X 转发能解决此类问题，只要不论实际使用的本地认证方式如何，都一直安装 MIT-MAGIC-COOKIE-1 类型的代理密钥即可。SSH 客户端检查了 X 客户端的密钥是否与代理密钥相符之后，就会生成本地认证所需的、与实际认证类型相符的密钥，并将原来的密钥替换掉。

然而，SSH 还没有实现到这一步。服务器把密钥作为位串如实比较，SSH 客户端也不会考虑密钥类型，而是逐字替换密钥。结果是如果使用更强的 X 认证方式，如 XDM-AUTHORIZATION-1，*sshd* 只会盲目地将加密的密钥与代理密钥进行比较，得出两者不相符的结论，从而抛弃该连接。这一错误不仅很奇怪，而且没有提示；如果软件能发现用了不支持的模式，在建立连接的时候就发出警告，那该多好呀。

如果 SSH 知道所有 X 认证方式的细节，那么就能在一端检验代理密钥，再在另一端为 X 服务器生成正确的密钥。然而即便有人可能把 X11 库加进 SSH 里，从而获得必要的算法，这样做的开发工作量仍然很大。SSH 还不得不区分密钥信息长度、并构造一条新的 X 消息携带代理密钥，而不能将其拷贝至已经存在的密钥中。

如果不采用认证欺骗可以实现 X 转发，那也非常有用。这样就可以用某种方式自行处置连接安全性，比如，用 *xhost* 允许本地主机的所有连接（因此也包括 SSH X 代理），同时依然使用基于密钥的机制认证从其他地方发起 X 连接。这可以用通常的端口转发实现，下面我们会讨论到，不过，在 SSH 中直接支持更方便些。

9.3.5.6 非标准 X 客户端

通常，X 客户端通过连接普通的 X 编程库 Xlib 实现 *xauth* 之类的 X 认证。但偶尔也会碰到没用 Xlib 的 X 客户端，这样的客户端会简单地忽略认证过程。由于不能关闭

SSH的X认证欺骗机制，所以无法通过SSH的X转发使用这样的程序；返回信息如下：

```
x11 connection requests different authentication protocol: 'MIT-MAGIC-COOKIE-1'  
vs. ''
```

这时可以转而使用一般的端口转发。举例如下：

```
foo% ssh -R6010:localhost:6000 bar  
bar% setenv DISPLAY bar:10
```

注意，这样就绕过了X转发的强制要求，不用对转发的X连接进行*xauth*认证了。如果真正的X服务器用*xhost*进行访问控制，那么该端口转发就允许主机foo上的任意用户连接X服务器。如果需要这样用，那么可得小心。

9.3.6 深入讨论

如前所述，X转发通常不需要什么特殊的操作就能正常工作。然而在某些特殊情况下，可能需要一些额外的步骤。

9.3.6.1 X服务器配置

X转发生效的前提是，X服务器必须能够接受来自SSH客户端的代理X连接。有时一开始不是这么设置的，因为在某些情况下这通常都不必要。例如，如果用PC机上的X服务器通过XDM访问远程Unix主机，可能永远也不能运行本地的X客户端，因此缺省状态下这可能也是不允许的。可用*xhost +localhost*允许接收本地PC的所有连接，对其他地方来的连接依然使用基于密钥的认证方法。这样，SSH转发（及认证）就是可接受的了。

9.3.6.2 设置DISPLAY环境变量

SSH只在X转发生效时才自动设置DISPLAY变量。如果不使用X转发，又想通过SSH登录某台远程主机，并在上面使用X，那么请记住要自己设置DISPLAY变量。真的，你应该只在两台机器都位于同一可信网络中时才这样做，因为X本身非常不安全。

千万不要随意修改DISPLAY！通常人们会在登录命令文件或其他什么地方设置

DISPLAY 变量。一不小心，就可能在你没有意识到的情况下使 X 连接变得不安全了。如果你在阻隔一般 X 连接的防火墙中用 SSH 建立通信隧道，那么 X 客户端无法工作时你当然就会注意到这一点。但是如果能建立一般的 X 连接，而实际上不期望这样，同时 X 转发又没有启动，那么未经保护的 X 程序最终会悄无声息的建立起来。这样就有很好的理由阻断引发安全危机的 X 连接，或者将 X 服务器设置为只接受来自本地主机（经过 SSH 转发的 X 连接的源地址）的连接。如果这样还不可行，还可在登录脚本中增加如下内容：

```

if ($?DISPLAY) then
    set display_host = `expr "$DISPLAY" : '\(.*\);'`
    set display_number = `expr "$DISPLAY" : ',.*:\([^.]*\)'`
    set my_host = `hostname`
    set result = `expr '(' "$display_host" = "$my_host" ')' & '(' \
                  "$display_number" '>' "0" ')'
    if ($result == 0) then
        echo "WARNING: X display $DISPLAY does not appear to be protected by SSH!"
        echo "unseting DISPLAY variable just to be safe"
        unsetenv DISPLAY
    endif
endif

```

9.3.6.3 共享账号

多人共享同一账号会引发一些 X 转发的问题。例如，一组系统管理员共享 root 账号是很普遍的情况。在使用 root 账号时，每个人都会维持自己的环境，他们可能会设置与 root 账号不同的 USER、LOGNAME 及 HOME 环境变量，以示区分。如果用 SSH 登录 root，同时开启 X 转发，SSH 就会在读登录脚本及设置环境变量之前将代理 xauth 密钥加入 root 的 .Xauthority 文件中。结果是，只要一登录进去开始使用 X，就会出错：因为 X 客户端要（根据 HOME 变量设置）在 .Xauthority 文件中查找密钥，但实际上密钥并不在那儿。

解决此问题的方式或者是将 XAUTHORITY 变量指向 root 的 .Xauthority，或者是在登录脚本中写如下的代码，把所需密钥拷贝到个人的文件中。

```

if ((!$uid == 0) && (!$?SSH_CLIENT) && ($?DISPLAY)) then
    # 如果我用 X 转发设置 ssh-1 root，则 X 代理服务器的 xauth 密钥会被加入 root 的 xauth
    # db 中，而不是我的。看看 root 的 xauth db 中是否有我显示区的人口……
    set key = `bash -c "xauth -i -f /.Xauthority list $DISPLAY 2> /dev/null"`
    # ..... 如要有，将其拷入我的 xauth db 中。
    if ($? == 0) then
        xauth -bi add $key
        chown res ~res/.Xauthority >& /dev/null

```

```
endif
endif
```

9.3.6.4 xauth 程序的位置

回忆一下，*sshd* 根据你的操作运行 *xauth* 程序，将代理密钥加入远程主机上你的 *.Xauthority* 文件中。在配置 SSH 包并编译 *sshd* 可执行程序时，*xauth* 程序的位置就确定了。如果 *xauth* 在此之后被移动，X 转发就无法工作 (*ssh -v* 会揭示其中详情)。对于 SSH1 和 OpenSSH，服务器端的系统管理员可用服务器范围配置关键字 *XAuthLocation*，这样不用重新编译 *sshd* 就能设置 *xauth* 程序路径。

```
# 仅对 SSH1
XAuthLocation /usr/local/bin/xauth
```

9.3.6.5 X 转发与 GatewayPorts 特性

前面讨论的 *GatewayPorts* (*-g*) 特性只对通常的端口转发有效，不适用于 X 转发。SSH1、SSH2 及 OpenSSH 上的 X 代理总会监听所有的网络接口，并接受来自任何地方的连接请求，当然随后这些连接会像前面说的那样去执行 X 认证。要限制 X 客户端源地址，我们可以使用下一节中讨论的 TCP-wrappers。

9.4 转发安全性：TCP-wrappers 及 libwrap

本章很多地方都曾经提到安全问题和转发的局限性问题。到现在为止，我们还没看到多少手段可以控制“谁可以连接一个转发的端口”。SSH1 及 OpenSSH 的缺省设置是仅能从本地主机发起连接，这对于单用户机器而言当然是安全的。但是如果需要接受别处发出的连接，就有问题了，因为这是一个全部或没有的问题：要想允许从别处发起连接（用 *-g* 或 *GatewayPorts=yes*），就必须允许从所有的地方发起连接。SSH2 更糟：转发端口点可以接受来自任何地方的连接。X 转发稍好一些，因为 X 协议有自身的认证机制；但可能还会需要限制访问，以免安全缺陷被入侵者利用，或受到拒绝服务攻击。基于 Unix 平台的 SSH 的确提供了一项可选的基于客户端地址的访问控制功能，称为“TCP-wrappers”。

“TCP-wrappers”是一个软件的名字，作者为 Wietse Venema。如果你的 Unix 尚未安装，可从下面的地址得到：

<http://ftp.porcupine.org/pub/security/index.html>

TCP-wrappers 是一种全局访问控制机制，它与其他 TCP 服务器（如，*sshd* 或 *telnetd*）集成在一起。其访问控制方式基于到达的 TCP 连接的源地址。即，一个 TCP-wrappers 按照 */etc/hosts.allow* 和 */etc/hosts.deny* 所指定的内容并根据连接的来源判断是允许还是拒绝。图 9-12 显示了 TCP-wrappers 与 SSH 配置体系相结合的情况。

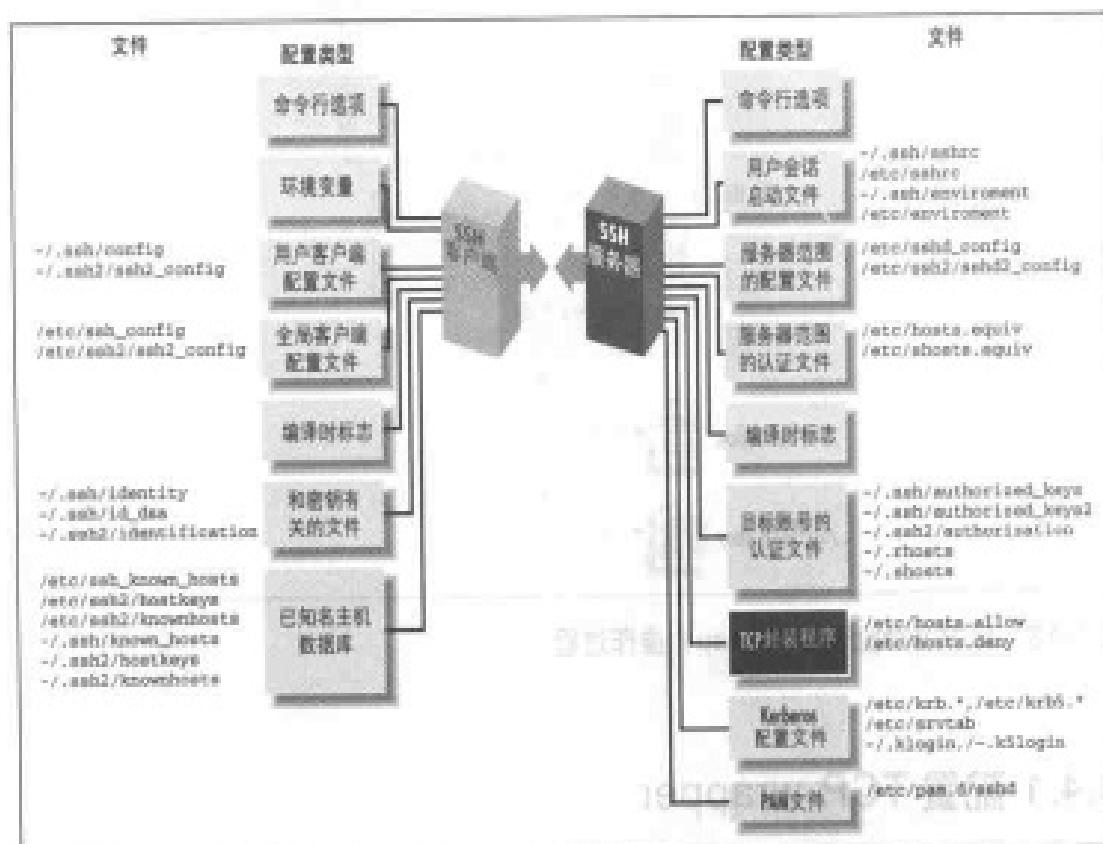


图 9-12: TCP-wrappers 与 SSH 配置（高亮显示部分）

有两种使用TCP-wrappers的方式。最常用的方式称为wrapping，应用于通常由inetd调用的TCP服务器中。对服务器施行“wrap”的办法是编辑/etc/inetd.conf，修改服务器的配置行。服务器不是直接调用的，而是先调用TCP-wrapper守护进程tcpd，由它再去调用先前的服务器。因此，可通过修改TCP-wrapper配置文件来指定需要的访问控制方式。tcpd则根据该配置文件的内容做出认证决定。

inetd 不用修改 TCP 服务器程序就可以进行访问控制。这样很好。不过 *sshd* 通常不是由 *inetd* 调用的,[5.4.3.2]因此必须使用第二种方法，修改源代码。SSH 服务器必须用 *--with-libwrap* 选项编译，从其内部支持 TCP-wrappers，才能实行 TCP-wrapper 控制.[4.1.5.3]随后，*sshd* 调用 TCP-wrapper 库函数，根据 */etc/hosts.allow* 及 */etc/hosts.deny* 中的规则做出确切的访问控制判断。因此从某种程度上讲，“wrapper”这个词有一定误导性，因为要支持 TCP-wrappers，我们并不是对 *sshd* 进行封装，而是修改它。

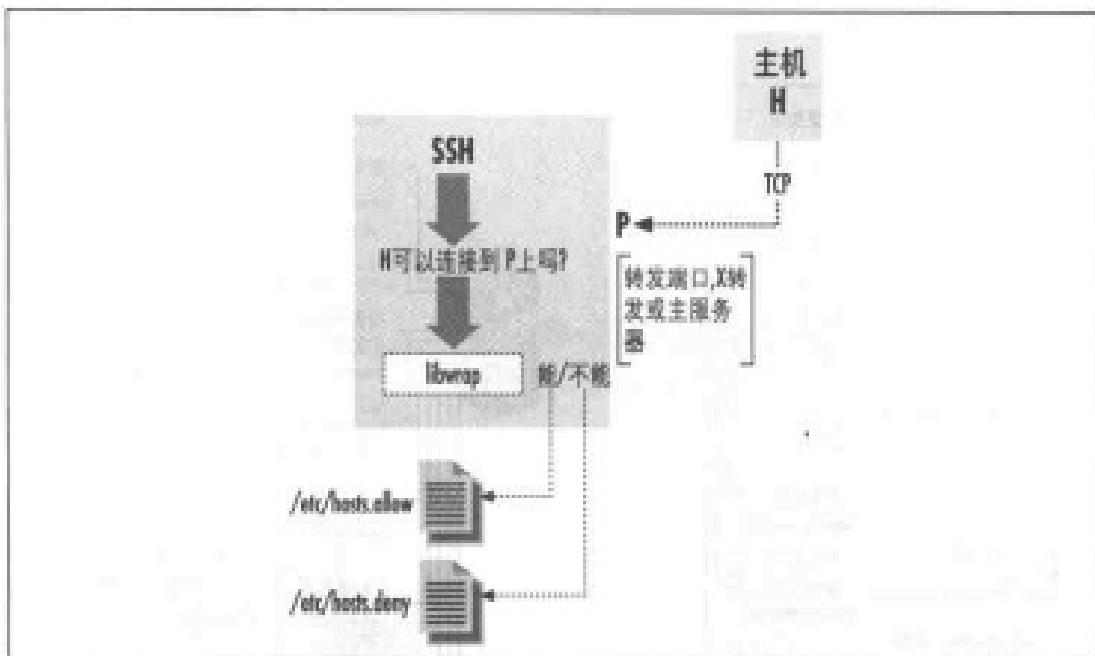


图 9-13: TCP-wrapper (libwrap) 操作过程

9.4.1 配置 TCP-wrapper

TCP-wrappers 的访问控制语言中有许多选项，根据发布者及版本的不同这些选项也有区别。在此我们不会涉及语言的全部内容。要全面理解，请参看本地文档：手册页中关于 *tcpd* (8), *hosts_access* (5) 及 *hosts_options* (5) 的内容。我们只介绍一些简单常用的配置选项。

TCP-wrapper 配置选项存于 */etc/hosts.allow* 和 */etc/hosts.deny* 文件中。这些文件包含如下的显示模式：

```
service_1 (service_2 service_3 ...) : client_1 (client_2 client_3 ...)
```

每一个模式都与某些（服务器，客户端）组相对应，因此也可能对应于某个特定的 C/S 模式的 TCP 连接。具体说来，客户端 C 与服务器 S 之间的连接符合这项规则的条件是：某项服务 *service_i* 与 S 相符，同时某个 *client_j* 与 C 相符（马上就解释这些子模式的格式及匹配规则）。*hosts.allow* 文件先被检索，然后是 *hosts.deny*。如果在 *hosts.allow* 中有匹配的模式，就允许连接。如果在这里没有，但在 *hosts.deny* 中匹配上了，连接就被终止。最终，如果每一个文件没有与之匹配的模式，也允许连接。如果这两个文件都不存在，就假设这些文件实际存在，但其中不包括任何匹配模式。因此请留意，缺省状态是允许所有连接。

hosts_options(5) 手册页中还有一种扩展语法。它也许能用，也许不能，这全在于 TCP-wrapper 库的创建方式。这其中包含的选项更多，不过有一点尤其值得一提，可以单独设定拒绝或抛弃某个匹配连接的规则；如：

```
"sshd1 : bad.host.com : DENY"
```

可利用此语法将所有规则都放进 *hosts.allow* 文件中，不用非得使用两个文件了。要拒绝所有未经明确允许的连接，只需在该文件的末尾加上 ALL:ALL:DENY。

在模式中，每一项 *service* 都是一个该模式应用的服务器名。SSH 可识别下列服务名：

sshd

主 SSH 服务器。可以是 *sshd*、*sshd1*、*sshd2* 或其他任何可激活守护进程的名字（用其 *argv[0]* 的值）。

sshdfwd-x11

X 转发端口。

sshdfwd-N

转发的 TCP 端口 N（例如，转发端口 2001 记为 *sshdfwd-2001* 服务）。

注意：X 及端口转发控制功能只在 SSH1 和 SSH2 中可用；OpenSSH 仅能使用 *libwrap* 控制对主服务器的访问。

每一个 *client* 都是匹配接入客户端的模式。它可以是：

- 点分十进制计数表示的 IP 地址（例如，192.168.10.1）。

- 主机名（DNS，或主机使用的任何名字服务）。
- 网络地址/掩码格式的IP网段（例如，192.168.10.0/255.255.255.0。请注意是“掩码”语法格式。192.168.10.0/24是无法识别的）。
- “ALL”，与任意客户端源地址相匹配。

例9-1给出了配置文件`/etc/hosts.allow`的一个例子。该配置允许从本地主机的回环地址及全部192.168.10.x地址连接到任意服务。该主机上运行了公用的SSH1、POP及IMAP服务器，因此我们允许从任何地方访问这些服务，但对于SSH-2客户端，我们只允许其来自另一个特定网段。

例9-1：`/etc/hosts.allow`文件示例

```

#
# /etc/hosts.allow
#
# 由tcpd(见inetd.conf)调用或使用libwrap程序的网络访问控制。见手册
# hosts_access(5)和hosts_options(5)

# 允许来自我的网络或当地主机(回环地址)的所有连接
#
ALL: 192.168.10.0/255.255.255.0 localhost

# 允许来自任何地方的连接连向这些服务
#
ipop3d imapd sshd1 : ALL

# 允许来自C类网络的SSH-2连接
# 192.168.20.0, 192.168.21.0, ..., 192.168.27.0
#
sshd2 : 192.168.20.0/255.255.248.0

# 允许从主机blynken到转发端口1234的连接
sshd fwd-1234 : blynken.sleepy.net

# 拒绝其他连接
#
ALL : ALL : DENY

```

我们允许从一个特定的主机`blynken.sleepy.net`访问转发端口1234。请注意，这台主机不一定得在上面列出的网络范围内，它可以在任何地方；上面列出的这些规则只说允许哪些，并没有禁止任何连接。所以举例来说，用`ssh1 -L1234:localhost:21 remote`命令建立的转发只能由本地主机访问，因为SSH1缺省在任何情况下只绑定在回环地址上。但`ssh1 -g -L1234:localhost:21 remote`也能让`blynken.sleepy.net`

访问，这样就好像每个命令都用的是 *ssh2*（因为 *ssh2* 不受本地主机限制，且忽略 *-g* 选项）。重要的区别在于，用了 TCP-wrappers 之后，*sshd* 将拒绝所有从其他地址到转发端口 11234 的连接。

sshfwd-x11 那行将 X 转发连接限制在本地主机上。这意味着如果 *ssh* 用 X 转发到这台主机的连接，那么只有本地的 X 客户端才能使用转发的 X 连接。这一功能 X 认证已经实现了，不过这样设置可提供额外的防护措施（注 7）。

最后一行拒绝不符合上面所有行的任何连接，这样，就使缺省状态变成关闭。如果你想拒绝某些特定连接，但允许所有其他连接，可这样用：

```
ALL : evil.mordor.net : DENY
telnetd : completely.horked.edu : DENY
ALL : ALL : ALLOW
```

最后一行从技术上讲不是必要的，但明确指明你的意图会更好。如果 *host_options* 语法不可用，那么可以换用一个空的 *hosts.allow* 文件，并在 *hosts.deny* 中加入下面的内容：

```
ALL : evil.mordor.net
telnetd : completely.horked.edu
```

9.4.2 TCP-wrappers 注意事项

使用 TCP-wrappers 要注意以下几点：

- 我们无法区分端口是用 SSH1 还是用 SSH2 转发的：*sshd fwd-** 这条规则可同时指向这两者。解决的办法可以是让每一条规则分别指向不同的 *libwrap.a*，这几个文件编译的时候就设置不同的名字，以区分是用于 *hosts.allow* 还是 *hosts.deny*；还有一个办法就是直接给 *ssh* 及 *sshd* 可执行文件打补丁。但就得维护这些改动和额外的文件。
- TCP-wrappers 有一个很大的缺点，它同时影响所有的用户。个别用户无法为自

注 7：SSH2 2.1.0 有一个 bug，至少在某些 Unix 系统中，会引起 SSH 会话在根据 TCP-wrappers 规则拒绝某个转发连接之后挂起。在此 bug 修正之前，不要用 TCP-wrappers 保护转发端口（不过看起来能限制访问主 *sshd2* 服务器）。

已定制访问规则；每台机器只有单独一组全局配置文件。这就制约了其在多用户机上的使用。

- 如果用 `--with-libwrap` 选项编译 SSH，那么 TCP-wrappers 就会一直处于自动开启状态；没有任何配置或命令行能关闭 TCP-wrappers 检查。请记住，SSH 检查的不仅是转发的端口和 X 连接，还有直接连到主 SSH 服务器的连接。只要安装了带 TCP-wrappers 的 `sshd` 版本，就必须确保 TCP-wrappers 的设置能允许连接主服务器，例如，在 `/etc/hosts.allow` 中增加规则 `sshd1 sshd2 sshd : ALL`。
- 如果在 TCP-wrappers 的规则中不用地址，而是用主机名，就会引发一种常见的安全问题。虽然名字用起来更方便，还可以避免将来主机地址改变引起的问题。但在另一方面，攻击者却能摧毁名字服务器，以此绕过访问控制机制。如果主机只用其自身的 `/etc/hosts` 文件查找名字，那么即使在高度安全的环境中也可能发生这类攻击。
- TCP-wrappers 包中包括一个称为 `tcpdchk` 的程序。这个程序检查 wrapper 控制文件，并报告其中可能出现问题的不一致性。很多站点都把这个程序作为一种安全检测的手段周期性地运行。不过设计 `tcpdchk` 的时候就是只根据 `inetd.conf` 做特定的 wrap。没有任何方式能令 `tcpdchk` 知道其他利用 `libwrap` 例程访问控制文件的程序，如，`sshd`。当 `tcpdchk` 检查包含 SSH 规则的控制文件时，会发现其中使用了 `sshd1`、`sshd fwd-n` 之类的服务名，但是在 `inetd.conf` 中没有对应的服务，因此这时 `tcpdchk` 会发出一条警告。现在还没有什么解决办法。

9.5 小结

本章讨论了 SSH 端口转发及 X 转发。端口转发是一种基本的 TCP 代理机制，它通过 SSH 会话建立 TCP 连接隧道。这可用于保护原本不安全的基于 TCP 的协议，或者让被防火墙拒绝的 TCP 连接从隧道中穿过。X 转发是端口转发用于 X Window 的特例，SSH 为其提供特别支持。这样用 SSH 保护 X 连接就变得容易了，因为 X 虽然很流行、很有用，但它的不安全可是臭名昭著的。通常只能在粒度较粗的层次上控制对转发端口的访问，不过用 TCP-wrappers 功能可实现粒度更细的控制。

第十章

推荐配置

本章内容：

- 基础知识
- 编译时配置
- 服务器范围配置
- 密钥管理
- 客户端配置
- 远程主目录(NFS, AFS)
- 小结

前面已经花了那么多章节来介绍 SSH 配置问题，不知你现在是否被弄得晕头转向了？有这么多种可选的办法，你可能会想究竟哪一种才是你应该用的呢？系统管理员该如何利用 SSH 最有效地管理他们的系统呢？

只要设置得当，SSH 就会工作得很好，用户也不会觉察到，但有时究竟如何设置才好，需要多次尝试之后才能得出结论。但是有些配置软件的办法根本就是错误的，一不小心，就可能给系统带来安全漏洞。

本章我们向你推荐一套编译、服务器配置、密钥管理及客户端配置的方法。假设：

- 在 Unix 机上运行了 SSH。
- 为了系统的安全有时我们只能损失一些灵活性。例如，我们不会告诉你如何仔细维护 *.rhosts* 文件，因为我们建议全面禁用 Rhosts 认证。

当然，一个配置方案不可能覆盖所有的可能性（也就是说所有的可配置点）。这里给出的只是一个配置样例，我们更多是站在安全性的立场上，告诉你如何着手，并涉及一些复杂的问题。

10.1 基础知识

在开始配置之前，首先要确认一下你运行的SSH是不是最新的版本。有些旧版本中有一些众所周知的安全漏洞，很容易被别人利用。我们不仅应该一直使用最新的稳定版本，而且应该及时升级软件或打补丁。（对其他安全软件而言也是如此。）

我们应该永远都记得对和SSH有关的文件及目录进行保护。服务器的主机密钥应该只有root才能读取。每个用户的主目录、SSH文件目录及`.rhosts`和`.shosts`文件只能由该用户所有，其他任何用户都不能读取。

同时请牢记，SSH既不会也不能保证系统不受到任何威胁。它可以保护网络连接，但对其他形式的攻击，如用字典攻击密码数据库，就无能为力了。在一个健全的安全策略中，SSH应该是一个重要的组成部分，但绝不是唯一的组成部分。

10.2 编译时配置

在第四章中我们介绍了很多编译SSH可执行文件时使用的标志。仔细设置某些标志可使服务器达到最高的安全性：

`--with-etcdir=...` (SSH1, SSH2)

确保`etc`目录位于本地硬盘，而非使用NFS加载的分区上。如果SSH通过NFS读取文件，那么所传输内容在网络上都是以明文形式传送的，这会有损于安全性。对主机密钥来说更是如此，因为它未经加密就保存在该目录中了。

`--prefix=...` (SSH1, SSH2, OpenSSH)

与此类似，要确保SSH可执行文件也安装在本地硬盘上，否则如果通过NFS加载，就会在网络上传输时被监听到。

`--disable-suid-ssh` (SSH1)

`--disable-suid-ssh-signer` (SSH2)

我们推荐在服务器范围配置中禁用可信主机认证，因此没必要给`ssh1`及`ssh-signer2`设置setuid权限。

`--without-none` (SSH1)

应该禁用允许不加密传输的“none”算法选项。入侵者只要能控制某个用户账

号 10 秒钟，就能把“Ciphers None”加入其客户端配置文件中，这样该用户客户端的加密机制就悄无声息地取消了。如果你要测试不加密的情况，就应该使用--with-none 选项单独生成一个服务器，并确保仅有系统管理员才能运行它。

--without-rsh (SSH1、OpenSSH)

不推荐将 *ssh* 降级到 *rsh*。可以使用两种方法来禁用这种特性：可以在编译时指定--without-rsh 选项，也可以在客户端使用配置文件进行设置。到底怎么做就由你选择了。

--with-libwrap (SSH1、SSH2)

--with-tcp-wrappers (OpenSSH)

libwrap 对客户端主机连接服务器的权限控制得更加精确。它也使端口转发和 X 转发特性变得更为灵活，理由如下：否则本地转发要么只能用于到本地主机的连接，要么就能用于从所有地方发出的连接。结合使用 *GatewayPorts*（或者 *ssh -g*）和 *libwrap*，就可以将转发访问限制精确到某台特定的主机。[9.2.1.1]

10.3 服务器范围配置

第五章详细讨论了 *sshd* 及配置其运行时行为的办法。现在，我们要决定哪些配置选项对于安全性而言是最重要的。

10.3.1 禁用其他访问方式

SSH 可以为你的系统提供一个安全的前门，但是别忘记还要关闭后门。如果现在还能允许声名狼藉的 *r-* 命令访问系统，那就赶快将其关掉。方法如下：

- 删除 */etc/hosts.equiv* 文件，或者将其修改成一个只读的空文件。
- 禁用 *rshd*、*rlogind* 和 *rexecd*，这可以通过在 */etc/inetd.conf* 文件中删除或注释掉相应的行来实现：

```
# 关闭——不使用它!
#shell  stream  tcp    nowait  root   /usr/sbin/in.rshd  in.rshd
```

做完这些操作之后别忘记重启 *inetd*（例如，*skill -HUP inetd*），这样所有的改动才会生效。

- 告诉用户不要创建*.rhosts*文件。

还可以考虑禁用*telnetd*等其他不安全的登录方式，仅保留SSH。

10.3.2 /etc/sshd_config

现在讨论推荐的*sshd_config*设置。前面忽略了一些与安全性关系不很紧密的关键字，比如*PrintMotd*，这个关键字仅在登录后打印一条消息。至于剩下的其他关键字，可以根据实际系统及需要来决定是否使用。

下面列出的文件可以放在机器本地硬盘的任何地方。出于安全性的考虑，请不要将其放在用NFS加载的分区中。否则，SSH服务器每次访问这些文件，其内容在网络上都是以明文形式传输的。

```
HostKey /etc/ssh_host_key
PidFile /etc/sshd.pid
RandomSeed /etc/ssh_random_seed
```

下面的设置用于控制文件及目录的权限。*StrictModes*的值要求用户保护其与SSH有关的文件及目录，以及其他不能做认证的东西。*Umask*值可以保证所有*sshd*创建的文件和目录都只能由其所有者(*sshd*运行的uid)读取。

```
StrictModes yes
Umask 0077
```

下面的代码是服务器的TCP设置。*Port*及*ListenAddress*的值都是标准的。设置空闲超时时间的目的是降低入侵者盗用无人看守的终端的机会。15分钟对使用而言不算太长，对用户而言也不会短得让他们厌烦，不过这还是取决于使用的习惯。你当然可以根据自己的判断设置不同的值，不过一定得仔细考虑。我们还启用了*keepalive*消息，这样在客户端死机或因为其他情况无法访问时，连接就会终止，而不会长期挂起，等待系统管理员手工将其关闭。

```
Port 22
ListenAddress 0.0.0.0
IdleTimeout 15m
KeepAlive yes
```

登录时成功认证的时限为30秒，这对于用户输入和自动执行方式而言都足够长了。

```
LoginGraceTime 30
```

下面的设置控制了服务器密钥的生成方式。我们推荐使用至少768位的服务器密钥，并且最少一小时（3600秒）就重新生成一次。

```
ServerKeyBits 768  
KeyRegenerationInterval 3600
```

下面的设置控制了认证的方式，这里仅启用了公钥认证。禁用密码认证的原因是密码比公钥更容易窃取和使用。这样的限制相当严格，所以根据需要你也可以打开密码认证。没有密码认证时，会碰到“鸡生蛋还是蛋生鸡”的问题：用户第一次怎样才能安全上传公钥呢？系统管理员必须为传输公钥制定一个标准的过程：例如，用户可以在客户端机器上生成密钥，然后请求管理员将其安装在服务器上。由于Rhosts认证会受到欺骗攻击，因此将其禁用。同时也禁用了RhostsRSA认证，因为总体而言这种方法的安全程度处于中等，我们介绍的是安全性更高的方案。

```
PasswordAuthentication no  
RhostsAuthentication no  
RhostsRSAAuthentication no  
RSAAuthentication yes
```

虽然我们已禁用了可信主机认证，但在这里依然要完全禁止`sshd`使用`.rhosts`文件（以免你又启用了可信主机认证）：

```
IgnoreRhosts yes  
IgnoreRootRhosts yes
```

禁用`UseLogin`，以防万一使用了其他登录程序。（这也不太管用。如果入侵者能安装其他登录程序，也许就能编辑`sshd_config`，更改此设置。）

```
UseLogin no
```

下面的设置限制访问服务器的权限，仅允许本地域（注1）内主机建立SSH连接（`fred`账号除外，该账号能接受来自任何地方的连接）。如果想控制特定的本地账号或Unix组的访问权限，就使用`AllowUsers`及`AllowGroups`（或`DenyUsers`及`DenyGroups`）。我们已经设置了`SilentDeny`，因此任何被`DenyHosts`拒绝的连接都不会向用户返回消息。不必告诉攻击者发生了什么，不过这同时也增加了排除问题的难度。

注1：此限制的可靠程度取决于DNS的健壮程度。不过由于`AllowHosts`的使用，基于IP地址的限制已经不再安全了。

```
AllowHosts fred@* *.your.domain.com      只是个例子  
SilentDeny yes
```

我们允许超级用户通过 SSH 连接服务器，但不能用密码认证方式。这有些多余，但和禁用 PasswordAuthentication 是一致的。

```
PermitRootLogin nopwd
```

对于错误消息日志，我们禁用 FascistLogging 方式，因为它会在日志中记录用户特定的信息，例如每个用户登录进来的日期和时间，而这些信息对攻击者都非常有用。同时也禁用 QuietMode 方式，这样记录的日志就更详细，敏感程度却更低。

```
FascistLogging no  
QuietMode no
```

我们允许进行 TCP 端口转发及 X 转发，这样用户就能保护其他的 TCP 连接了：

```
AllowTcpForwarding yes  
X11Forwarding yes
```

10.3.3 /etc/ssh2/sshd2_config

现在介绍推荐的 *sshd2_config* 设置。再次说明，这里忽略了一些与安全无关的关键字。

如前所述，所有与 SSH 有关的文件都应该位于本地磁盘上，不能将其放在从远程加载的分区中：

```
HostKeyFile /etc/ssh2/hostkey  
PublicHostKeyFile /etc/ssh2/hostkey.pub  
RandomSeedFile /etc/ssh2/random_seed
```

对于下列设置，请仔细考虑用 NFS 文件系统存储用户文件的优缺点：[10.7]

```
UserConfigDirectory  
IdentityFile  
AuthorizationFile
```

下面这样设置的原因请参看 SSH1：

```
StrictModes yes
```

下面的前三项设置的依据与 SSH1 相同，然而 `RequireReverseMapping` 的设置就有些技巧了。你可能会想，对接人的连接进行 DNS 反向查找能增强安全性，但事实上 DNS 本身不够安全，不能保障查找结果的正确性。还有，由于 Unix 及网络环境中的其他一些因素，反向 DNS 映射可能根本就不能正常工作。

```
Port 22
ListenAddress 0.0.0.0
KeepAlive yes
RequireReverseMapping no
```

下面这样设置的原因请参看 SSH1：

```
LoginGraceTime 30
```

此外，由于 `sshd2` 没有设置服务器密钥的位数的关键字，用户得在启动时使用 `-b` 选项：

```
$ sshd2 -b 1024 ...
```

下列设置与 SSH1 类似：

```
AllowedAuthentications publickey
RequiredAuthentications publickey
```

禁用 `UserKnownHosts` 可防止用户给未知主机提供信任权限，以便使用可信主机认证。超级用户仍可在 `/etc/ssh2/knownhosts` 中指定可信主机。

```
IgnoreRhosts yes
UserKnownHosts no
```

下面这样设置的原因请参看 SSH1：

```
PermitRootLogin nopwd
```

根据需要从下面两项中任选一项。请注意，其中每个都不能使用 `none cipher`；前面讨论 `--without-none` 时曾提到，`none cipher` 是一个安全隐患。

```
Ciphers anycipher
Ciphers anystdcipher
```

下面的选项可产生足够多的有用日志信息：

```
QuietMode no  
VerboseMode yes
```

由于SSH-2协议的安全性更高，因此我们就禁用SSH-1兼容模式。不过出于实用的原因，我们也可以启用此模式，启用之后还要将 `Sshd1Path` 指向 SSH1 服务器可执行文件所在的路径。

```
Ssh1Compatibility no  
#Sshd1Path /usr/local/bin/sshd1 #注释掉
```

10.4 每账号配置

我们应该告知用户，不要创建 `.rhosts` 文件。如果在本地 SSH 服务器上启用可信主机认证，那么建议用户尽量使用 `.shosts` 文件代替 `.rhosts`。

对于 SSH1 和 OpenSSH，`~/.ssh/authorized_keys` 中的每一个密钥都必须用适当的选项对其进行限制。首先，我们在适当的时机使用 `from` 选项限制只能从特定的主机访问特定的密钥。例如，假设 `authorized_keys` 文件中包含你家里那台 PC (`myhome.isp.net`) 的公钥。而其他机器根本不可能用那个密钥来认证，就可以明确地限定这一关系：

```
from="myhome.isp.net" ...key...
```

还要对适当的密钥设置空闲超时时间：

```
from="myhome.isp.net",idle-timeout=5m ...key...
```

最后，考虑每一个密钥是否需要对到达的连接使用端口转发、代理转发以及分配 `tty` 等。如果不需要，就可以分别用 `no-port-forwarding`、`no-agent-forwarding` 和 `no-pty` 禁用这些特性。

10.5 密钥管理

我们推荐创建至少 1024 位长的用户密钥，并用好的口令对密钥进行保护。口令要足够长，而且应该混用小写字母、大写字母、数字及其他符号字符。不要用能在字典中找到的单词。

除非绝对需要使用空口令（比如在自动批处理脚本中使用时），否则就不要使用。

10.6 客户端配置

多数SSH安全问题都与服务器有关，但SSH客户端也有一些与安全性有关的设置。这里有一些小建议：

- 一旦离开一台正在运行SSH客户端的计算机，就必须用有密码保护的屏保锁定显示器。如果正在运行代理，这一点就尤其重要，因为代理能让入侵者不用口令就可以访问远程账号。
- 在客户端配置文件中启用某些安全功能，并将其设置为最强：

```
# SSH1, OpenSSH
# 放于配置文件的顶部
Host *
    FallBackToRsh no
    UseRsh no
    GatewayPorts no
    StrictHostKeyChecking ask

# 仅对 SSH2
# 放于配置文件的底部
#
    GatewayPorts no
    StrictHostKeyChecking ask
```

FallBackToRsh和UseRsh的设置可以禁止SSH在你没觉察的情况下使用不安全的r-命令。（SSH2没有这个问题。）GatewayPorts的设置禁止远程客户端连接本地的转发端口。最后一个StrictHostKeyChecking可在主机密钥发生变化时提醒你，请求你的处理意见，因此能防止盲目地进行连接。

10.7 远程主目录 (NFS, AFS)

我们在讨论SSH设置的安全隐患问题时曾多次提到NFS，现在深入讨论这个话题。

如今网络如此普及，通过网络文件共享协议在多台计算机之间共享用户主目录也是很平常的事情，例如用SMB为Windows共享数据，用NFS及AFS为Unix共享数据。这样非常方便，但同时也产生了一些与SSH有关的技术和安全问题。

SSH会检查目标账号的主目录中的文件，根据这些文件作出关键的认证及授权决定。除密码认证之外的每一种形式，都是通过主目录下这些不同的控制文件来控制SSH对你的账号的访问权限的。因此有两件事情非常重要：

- 必须保证主目录的安全，令其远离攻击。
- SSH必须有权访问主目录。

10.7.1 NFS 安全隐患

共享主目录的安全性通常不是很高。尽管有些版本的NFS协议及产品可以提供更强的安全保障，但多数环境中的NFS都不安全。通常情况下NFS根本没有任何可靠的认证方式，其认证模式与rsh相同：它使用源IP地址和DNS对客户端进行验证，判断可靠性的唯一证据就是特权端口。然后它就可以获得NFS请求中的uid号，从而获得访问该用户的权限。这样以来，对主目录进行攻击就会变得非常简单：

1. 窃取NFS请求中的uid号，在一台运行Unix的计算机上创建一个uid相同的账号。
2. 盗用某个可信任的主机IP地址，连接网络上的那台主机。
3. 执行mount命令，在具有那个uid的账号之下执行su，就可以开始偷窃文件了。

这时入侵者要想在*authorized_keys*中加入其他公钥也是很容易的，这样这个账号就完全公开了。所以在设计一个系统时要牢记，只有用户主目录的安全性能得到保证，SSH的安全性才有立足之地。至少我们应该了解这种情况下如何在安全与方便之间达到平衡。如果你用的是不安全的NFS，那么可以使用以下方法解决这些缺陷：

- 使用SSH2，利用其UserConfigDirectory选项存放每个用户的SSH配置文件，通常是`~/.ssh2`，不然就是`/var/ssh/<username>`。你依然可以把这些目录的权限设置成由其所有者进行控制，然而不再通过NFS共享目录了，因此也就没有危险。SSH1或OpenSSH没有这样的选项，倒是可以修改源代码实现。
- 禁用所有基于主机的认证方式，因为`~/.shosts`控制文件不安全，也没法改变其存放位置。或者，如果真想用这些认证方式，就得设置IgnoreRhosts选项。这样`sshd`会忽略`~/.shosts`，仅依赖服务器级的`/etc/shosts.equiv`文件。

- 如果你是个技术狂热爱好者，可以禁用 Unix 主机的交换（swapping）功能。否则，诸如服务器、主机、用户密钥或密码之类的敏感信息就会写入磁盘，它是作为 Unix 虚拟内存系统的一部分写入磁盘的（这样的话，*sshd* 也会交换到磁盘上）。如果有 root 权限（同时也要具备相当的知识和运气），就可能读取交换分区的内容，从里面一大堆东西中找到这些信息——当然这是非常困难的。另外一种选择是使用对磁盘上的交换页进行加密的操作系统，如，OpenBSD。

10.7.2 NFS 访问问题

SSH 与 NFS 一起使用时，还可能产生一种访问权限的问题。在公钥或可信主机方式中，如果每账号控制文件在缺省位置，*sshd* 必须读取目标账号的主目录，这样才能执行认证操作。如果该目录与 *sshd* 在同一主机上，那没有什么问题。*sshd* 以 root 身份运行，因此可以访问所有文件。然而，如果这个目录是通过 NFS 从别的地方加载上来的，*sshd* 就未必能访问。NFS 的通常设置方式是设定 root 账号具有的特权不能扩展到远程文件系统上。

目前这项限制并非真的那么严格。由于 root 有一项特权是以任何 uid 创建进程，那么只要“变成”恰当的用户就能访问远程目录了。SSH1 及 SSH2 运用了这一机制，但 OpenSSH 现在还没有。^[3.6]

你也可以自己解决这个问题，但是这样的话就必须将你的 *authorized_keys* 文件变成全部用户都可读的；因为让 root 远程访问的惟一办法是让所有人都能访问。这也没有什么不好的。*authorized_keys* 文件里没什么秘密；然而你可能还是想防止别人知道允许哪一个密钥访问账号，不然相当于告诉攻击者可以窃取哪个密钥了。然而，如果要保证 root 能访问配置文件，你就必须将主目录及 *~/.ssh* 设成全局可检索的（即，权限至少是 711）。这样其他用户没法偷走文件的内容，但是他们还是可以猜测文件名，并验证其猜测。这也意味着必须小心设置文件的访问权限，因为目录如果有最高访问权限，就不能阻止其他用户访问了。

这一切既可能完全无法接受，也可能根本没有问题；这取决于你对自己的用户主目录所在的那台主机上的文件和其他用户的态度。

10.7.3 AFS 访问问题

Andrew 文件系统 (Andrew File System, AFS) 是一个与 NFS 目的相似的文件共享协议，但它考虑问题更精细。它使用 Kerberos-4 进行用户认证，总体而言比 NFS 更安全。前面讨论的访问权限问题 AFS 也有，不过解决起来更费功夫，这里用 OpenSSH 更容易设置。

由于 AFS 使用了 Kerberos，因此访问远程文件的权限受到适当的 Kerberos 证书的控制。现在 root 不能再切换成其他 uid 了；*sshd* 必须获取正确有效的 AFS 证书才能访问主目录。如果已经登录进远程机器，那当然就可以用 Kerberos 及 AFS 命令得到一个证书。不过 *sshd* 要在登录之前就要用证书，因此问题变得有些麻烦。

当然并不是只有 SSH 会碰到将证书从一台主机传到另一台主机的问题，所以已经有一个解决办法了：证书转发。这需要一些特殊的 support 功能，因为仅仅将证书拷贝到远程主机上是不够的；证书是针对特定主机发出的。一般情况下 Kerberos-4 没有证书转发功能（不过 Kerberos-5 中有），但 AFS 为 Kerberos-4 的 TGT 及 AFS 访问证书定制了这项功能，并且 OpenSSH 可以自动执行这种转发。使用方法是，用 `--with-kerberos` 及 `--with-afs` 编译 SSH 客户端及服务器，在每一端都启用 AFS-TokenPassing（缺省值就是启用）。然后，如果你用 SSH 登录的时候有 Kerberos-4 及 AFS 证书，这些证书就会自动传到 SSH 服务器上，*sshd* 就可以据此访问你的远程主目录，执行公钥认证或可信主机认证。

如果没有用 OpenSSH，那么在 AFS 环境中使用 SSH 就会有问题。在 Internet 上有很多人开发了给 SSH1 增加 AFS 转发功能的补丁（注 2），不过我们还没机会试验这些程序。

10.8 小结

SSH1、SSH2 及 OpenSSH 非常复杂，选项众多。在安装和运行 SSH 服务器及客户端时很有必要理解所有的选项，这样才能保证 SSH 的运行方式与你的安全策略是一致的。

注 2：例如 Dug Song 的 *ssh-afs* 补丁；请参看 <http://www.monkey.org/~dugsong/ssh-afs>。

我们介绍的推荐配置是一种安全性很高的设置方式，而你的实际需求可能会有所不同。例如，你也许需要灵活使用一些我们的设置里禁用的认证方法。

第十一章

案例分析

本章内容

- 无人值守的 SSH：批处理或 cron 任务
- FTP 转发
- Pine、IMAP 和 SSH
- Kerberos 与 SSH
- 路由器连接

本章中，我们将深入分析几个高级主题：复杂的端口转发、SSH 与其他应用程序的集成等等。SSH 有很多有趣的特点，只有仔细研究才能发现，所以我们希望读者能从这些例子中学到许多东西。现在，卷起袖子开始干吧！祝你开心！

11.1 无人值守的 SSH：批处理或 cron 任务

SSH 不仅是一种交互性很强的工具，而且还能进行自动操作。只要使用得当，批处理脚本、*cron* 任务以及其他自动化任务都能很好地利用 SSH 的安全机制。最大的挑战来自于认证：如果没有人在那儿输入密码或者口令（从现在起这两个名词将统称为“密码”），客户端怎么能让服务器知道它的身份呢？必须小心地选择一种认证机制，然后同样小心地使用它。一旦建立好基础框架，必须使用适当的方式来调用 *ssh* 命令，而不能要用户再输入密码之类的内容。在本例中，我们讨论不同认证方法在无人参与的 SSH 客户端上运行的利弊。

注意：任何形式的无人参与的认证都会引发安全问题，这都需要进行折衷处理，SSH 也不例外。无人参与意味着如果需要提供证据（输入密码、按手印等），这些证据肯定得一直存储在主机系统中。因此，攻击者如果得到足够的访问权限，就能获取这些证据，可以以真正用户的身份执行所有的程序，访问所有的资源。我们必须在理解技术优缺点的基础上正确选择使用哪种技术；不过这仍然是饮鸩止渴。如果你无法接受这一点，也就不能指望无人参与的 SSH 能提供多大的安全保证。

11.1.1 密码认证

记住一条最重要的规则：如果想保证批处理任务的安全性，就别用密码认证。要想在批处理中使用密码认证，那么密码就必须嵌入到批处理脚本或脚本能读取的文件中。不管你怎么做，别人只要一读这个脚本，就能发现密码在什么地方。我们不提倡使用这种技术，而是推荐采用更安全的公钥认证。

11.1.2 公钥认证

在公钥认证中，客户端登录的证据是私钥。因此，批处理任务需要访问私钥，必须将其存储在批处理任务有权访问的地方。可以使用三种方法来保存私钥，后文中我们会分别对其进行介绍：

- 将加密的私钥及口令存储在文件系统中。
- 在文件系统中存储私钥的明文（未经加密），这样就不需要口令了。
- 将密钥保存在代理中，这样就可以摆脱文件系统的限制，从而保证密钥的安全性，不过在系统启动时需要有人给私钥解密。

11.1.2.1 在文件系统中存储口令

使用这种方法，可以把加密的私钥及其口令存储在文件系统中，这样脚本就可以访问这个私钥。我们并不推荐这种方法，因为在文件系统中存储未加密的密钥，其安全等级与这种方法完全相同，却远没有它复杂。在任何一种情况下，都只能依赖文件系统来保护密钥的安全性。这也正是选择下一个方法的原因。

11.1.2.2 使用明文密钥

访问明文或者未加密的密钥不需要口令。要创建明文密钥，可以运行 `ssh-keygen`，要求输入口令时直接按回车键（或者类似地，运行 `ssh-keygen -p`，去掉已有密钥的口令），然后用命令行 `ssh -i` 或在客户端配置文件中使用 `IdentityFile` 关键字给出密钥文件名。[\[7.4.2\]](#)

通常情况下不推荐大家使用明文密钥，这与把账号密码保存在账号的文件中没什么两样。在交互式登录过程中这样做永远都不是什么好办法，因为 SSH 代理可以提供

同样的便利，不过却安全得多。但是，在自动操作中可以使用明文密钥，因为此时没有人会输入口令，必须依赖系统中某些永久性的东西，例如文件系统。

不论存储的是明文密钥、加密密钥与口令，还是密码，这三种方法从某种意义上讲是相同的；但即使如此，由于以下三个原因，我们还是推荐使用明文密钥：

- SSH在服务器端对公钥认证的控制要比密码认证好得多，这在设置批处理任务时显得尤为重要；下面会简要进行讨论。
- 在所有其他条件相同的情况下，公钥认证比密码认证更安全，因为它不会把认证的机密经由恶意主机泄漏给窃听者。
- 用其他程序给 SSH 提供密码很不方便。SSH 设计的初衷就是只从用户那里接收密码：它并不通过标准输入设备来读取密码，而是直接打开自己的控制终端与用户进行交互。如果没有终端，SSH 就会出错。要想在其他程序中使用这一机制，必须用伪终端与 SSH 交互（例如，使用类似 Expect 的工具）。

尽管如此，明文密钥还是一种非常危险的方法。攻击者只需突破文件系统的保护就可以窃取密钥，而这并不一定需要什么黑客技巧：只需偷走一个备份的磁带即可。所以在多数情况下，我们推荐下面这种方法。

11.1.2.3 使用代理

*ssh-agent*提供了另一种存储密钥的方法，来支持批处理任务，它在某种程度上不容易受到攻击。只需要有人调用代理，从使用口令保护的密钥文件中加载所需的密钥即可，这个过程只需执行一次。此后，无人参与的任务就可以一直使用这个代理进行认证。

本例中，密钥仍然是明文的，不过它存储在代理运行的内存空间中，而不是存储在磁盘文件上。通常的破解技术容易做到非法访问文件，但是从正在运行的进程的地址空间中提取一个数据结构却是非常困难的。这种方法同时也避免了入侵者从备份磁带中得到明文密钥的问题。

不过，如果攻击者突破了文件系统的访问限制，这种方法仍会危及密钥的安全。代理中有一个 Unix 域套接字，用户通过它访问代理提供的所有服务。在文件系统中，此套接字表现为一个节点。任何人只要能读写该套接字，就可以命令代理对其认证

请求进行签名，这也就等于窃用了密钥。但是这样的危险并不很大，因为攻击者不能通过代理套接字得到密钥本身，只有代理正在运行，而且依然能对主机进行访问时，他才能使用该密钥。

使用代理有一个缺点：系统重启之后无人参与的任务就不能继续执行。当主机自动重新开始运行后，如果没有人手工再次运行代理，并用口令加载密钥，那么批处理任务就无法获取密钥。这只是增强安全性的代价而已。使用代理另一点复杂的地方是：必须要给批处理任务安排好找到代理的方法。SSH客户端通过一个指向代理套接字的环境变量定位代理，在SSH1和OpenSSH中这个环境变量叫SSH_AUTH_SOCK。[\[6.3.2.1\]](#)要执行批处理任务，你必须在启动代理时记录其输出，以便对其进行定位。例如，用shell脚本启动代理时，可以将环境变量存成一个文件：

```
$ ssh-agent | head -2 > ~/agent-info
$ cat ~/agent-info
setenv SSH_AUTH_SOCK /tmp/ssh-res/ssh-12327-agent;
setenv SSH_AGENT_PID 12328;
```

此时可以向代理中加载密钥（此处假定用C shell语法格式）：

```
$ source ~/agent-info
$ ssh-add batch-key
Need passphrase for batch-key (batch job SSH key).
Enter passphrase: *****
```

然后构造任意一个脚本，给环境变量赋相同的值：

```
#!/bin/csh
# 根据agent-info文件访问ssh-agent
set agent = ~/agent-info
if (-r $agent) then
    source $agent
else
    echo "Can't find or read agent file; exiting."
    exit 1
endif
# 现在使用SSH……
ssh -q -o 'BatchMode yes' user@remote-server my-job-command
```

还必须确保只有批处理任务才能读写该套接字（其他任何任务都不行！）。如果只有一个uid能调用代理，那么最简单的方法就是在此uid下启动代理（例如：在root下执行`su <batch_account> ssh-agent`）。如果多个uid使用代理，就必须调整套接字及其上层的目录的访问权限（也许可以用组来授权），让这些uid都可以访问该套接字。

注意：某些操作系统在进行 Unix 域套接字的权限设置时表现得很奇怪。比如：某些版本的 Solaris 完全不管套接字的模式，给任何进程都赋予完全的控制权限。要保护这样的套接字，禁止某些访问，就得对包含它的目录进行设置。举个例子来说，如果包含套接字的目录的模式为 700，那就只有目录所有者才能访问该套接字。（这是假设其他地方没有指向套接字的快捷方式，如硬链接等。）

基于代理的自动化操作比基于明文密钥的更复杂，受到的限制更多；不过这种方式对攻击的抵抗力更强，它不会把密钥留在磁盘、磁带等会被偷走的介质上。然而，考虑到错用文件系统仍会使代理易受攻击，而且一般假设代理都是无限期运行的，因此，这种方法的优越性就得打上个问号。尽管如此，我们还是推荐使用代理方法，在安全性要求很高的环境中，使用代理是最安全、最灵活的 SSH 自动化策略。

11.1.3 可信主机认证

如果对安全性的要求相对较低，就可以考虑用可信主机方式对批处理任务进行认证。在这种情况下，认证的凭据就是操作系统中进程的 uid：这是正在运行的进程的身份标识，它决定了进程对受保护对象操作的权限。攻击者只需设法控制一个在你的 uid 下运行的进程，就可以伪装成你访问远程 SSH 服务器。如果他攻破了客户端的 root，那么这就特别简单了，因为 root 可以在任意 uid 下创建进程。但是真正的麻烦是客户端的主机密钥：如果攻击者得到它，就可以伪装成任何用户，发出伪造的认证请求，而 *sshd* 会信以为真。

可信主机认证从很多方面来看都是最不安全的 SSH 认证方式。[\[3.4.2.3\]](#) 系统一旦受到损害，就很容易将其传播出去：如果攻击者在主机 H 上获得某个账号的控制权，那么他不用再花什么力气，立刻就在所有信任 H 的主机上获得了同一账号的控制权。而且，可信主机的设置方式不仅有限、脆弱，而且很容易出错。公钥认证在安全性和灵活性方面都比它强，尤其是你可以在强制命令和认证文件的其他选项中限制可调用的命令和连接上来的客户端。

11.1.4 Kerberos

Kerberos-5 [\[11.4\]](#) 采用可更新许可证，支持任务长期运行。尽管这一点在 SSH 中没有明确地支持，但我们可以把批处理任务设计成使用 Kerberos。这与使用代理的方法相同，操作员首先手工执行 *kinit* 命令，用 -r 开关为执行批处理的账号申请一个

TGT（可更新许可证 Ticket-Granting-Ticket）。然后批处理命令用 *kinit -R* 周期性地更新 TGT，以免其过期。此过程可以一直重复，直到许可证到达最大可更新生命周期为止，这一般有几天时间。

不过，SSH 支持的 Kerberos 也像可信主机认证一样，缺乏公钥认证那种严密的认证控制选项。即使在一个使用 Kerberos 对用户进行认证的系统中，也最好换用某种形式的公钥认证来控制无人参与的任务。有关可更新证书的更多信息，请参看 Kerberos-5 文档。

11.1.5 批处理任务的一般安全预防措施

无论选择什么（认证）方法，总有些额外的预防措施能帮助你保护你的环境。

11.1.5.1 给账号最低权限

运行批处理任务的账号应该只有运行该任务所需的最低权限，不能再多了。不要只贪图方便就用 root 运行所有的批处理任务。合理地安排文件系统和其他保护措施，这样就可以用更低权限的用户执行批处理任务了。记住：无人参与的远程任务增大了账号泄密的风险，所以一定不能怕麻烦，只要有可能就别用 root 账号。

11.1.5.2 使用专用、被锁定的账号执行自动化操作

创建一些只用于执行自动化操作的账号。尽量避免在用户账号下运行系统批处理任务，因为你可能无法把一个账号的权限降到支持任务所需的最低程度。在很多情况下，自动操作账号甚至不需要交互式登录的权限。如果在一个账号下执行的任务是由批处理任务管理器（比如，*cron*）直接生成的，那么这个账号就不需要密码，应该被锁定。

11.1.5.3 限制使用密钥

尽可能多地限制目标账号，令其只能执行必需的工作。在公钥认证中，自动化任务不应该与交互登录共享密钥。因为可能某天你会因为安全的原因更换密钥，而又不能因此影响到其他的用户或任务。最强的控制方式是给每一个自动化任务单独设置一个密钥。另外，要利用认证文件中的选项给密钥加上所有可能的限制。

[8.2]command选项限制密钥只运行所需要的远程命令，from选项将使用权限限制在适当的客户主机上。只要不与任务冲突，最好总是加上下面这些选项：

```
no-port-forwarding, no-X11-forwarding, no-agent-forwarding, no-pty
```

这样即使密钥被盗用，也很难将其滥用。

如果你正在使用可信主机认证，就没法使用这些限制手段了。此时，最好给该账号指定一个单独的shell，然后用shell限制能运行的命令。这是一种有效的限制方式，因为用户指定的任何命令sshd都是用目标账号的shell执行的。一个标准的工具是Unix的“restricted shell”。restricted shell通常也称为“rsh”，但它却和Berkeley r命令中打开一个远程shell的命令“rsh”毫无关系。这很容易混淆。

11.1.5.4 一些有用的 ssh 选项

如果在批处理任务中运行SSH命令，加上下面这些选项效果会很好：

```
ssh -q -o 'BatchMode yes'
```

-q是安静模式，防止SSH打印警告信息。如果把SSH当作从一个程序到另一个的管道来运行，那么此选项有时非常有用。否则，SSH的警告信息可能被本地程序当成远程程序的输出，造成混乱。[7.4.15]

关键字BatchMode告诉SSH不要让用户确认，因为在执行批处理时用户不在现场。这样出错报告变得更加直接，也看不到“访问tty失败”之类令人费解的SSH消息了。[7.4.5.4]

11.1.6 建议

我们推荐使用最安全的方法操作无人参与的SSH：公钥认证加代理存储密钥。如果不能用这种方法，可以使用可信主机或者明文密钥认证代替；在上面讨论的方针指导下，你关心、需要的内容决定了哪种方法更适合用来保障本地安全性。

尽可能用不同的账号和密钥来执行每一个任务。这样，在任何一个账号泄密，或者一个密钥被盗时，系统受到的危害也是有限的。当然，这还要对复杂程度进行权衡；如果有上百个批处理，要给每一个创建不同的账号和密钥，工作量就太大了。在这

种情况下，应按照这些任务各自需要的权限，将其分组，再给每一组任务分别指定账号和口令。

有一个小小的自动操作可以减轻加载多个密钥的负担。可以用同一个口令存储所有的密钥：写一个脚本，先让用户输入口令，再多次运行 `ssh-add`，加载不同的密钥。要么，给密钥设置不同的口令，当需要输入时手工插入一张存储这些口令的磁盘。此口令列表也可以用一个人工输入的密码加密。如果是这样，连密钥本身都不用存在文件系统里，只要保存在磁盘上就可以了：一切取决于你的需要和喜好。

11.2 FTP 转发

有关 SSH 问得最多的一个问题是：“我怎么才能用端口转发实现安全的 FTP？”很遗憾，简单地说这通常不能实现，至少不能完全实现。端口转发可以保护账号的密码，不过通常不能保护正在传输的文件。尽管如此，能保护密码也是巨大的胜利，因为 FTP 最糟糕的问题就是会把密码泄漏给网络嗅探器（注 1）。

本节将详细介绍 FTP 和 SSH 能做什么，不能做什么及其原因。一些困难是 FTP 本身的限制造成的，不仅在与 SSH 交互的时候会发生，在有防火墙以及网络地址转换（network address translation, NAT）存在时也会出现。防火墙和 NAT 如今很常见，因此我们会讨论每种情形；它们的存在也许就是你想用安全的 FTP 转发的原因吧。如果你是负责 SSH 和其他这些网络组件的系统管理员，我们会尝试引导你对整个系统的设计和排疑解难工作建立一般性的理解。

根据网络环境的差异，SSH 与 FTP 结合时会出现不同的问题。我们不可能覆盖每一种可能的环境，只能单独描述每一种问题，说明其症状，并推荐解决的方法。如果许多问题同时出现，软件的表现可能与我们给的例子不同。我们建议你在你的系统中做试验之前，通读整个案例（至少应该粗略读一遍），这样，你对可能遇上问题就会有点儿概念。然后，就可以在你的计算机里试验这些例子了。

注 1： 至少普通的FTP会这样。一些FTP实现支持更多的安全认证方法，如Kerberos。甚至对FTP协议也做了扩充，支持对数据连接加密及密码一致性检查。但是，这些技术还没有广泛实现，因而现在明文密码和未受保护的数据连接仍是Internet上用于FTP服务器的常见形式。

Van Dyke 的 SecureFX (<http://www.vandyke.com/>)

Van Dyke 科技公司 (Van Dyke Technologies) 提供了一套非常有用的 Windows 产品，是为基于 SSH 的 FTP 转发特别设计的，可以转发包括数据连接在内的所有连接。它将 SSH-2 与 FTP 客户端结合在一起。先通过 SSH-2 连接到服务器主机，然后经由一个 SSH-2 会话中的“tcpip-direct”通道连接至 FTP 服务器（仍然在运行 SSH 服务器的主机上）。这与通常的 SSH-2 客户端实现本地端口转发的机制相同，但因为它是专门制作的工具，因此可以与服务器直接对话，而不用使用一个回环地址来转发本地 TCP 端口的连接。

SecureFX 是一个 GUI FTP 终端。在要建立 FTP 数据连接时，它就动态创建任何数据端口所必需的通道或者远程转发机制（对主动式 FTP 来说向外的 tcpip-direct 通道更多些，对被动式 FTP 来说普通的远程转发机制更多些）。这个产品的功能非常好，因此我们向你推荐它。

注意：本书出版时（英文版）SecureFX 只能在 Windows 平台上（98, 95, NT4.0, 2000）使用，所需的服务器是 SSH-2；它不支持 SSH-1。

11.2.1 FTP 协议

你需要知道 FTP 协议的一些情况，这样才能理解 FTP 和 SSH 之间的问题。多数 TCP 服务只涉及单独一条从客户端到服务器固定端口的连接。然而在 FTP 中，这样的连接在两个方向上都有多条，而且多数都连在不可预知的端口号上：

- 一条单独的控制连接，用于从客户端向服务器传输请求和从服务器向客户端传输响应。它连在 TCP 协议的 21 端口，在整个 FTP 会话过程中保持有效。
- 若干数据连接，用于传输文件和其他数据，如列目录。每个文件传输都是一个新的数据连接打开关闭的过程，每一个连接都可能在不同的端口上。这些数据连接既可能发自客户端，也可能发自服务器。

让我们运行一个典型的 FTP 客户端，看看控制连接是什么样子。我们将用调试模式查看客户端在控制连接上发送的 FTP 命令 (*ftp -d*)，这些信息通常不显示出来。调试模式用前置字符串“--->”标识这些命令，比如：

```
---> USER res
```

你也可以看到服务器的响应，这是客户端的缺省方式。这些响应之前有一个数字代码：

```
230 User res logged in.
```

在下面的会话中，用户 `res` 连上某个 FTP 服务器，登录，然后两次试图改变当前目录，一次成功，一次失败：

```
$ ftp -d aaor.lionaka.net
Connected to aaor.lionaka.net.
220 aaor.lionaka.net FTP server (SunOS 5.7) ready.
-> SYST
215 UNIX Type: L8 Version: SUNOS
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> user res
-> USER res
331 Password required for res.
Password:
-> PASS XXXX
230 User res logged in.
ftp> cd rep
-> CWD rep
250 CWD command successful.
ftp> cd utopia
-> CWD utopia
550 utopia: No such file or directory.
ftp> quit
-> QUIT
221 Goodbye.
```

用标准的端口转发就可以保护控制连接，因为它在一个知名端口 21 上。[\[9.2\]](#) 与之不同的是，数据连接的目的端口号通常无法事先知道，因此为其建立 SSH 转发要困难得多。FTP 协议有第二个标准端口号 20，称为 *ftp-data* 端口。但是这仅仅是从服务器向客户端发起数据连接的源端口，上面永远不会有任何监听。

很奇怪，数据连接的方向通常与控制连接相反；就是说，服务器建立一个回到客户端的 TCP 连接，在其上传输数据。这些连接建立的端口是 FTP 客户端与服务器动态协商决定的，协商的方法是通过 FTP 协议传输十分详细的 IP 地址信息。由于这些特性的存在，在转发 SSH 连接和涉及防火墙或 NAT 的其他情况下，普通的 FTP 操作也会有困难。

另一种 FTP 工作模式称为被动模式（passive mode），能用来解决其中一个问题：它令数据连接反向，连接从客户端发起，连入服务器。被动模式是 FTP 客户端的一种行

为，由客户端的设置决定。通常，数据连接的建立是从服务器到客户端的，这叫做主动式FTP，是FTP客户端的缺省设置；不过这种情况正在发生变化。在命令行方式的客户端中，使用`passive`命令就可以切换到被动模式。客户端传给服务器，令其进入被动模式的内部命令叫PASV。在下面几节中，我们将讨论一些具体的问题，以及被动模式是如何解决这些问题的。图11-1总结了被动/主动模式FTP的工作过程。

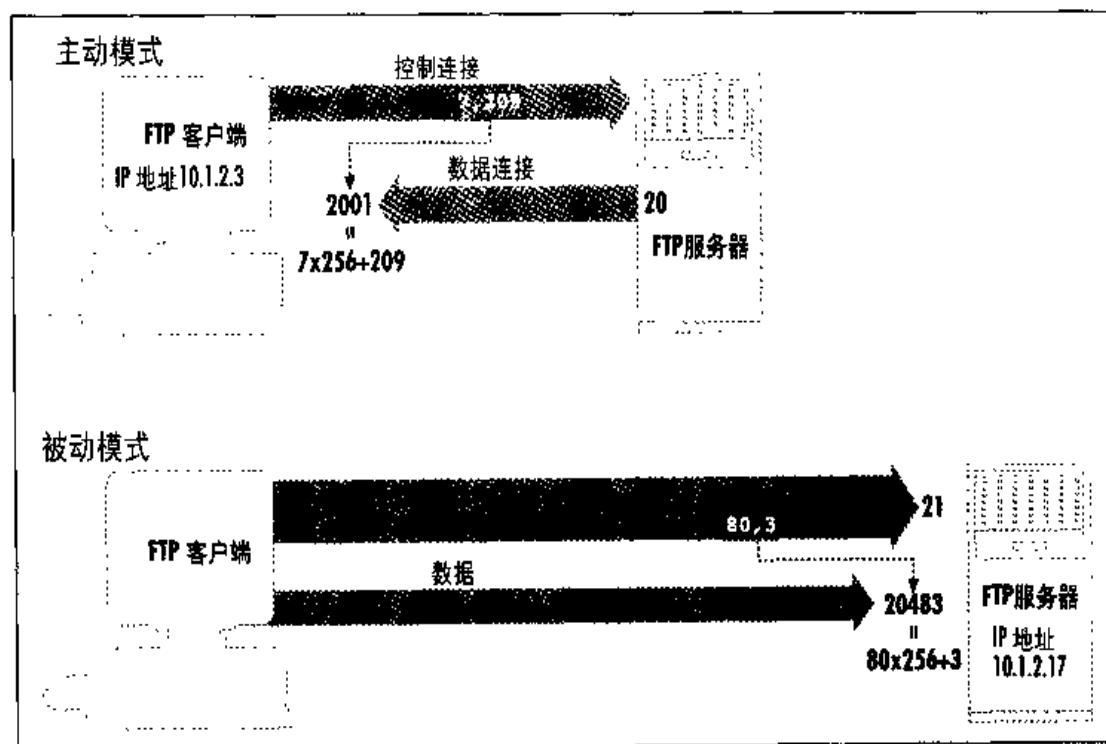


图 11-1：基本 FTP 操作：控制连接及主动 / 被动模式传输对比

11.2.2 转发控制连接

由于FTP控制连接只是已知端口上单一持续的TCP连接，因此可以对其使用SSH转发。通常运行FTP服务器的机器必须同时运行SSH服务器，你必须能通过SSH访问这台机器的一个账号（见图11-2）。

假设你现在已经登录到一台主机`client`上，想与主机`server`上的FTP服务器建立安全连接。为了转发FTP控制连接，你得在`client`上运行端口转发命令（注2）：

注 2：如果你使用另一种流行的FTP客户端`ncftp`，那么运行的命令变为：`ncftp ftp://client:2001`。

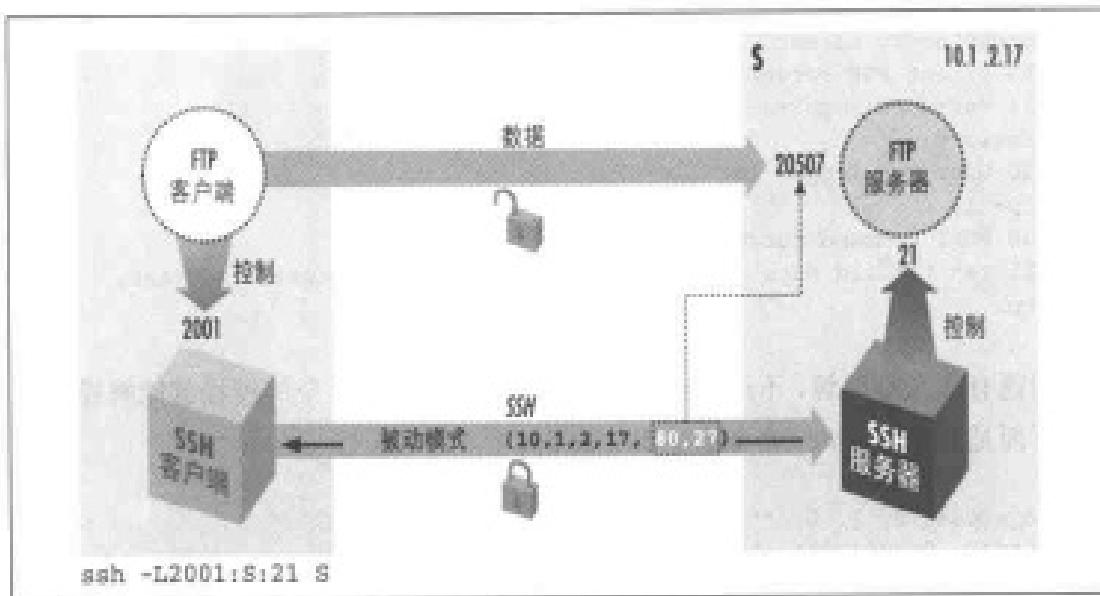


图 11-2：转发控制连接

```
client% ssh -L2001:server:21 server
```

然后，连接到转发的端口上：

```
client% ftp localhost 2001
Connected to localhost.
220 server FTP server (SunOS 5.7) ready.
Password:
230 User res logged in.
ftp> passive
Passive mode on.
ftp> ls
```

使用刚才我们推荐的命令时，要注意两个重要的问题。下面分别讨论。

- 转发的目的主机是 *server*，不是 *localhost*。
- *client* 使用被动模式。

11.2.2.1 选择转发目标

我们把 *server* 当作转发的目标，而不是 *localhost*（换句话说，不用 *-L2001:localhost:21*）。先前我们建议尽可能把 *localhost* 设为转发目标。[9.2.8] 那里的技术在这儿不适用。看看那样做的结果如何：

```

client% ftp localhost 2001
Connected to client
220 client FTP server (SunOS 5.7) ready.
331 Password required for res.
Password:
230 User res logged in.
ftp> ls
200 PORT command successful.
425 Can't build data connection: Cannot assign requested address.
ftp>

```

这个问题有点儿难理解，不过跟踪一下FTP服务器响应ls命令的过程就能解释清楚了。下面是Linux *strace*命令的输出（注3）：

```

socket(2, 2, 0, "", 1) = 5
bind(5, 0x0002D614, 16, 3) = 0
      AF_INET name = 127.0.0.1 port = 20
connect(5, 0x0002D5F4, 16, 1) Err#126 EADDRNOTAVAIL
      AF_INET name = 192.168.10.1 port = 2845
write(1, "4 2 5  C a n ' t b u...", 67) = 67

```

在此，FTP服务器尝试建立一条到客户端的TCP连接，其目的地址是对的，不过源套接字却错了：这个套接字建立在回环地址127.0.0.1的ftp-data端口上。而回环地址只能与同一台机器上的其他回环地址对话。TCP协议知道这一情况，因此返回“address not available (EADDRNOTAVAIL，地址不可达)”的错误。客户端建立控制连接时以某个服务器地址为目的地址，FTP服务器小心地从同一地址上发起数据连接。在这个问题发生的环境中，控制连接已经使用SSH进行转发了，因此对FTP服务器来说，连接就像来自本地主机一样。因为我们用回环地址作为转发目标，所以其中一个方向的转发路径（从*sshd*到*ftpd*）的源地址也就是回环地址。为消除此问题，就得用server的非回环IP地址作为转发目标；这样，FTP服务器就能从那个地址发起数据连接了。

既然服务器无法发起任何连接，你可能会想到用被动模式解决问题。不过如果你试试：

```

ftp> passive
Passive mode on.

```

注3：如果你的操作系统是Solaris 2 (SunOS 5)，那么系统提供的相应命令名叫。Solaris中也有一条命令叫做，不过它与我们的问题完全无关。Solaris 1 (SunOS 4及更早) 中有命令，在BSD中则为。

```
ftp> ls
227 Entering Passive Mode (127,0,0,1,128,133)
ftp: connect: Connection refused
ftp>
```

你会发现这种情况下，失败的原因依然相同，不过表现略有不同而已。这次，server 确实在监听从 client 来的数据连接，但是同样地，它认为 client 在本地，因此它在回环地址上监听。它告诉 client 那个监听 socket 的地址（地址 127.0.0.1，端口 32901），client 试着连接进来。但是结果是 client 向 client 主机的 32901 端口发起连接，而不是 server 的！那里没有任何监听套接字，所以连接当然会被拒绝。

11.2.2.2 使用被动模式

注意，我们必须把 client 设置成被动模式。后面你将看到，被动模式总体上对 FTP 是有益的，因为它可以避免一些常见的防火墙或者 NAT 问题。不过在这里，使用被动模式是为了解决 FTP/SSH 中的特定问题；如果不这样做，下面就是后果：

```
$ ftp -d localhost 2001
Connected to localhost.
220 server FTP server (SunOS 5.7) ready.
---> USER res
331 Password required for res.
Password:
---> PASS XXXX
230 User res logged in.
ftp> ls
---> PORT 127,0,0,1,11,50
200 PORT command successful.
---> LIST
425 Can't build data connection: Connection refused.
ftp>
```

这个问题是 localhost 作为转发目标产生的问题在此的一个镜像，不过这次发生在 client 一边。client 给 server 一个套接字供其连接，因为它认为 server 是在本地主机上，所以这个套接字绑定在回环地址上。这样 server 企图连接的就是它自己的本地主机而不是 client。

我们并不能随时使用被动模式：FTP 客户端或服务器可能不支持这种模式，或者服务器端的防火墙/NAT 出于某种考虑禁止使用被动模式（马上就能看到例子）。如果是这样，可以用 SSH 的 `GatewayPorts` 特性解决这个问题，就和解决前一个问题一样：用主机的真实 IP 而不是回环地址。即：

```
client% ssh -g -L2001:server:21 server
```

然后用主机名而不是 localhost 来连接 client:

```
client% ftp client 2001
```

这样就连接到了 SSH 代理上，这个代理是建立在 client 非回环地址上的，FTP 客户端就在那个地址上监听数据连接。不过 -g 有安全方面的隐患。[9.2.1.1]

当然，如前所述，通常会发生主动模式不可用的情况。而且很可能你本机防火墙 / NAT 的设置也需要被动模式，但你还是无法使用它。如果真是那样，只能说运气不好。那就只有把数据存在磁盘里让快递帮忙了。

我们讨论的各种问题尽管很普遍，但还是与你的特定 Unix 版本和 FTP 实现方式有关。例如，一些 FTP 服务器在开始连接回环套接字之前就会出错；一看到客户端的 PORT 命令就立刻拒绝，打印一条“illegal PORT command”。不过，如果你能理解不同出错方式的原因，那么不管它表现成什么样子，你都能学着识别。

11.2.2.3 “PASV 端口窃听” 问题

在 FTP 中使用 SSH 有点像玩计算机上的城堡游戏：你会发现自己在 TCP 连接组成的曲曲折折的迷宫里，所有的连接看上去都有点儿像，但每一个却都没法解决问题。即使你遵从了到目前为止我们给你提出的每一条建议，理解并避免了我们提到的每一种危险，连接还是会失败：

```
ftp> passive
Passive mode on.
ftp> ls
connecting to 192.168.10.1:6670
Connected to 192.168.10.1 port 6670
425 Possible PASV port theft, cannot open data connection.
! Retrieve of folder listing failed
```

假如你还没决定完全放弃，另找个没这么烦人的职业，你就会想知道“现在这是为什么呢？”这里的问题源于 FTP 服务器的一个安全特性，特别是目前流行的来自华盛顿大学 (Washington University) 的 *wu-ftp*。（请参看 <http://www.wu-ftpd.org/>。此特性可能在其他的 FTP 服务器上也有，不过我们还没有看到。）这种服务器接受了从客户端来的数据连接之后，发现源地址与控制连接的源地址不同（因为控制连

接被 SSH 转发，所以源地址是服务器的主机地址）。它就做出一个有人在实施攻击的结论！FTP 服务器认定有人在监听你的 FTP 控制连接，它看到了服务器对包含监听套接字的 PASV 命令所做的响应，在合法客户端建立连接之前跳进连接里来。因此，服务器中断该连接，还报告说怀疑存在“端口窃听”（见图 11-3）。

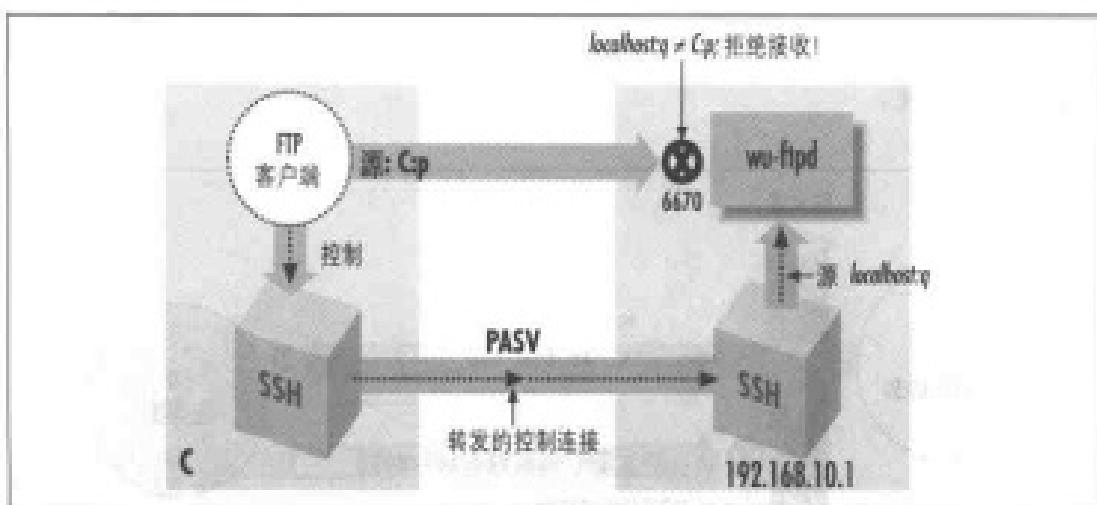


图 11-3：“PASV 端口窃听”

要解决这个问题没别的办法，只能让服务器停止检测端口。这个特性从一开始就有问题，因为它不仅阻止了攻击，也妨碍了合法的 FTP 操作。比如，被动模式的操作的设计初衷是让 FTP 客户端直接干预两台远程服务器之间的文件传输，而不是先把文件取到客户端再传给第二台服务器。这种操作不常见，但的确是 FTP 协议设计中的一部分，而 wu-ftpd 的“端口窃听”检测阻碍了这样的应用。去掉 FIGHT_PASV_PORT_RACE 选项（用 `configure --disable-pasvip` 命令）之后重新编译 wu-ftpd 就可以关掉端口检测。你也可以保留此项，而只允许特定的账号用多个 IP 进行数据传输（用 `pasv-allow` 和 `port-allow` 设置语句可以实现）。细节请参看 `ftpaccess (5)` 手册页。注意，这些是最近比较新的 wu-ftpd 功能，较早版本中没有。

更多关于如何配置 wu-ftpd 以支持被动模式的信息可以在 <http://www.wu-ftpd.org/doc/ftpaccess.html> 以及 <http://www.wu-ftpd.org/doc/ftpcap.html> 上找到。

11.2.3 FTP、防火墙和被动模式

回想一下，在主动模式中，FTP 数据连接与你想像的可能正相反——它是从服务器回到客户端的。这种平常的操作模式（如图 11-4 所示）在有防火墙存在的情况下通常会出问题。假设客户端在一个防火墙后面，这个防火墙允许所有向外的连接通过但禁止所有向内的连接。这样，客户端可以建立控制连接、登录、处理命令，但

数据传输操作（如，*ls*、*get*、*put*都会失败），因为防火墙阻隔了从服务器返回客户端主机的数据连接。简单的包过滤防火墙无法设置成让这些连接通过，因为它们看上去是从独立的TCP目标到随机的端口去的，与已经建立的FTP控制连接没有明显的关系（注4）。也许很快就返回一个“connection refused”的消息，也许连接会挂起一段时间然后失败。这取决于防火墙是用ICMP或TCP RST消息明确地拒绝一个连接，还是仅仅悄无声息地丢弃这些包。注意：无论SSH是否转发控制连接，这个问题都可能出现。

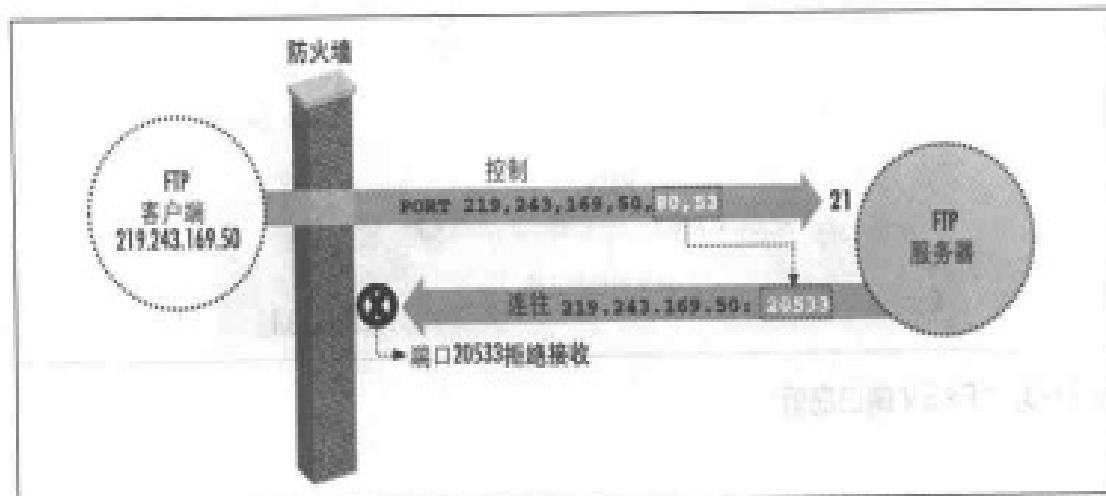


图 11-4：防火墙后的 FTP 客户端

被动模式通常可以解决这个问题，因为它把数据连接的方向反过来，变成从客户端到服务器。不过，不是所有的FTP客户端或者服务器都支持被动模式传输。命令行方式的FTP客户端通常用*passive*命令切换被动传输模式；如果哪个终端不识别该命令，可能就不支持被动模式了。如果客户端支持而服务器不支持，那么服务器会发来这样的消息，“PASV: command not understood”。PASV 是 FTP 协议通知服务

注4：更高级的防火墙可以解决此问题。这类产品是应用层代理和包过滤器的混合体，通常称为透明代理（transparent proxy），或有状态的包过滤器（stateful packet filter）。这类防火墙能理解FTP协议，并保持FTP控制连接。当它看到FTP客户端发出PORT命令时，就在防火墙上动态打开一个临时的口号，让特定的FTP数据连接回来时通过。这个口号很短时间之后就会自动消失，而且仅能在发出PORT命令的套接字和FTP服务器的ftp-data套接字之间存在。这种产品通常也能进行NAT操作，能使我们下面将要提到的FTP/NAT问题透明化。

器监听数据连接的命令。最终就算被动模式解决了防火墙的问题，对SSH转发仍然无济于事，因为（数据连接）要用的端口仍然是动态选择的。

这里是防火墙阻隔反向数据连接的例子：

```
$ ftp lasciate.ogni.speranza.org
Connected to lasciate.ogni.speranza.org
220 ProFTPD 1.2.0pre6 Server (Lasciate FTP Server)
[lasciate.ogni.speranza.org]
331 Password required for slade.
Password:
230 User slade logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful.
[……等较长的一段时间……]
425 Can't build data connection: Connection timed out
```

这时用被动模式就行了：

```
ftp> passive
Passive mode on.
ftp> ls
227 Entering Passive Mode (10,25,15,1,12,65)
150 Opening ASCII mode data connection for file list
drwxr-x--x 21 slade web 2048 May 8 23:29 .
drwxr-xr-x 111 root wheel 10240 Apr 26 00:09 ..
-rw----- 1 slade other 106 May 8 15:22 .cshrc
-rw----- 1 slade other 31384 Aug 18 1997 .emacs
226 Transfer complete.
ftp>
```

在讨论如何穿过防火墙使用FTP时，我们根本没有提到SSH；这是FTP协议和防火墙固有的问题。然而，就算是已经通过SSH转发了FTP控制连接，问题仍然会存在，因为困难实际上 是数据连接，而不是控制连接，数据连接并没有经过SSH转发。所以，这是通常得在FTP和SSH中用被动模式的另一个原因。

11.2.4 FTP 和 NAT

被动模式还可以解决另一个常见的FTP问题：与网络地址转换(NAT)的矛盾。NAT是一种用网关连接两段网络的手段，在数据包经过时修改其源与目的地址。用NAT的一个好处是，可以将网络接入Internet或者更换ISP，而不需要给整个网络重新编址（指更换所有的IP地址）。它也可以让网络上大量使用不可路由的私有地址的机

器分享有限的可路由 Internet 地址。人们经常称 NAT 的这个好处为伪装 (masquerade)。

假设 FTP 客户端所在的机器有一个私有地址，它只在本地网络中可用，你通过一台 NAT 网关接入 Internet。客户端可以创建到外部 FTP 服务器的控制连接。然而，如果客户端尝试用通常的逆向方式创建数据连接，就会有问题出现。客户端不知道存在 NAT 网关，因此它（通过 PORT 命令）告诉服务器连接包含它的私有地址的套接字。在远程主机那儿这个地址是没有用的，所以服务器通常会发一个“no route to host”响应，然后断开连接（注 5）。图 11-5 对这种情况做了解释。用被动模式同样能解决，因为这样做服务器不必连回客户端，客户端的地址也就无关紧要了。

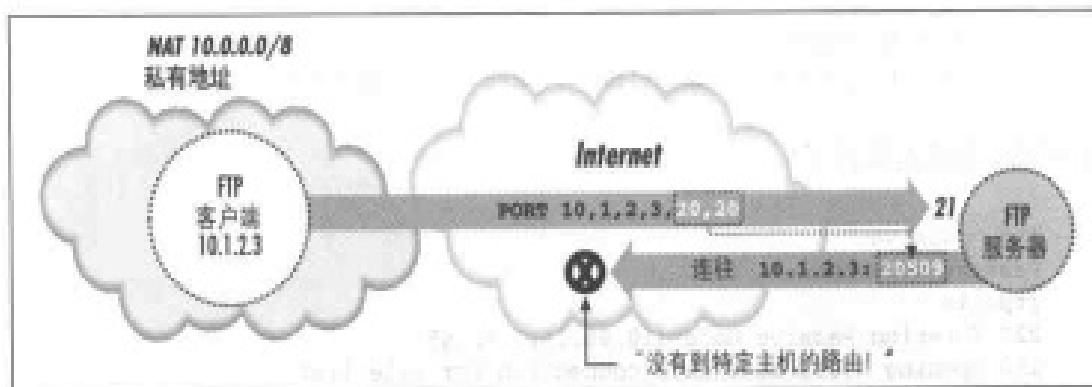


图 11-5：客户端 NAT 阻碍了主动模式的 FTP 传输

到此为止，我们已经列举了三种需要被动 FTP 的情况：控制连接转发、客户端位于防火墙之后以及客户端位于 NAT 之后。既然主动 FTP 有这些潜在的问题，而且据我们目前已知的情况看，被动 FTP 没有什么不利因素，因此，我们推荐只要可能，就一直使用被动 FTP。注 5：也可能出现更糟糕的情况。服务器可能本身也有端址机制，如果运气实在太差，客户端的私有地址可能恰好是服务器端某台完全不同的机器的地址。不过服务器端的那台机器不太可能碰巧也在你的 FTP 客户端随机选择的端口上监听。所以，结果也许只会产生一个“connection refused”的错误。

11.2.4.1 服务器端的 NAT 问题

刚才讨论了客户端的 NAT 问题。如果 FTP 服务器在 NAT 网关之后，而且你又用 SSH 转发控制连接的话，问题会更复杂。

注 5：也可能出现更糟糕的情况。服务器可能本身也有端址机制，如果运气实在太差，客户端的私有地址可能恰好是服务器端某台完全不同的机器的地址。不过服务器端的那台机器不太可能碰巧也在你的 FTP 客户端随机选择的端口上监听。所以，结果也许只会产生一个“connection refused”的错误。

首先，让我们看图理解一下没有 SSH 时的基本问题。如果服务器在 NAT 网关之后，你遇到的是我们以前讨论过的镜像问题。在前面的问题中，因为客户端把非NAT解析的内部地址用PORT命令传给服务器，而这个地址实际上不可达，所以主动模式无法工作。现在的情况是，被动模式不能工作，因为服务器在响应PASV命令时把它的内部地址传给客户端，而这个地址对客户端来说是不可达的（见图 11-6）。

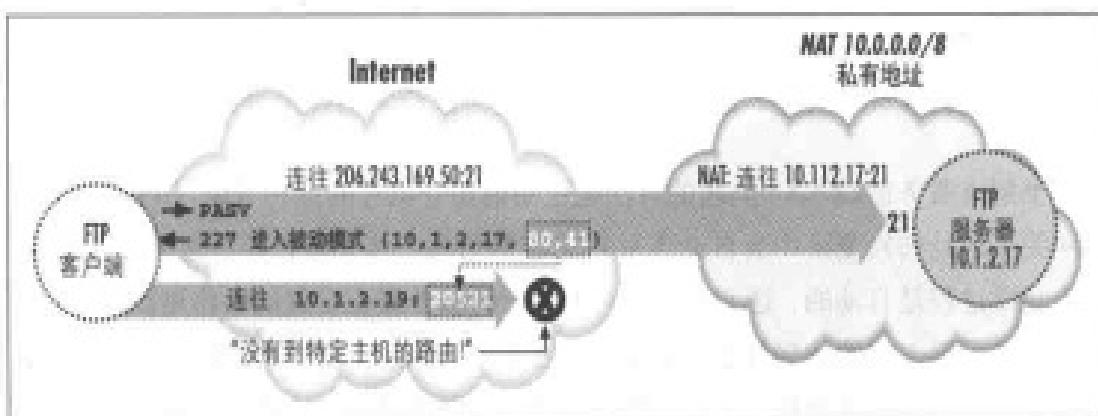


图 11-6：服务器端 NAT 阻碍了被动模式的 FTP 传输

以前说要用被动模式，现在，最简单的回答是反过来：用主动模式。不过这个回答也没太大用处。因为如果一台服务器想给整个网络服务，那它就应该给最多的人提供有用的东西。由于客户端NAT和防火墙的设置问题，普遍需要的是被动模式FTP，因此就不应该再使用服务器端的NAT设置，它需要使用主动模式；否则还是无法访问服务器。我们的办法是采用那些专门为解决这个问题设计的FTP服务器。前面我们介绍的 *wu-ftpd* 服务器就具有此功能。下面的话引自 *ftpaccess (5)* 手册页：

```
passive address <externalip> <cidr>
    允许改变 PASV 命令的响应中报告的地址。当任何其地址能
    与 <cidr> 匹配的控制连接请求被动模式 (PASV) 数据连
    接时，就报告 <externalip> (外部 IP) 地址。注意：这
    并不改变守护进程实际监听的地址，只改变了报告给客户
    端的地址。此特性使得守护进程在位于 IP 重叠的防火墙后仍
    能正确工作。
```

例如：

```
passive address 10.0.1.15 10.0.0.0/8
passive address 192.168.1.5 0.0.0.0/0
```

从 A 类地址 10 来的客户连接都被告知被动连接监听建立在 10.0.1.15 上，
而对所有其他的连接，则告知监听地址为 192.168.1.5。

还可以设置多个这样的被动地址，以处理复杂的或多网关的网络。

这种方法很灵活，除非你碰巧在用 SSH 做控制连接转发。网站管理员可以对 FTP 服务器进行设置，使所有服务器私有网络之外来的控制连接请求都向其 PASV 命令报告外部地址。但是经过转发的控制连接看起来是从服务器主机本身发出的，而并非外部网络。从私有网络中来的控制连接应该得到内部地址，而不是外部地址。解决这个问题的唯一办法是：如果连接来自外部或是服务器自身，则给控制连接返回外部地址。这是切实可行的，因为很少有可能要把数据通过 FTP 传回这台机器本身。在有服务器端 NAT 的情况下，你可以运用此技术，在存在服务器端 NAT 的情况下让控制连接转发成为可能，遇到同样问题时还可以给网站管理员提供建议。

另一类解决服务器端 NAT 问题的方法是使用前面介绍过的那种智能 NAT 网关。这种网关在地址转换变换中会自动重写 FTP 控制信息。从某些方面看这很有吸引力，因为转换过程是自动的、透明的；在网关后面的服务器上设置的工作更少了，服务器和网络设置之间的依赖性也降低了。不过实际上，这种方法对于我们的目的而言比在服务器上设置更糟糕。因为它依赖于网关识别并替换 FTP 控制连接的能力。而这正是 SSH 力图避免的操作！如果控制连接通过 SSH 转发，网关根本不知道那是控制连接，因为它只是嵌入在 SSH 会话内的一个通道。控制连接本身不是一个单独的 TCP 连接；它在 SSH 端口上，而在不在 FTP 端口上。网关无法读取，因为数据是加密的；而且即便网关能读取数据，也不能修改，因为 SSH 有一致性保护。如果你碰到这种情况——客户端必须用被动模式的 FTP，服务器又在一个会改写 FTP 控制数据的网关之后，你就必须说服服务器管理员，让他除了在网关上设置以外，再采取一些服务器级的技术，特别是要允许转发。否则，问题是没法解决的，你将来就只能用卡车运磁带了。哦，可能还能用基于 SSL 的 HTTP PUT 命令。

我们已经讨论了如何转发 FTP 控制连接，如何保护你的登录名、口令和 FTP 命令。如果你想要的就是这些，那么这个例子就算学完了。不过我们还要继续深入探索漆黑一片的数据连接的秘密。你需要有一些技术背景，因为我们将涉及详尽的细节和很少有人知道的 FTP 模式。（你可能觉得奇怪，是不是我们不小心把一部分 FTP 的内容插进这本 SSH 里来了）。前进吧，勇敢的读者！

11.2.5 数据连接问题

问问 SSH 用户怎么样转发 FTP 数据连接，他们多半会这样回答，“这不可能。”然而恰恰相反，这是可能的。我们找到的方法既难懂，又不方便，而且通常抵不上为它

付出的努力，但的确可以实现。在向你解释这种方法之前，必须首先讨论一下FTP在客户端和服务器之间传输文件的三种主要方法：

- 普通方法。
- 被动模式传输。
- 使用缺省数据端口传输。

对前两种我们只会简单介绍一下，因为前面已经讨论过了；我们可能会扩充一点儿更具体的东西。然后我们要讨论第三种方法，这种方法知道的人最少，但是如果你真的很想转发FTP数据连接，就非它莫属了。

11.2.5.1 普通的文件传输方法

多数FTP客户端都会用下面的方式传输数据：建立控制连接，认证，然后用户发出传送文件的命令。假设命令是get *fichier.txt*，表示请求服务器把文件*fichier.txt*传到客户端来。这个命令的响应过程如下：客户端先选择一个空闲的本地TCP套接字，比方说叫C，在其上开始监听；然后，客户端向服务器发出PORT命令，指明套接字C，服务器确认之后，客户端再发出RETR *fichier.txt*命令，这就是通知服务器连接到先前指定的套接字（C），在新的数据连接上传输文件内容；最后，客户端接受数据连接，读取数据，写入本地*fichier.txt*文件。传输结束后，数据连接关闭。下面记录的就是这样一次会话过程：

```
$ ftp -d aaor.lionaka.net
Connected to aaor.lionaka.net.
220 aaor.lionaka.net FTP server (SunOS 5.7) ready.
---> USER res
331 Password required for res.
Password:
---> PASS XXXX
230 User res logged in.
---> SYST
215 UNIX Type: L8 Version: SUNOS
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get fichier.txt
local: fichier.txt remote: fichier.txt
---> TYPE I
200 Type set to I.
---> PORT 219,243,169,50,9,226
200 PORT command successful.
```

```
---> RETR fichier.txt
150 Binary data connection for fichier.txt (219.243.169.50,2530) (10876
bytes).
226 Binary Transfer complete.
10876 bytes received in 0.013 seconds (7.9e+02 Kbytes/s)
ftp> quit
```

注意PORT命令，PORT 219,243,169,50,9,226。这表示客户端监听219.243.169.50的2530 (= (9<<8)+226) 端口；此列表由逗号分开，最后两个整数是用两个8位字节表示的16位端口号，高字节在前。服务器的响应以150开始，表示确认建立到该套接字的数据连接。数据连接的源端口总建立在标准FTP数据端口20上（还记得FTP服务器在21端口上监听到达的控制连接吗），这一点没有明确表示。

在这个过程中，要注意两个重要的问题：

- 数据连接套接字由客户端动态选择。这对转发构成障碍，因为用SSH转发的端口是无法预见的。用telnet手工创建FTP进程可以解决这个问题。也就是说，预先选好数据套接字，用SSH将它从telnet转接到FTP，手工发出所有FTP控制命令，在PORT命令中用已转接的端口号。但这简直太不方便了。
- 数据连接的方向与控制连接相反；从服务器回到客户端。在本章开头，我们说通常被动模式是比较常用的。

11.2.5.2 深入理解被动模式

回想一下，在被动模式传输中，客户端发起至服务器的连接。特别需要指出的是，客户端不是在本地套接字上启动监听之后向服务器发PORT命令，而是发出PASV命令。服务器在它那边选一个监听套接字，并将其作为PASV命令的响应告知客户端。然后，客户端连接此套接字，形成数据连接，同时在控制连接上发送文件传输命令。在命令行方式的客户端中，启动被动模式的常用方法为passive命令。再举个例子：

```
$ ftp -d aaor.lionaka.net
Connected to aaor.lionaka.net.
220 aaor.lionaka.net FTP server (SunOS 5.7) ready.
---> USER res
331 Password required for res.
Password:
---> PASS XXXX
230 User res logged in.
---> SYST
215 UNIX Type: L8 Version: SUNOS
```

```
Remote system type is UNIX.  
Using binary mode to transfer files.  
ftp> passive  
Passive mode on.  
ftp> ls  
---> PASV  
227 Entering Passive Mode (219,243,169,52,128,73)  
---> LIST  
150 ASCII data connection for /bin/ls (219.243.169.50,2538) (0 bytes).  
total 360075  
drwxr-xr-x98 res 500 7168 May 5 17:13 .  
dr-xr-xr-x 2 root root 2 May 5 01:47 ..  
-rw-rw-r-- 1 res 500 596 Apr 25 1999 .FVWM2-errors  
-rw----- 1 res 500 332 Mar 24 01:36 .ICEauthority  
-rw----- 1 res 500 50 May 5 01:45 .Xauthority  
-rw-r--r-- 1 res 500 1511 Apr 11 00:08 .Xdefaults  
226 ASCII Transfer complete.  
ftp> quit  
---> QUIT  
221 Goodbye.
```

注意，用户发出 `ls` 命令之后，客户端传输的是 `PASV` 而不是 `PORT`。服务器的响应中包含了它要建立监听的那个套接字。客户端发出 `LIST` 命令，要求列出远程目录当前的内容，同时连接到远程的数据套接字；服务器接受并确认连接，在这个新的连接上传输目录列表。

以前曾提到一件非常有意思的事情，就是 `PASV` 命令原本不是这样用的；设计它的目的是让一个FTP客户端指挥两台远程服务器进行文件传输。客户端分别创建到两台远程服务器的控制连接，向一个发出 `PASV` 命令，令其启动一个监听套接字，再向另一个发出 `PORT` 命令，令其连接前一台服务器的那个监听套接字，然后发出数据传输命令（`STOR`、`RETR` 等等）。现在大多数人根本不知道还能这样做，通常只会先把文件从一个服务器上下载到本地，然后再上传到另一台远程机器。这种直接传输的方式太少见了，很多FTP客户端甚至都不支持，还有一些服务器也出于安全的考虑禁用此项功能。[\[11.2.2.3\]](#)

11.2.5.3 缺省数据端口 FTP

在第三种文件传输模式中，客户端既不发出 `PORT` 命令也不发出 `PASV` 命令。在这种情况下，服务器发起数据连接，从已知的 `ftp-data` 端口（20）连接到控制连接的源套接字，因为在那个套接字上一定有客户端的监听（对FTP会话而言，这些套接字叫做“缺省数据端口，default data ports”）。通常使用这种模式要用到FTP客户

端的*sendport*命令，该命令是客户端的状态转换开关，确定其在每一次数据传输中是否发送PORT命令。缺省状态通常是开（发送），为了用这里提到的传输模式，此开关将关闭（不发送）。操作步骤如下：

1. 客户端从本地套接字C发起控制连接，连往服务器的21端口。
2. 依次发出*sendport*命令，数据传输命令，如，*put*或*ls*。FTP客户端开始监听到达C的TCP连接。
3. 服务器确定数据连接的目的套接字是控制连接另一方的C。这一点不需要客户端专门用FTP协议告知服务器，用TCP协议就能知道（例如调用socket API例程*getpeername()*）。然后，服务器建立从ftp-data端口到C的连接，开始在此连接上发送或接收客户端请求的数据。

这样做当然比每个数据传输用不同的套接字要简单，所以得问问为什么通常情况下要传输PORT命令。如果你试试看，就会发现原因了。首先设为off，客户端可能会出错，错误消息是“bind: Address already in use”。就算不出错，也只能做一次数据传输操作，第二个*ls*命令同样会引起地址错误，这次错误来自服务器：

```
aaor% ftp syrinx.lionaka.net
Connected to syrinx.lionaka.net.
220 syrinx.lionaka.net FTP server (Version wu-2.5.0(1) Tue Sep 21
16:48:12 EDT
331 Password required for res.
Password:
230 User res logged in.
ftp> sendport
Use of PORT cmd off.
ftp> ls
150 Opening ASCII mode data connection for file list.
keep
fichier.txt
226 Transfer complete.
19 bytes received in 0.017 seconds (1.07 Kbytes/s)
ftp> ls
425 Can't build data connection: Cannot assign requested address.
ftp> quit
```

这些错误是由TCP协议的技术细节引起的。在本例中，每个数据连接都建立在两个相同的套接字之间：服务器的ftp-data和C。TCP连接完全靠源/目的套接字对区分，因此TCP无法分辨这些数据连接；它们是同一连接的不同应用，不能同时存在。实际上，每个应用关闭之后有一段等待时间，以保证属于不同应用的数据包不会混在

一起，在此期间不能建立新的应用。用 TCP 术语来说，连接的一端执行“主动关闭（active close）”后，停留在 TIME_WAIT 状态。处于这种状态的时间大约是网络数据包最长生命期的两倍。（或者叫“2MSL”，表示 Maximum Segment Lifetime 的两倍）。此后，连接完全关闭，新的应用可以开始。实际的超时值随系统不同而不同，通常是 30 秒到 4 分钟（注 6）。

实际上一些 TCP 限制得比这还要强。如果某个端口上的套接字正处于 TIME_WAIT 状态，那么这个端口通常也不可用，甚至不能连接另一个远程套接字。我们还曾碰到一些系统，不管连接处于什么状态，都会禁止连接终点上的套接字继续监听（其他连接）。这些限制不是 TCP 规定的，但是很常见。这样的系统通常会提供禁用这些限制的手段，比如 Berkeley 套接字 API 的 SO_REUSEADDR 选项。当然，FTP 客户端一般会使用这种功能，但可并不一定管用。

缺省端口 FTP 传输过程中有两个地方存在地址重用问题。第一，当客户端在缺省数据端口启动监听时，这个端口肯定已经是控制连接在本机使用的接口了。即便应用程序真的需要地址重用，一些系统也只是简单地将其禁用；这也正是连接立即失败的原因，“地址已使用（address already in use）”

另一个地方是第二个数据传输。当第一个传输结束后，服务器就关闭了数据连接，服务器端的连接进入 TIME_WAIT 状态。如果在 2MSL 时间没到的时候请求建立新数据连接，服务器会试图在同一个连接上建立另外的应用，当然就会出现这样的错误：“无法分配请求的地址（cannot assign requested address）”。不论地址重用设置如何，都会发生这样的情况，这是 TCP 的规则所致。当然了，过几分钟你就能接着传文件了，不过多数用计算机的人都等不了几秒钟，更别说几分钟了。这就是每次传输必须使用 PORT 命令的原因；这样每次连接的端口都不同，就不会出现 TIME_WAIT 冲突了。

由于以上问题，通常不使用缺省端口传输模式。然而这种方式对我们有个很重要的好处：如果数据连接的目的端口是固定的，而且在传输命令发出前就可知，这是唯一的方式。知道了这一点，再加上一些耐心和足够的运气，没准儿你还真能在 SSH 上转发 FTP 数据连接呢。

注 6：更多有关 TIME_WAIT 状态的技术信息，请参看《TCP/IP Illustrated, Volume 1: The Protocols》，W. Richard Stevens 著，Addison-Wesley 出版。

11.2.6 转发数据连接

介绍了前面这些内容之后，现在我们简要介绍一下建立数据连接转发的步骤。关键是 SSH 必须向 TCP 协议请求地址重用，才能转发数据连接。SSH2 与 OpenSSH 已经支持地址重用，SSH1 不行。不过修改 SSH1 的源码也很容易。在 *newchannels.c* 的 *channel_request_local_forwarding* 中，调用 *bind()* 之前加入下面的代码（在 1.2.27 版中是 1438 行）：

```

...
sin.sin_port = htons(port);

{
    int flag = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (void *)&flag,
                sizeof(flag));
}

/* 将套接字绑定到地址 */
if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    packet_disconnect("bind: %s", strerror(errno));
...

```

然后在服务器上重编译安装 *sshd*。如果你不具备进行这些操作的权限，还可以把改好的 *ssh* 客户端程序拷贝到服务器上，然后在下面第（3）步中，从服务器向客户端运行 *ssh -L*，用以代替从客户端向服务器运行的 *ssh -R*。

另一个限制是：FTP 客户端所在机器的操作系统必须允许进程在已建立连接的套接字上继续监听。一些操作系统是不允许这样的。测试的方法是：不通过 SSH，在缺省数据端口上传输 FTP 数据，使用 FTP 的方式与平常一样，只要在 *ls*、*get* 等数据传输命令之前执行 *sendport*。如果返回：

```
ftp: bind: Address already in use
```

那么你的操作系统可能就不支持这种方式。此时必须查看操作系统手册页，看是否有改变的办法。图 11-7 描述了转发的具体步骤：

1. 如本章前面所述，启动 SSH 连接，转发控制通道，用 FTP 客户端连接服务器。确认被动模式已经关闭。

```
client% ssh1 -f -n -L2001:localhost:21 server sleep 10000 &
```

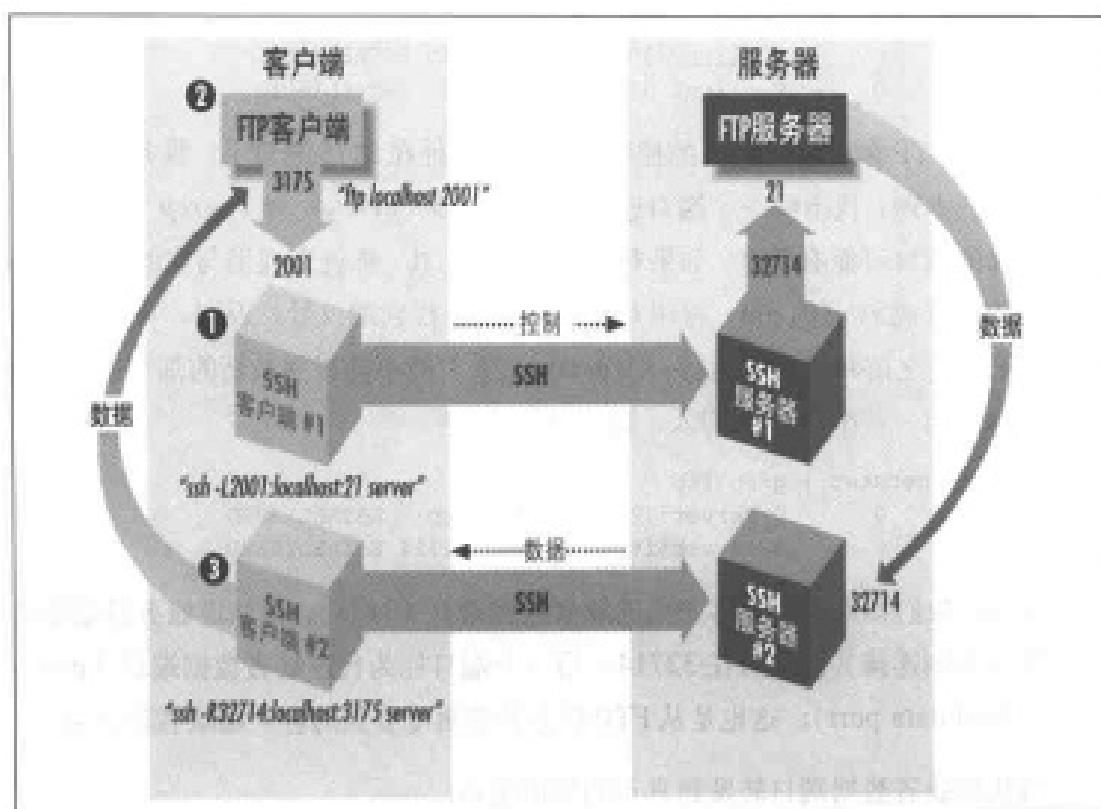


图 11-7：转发 FTP 数据连接

或者，在 SSH2 中：

```
client% ssh2 -f -n -L2001:localhost:21 server
```

然后：

```
client% ftp localhost 2001
Connected to localhost.
220 server: FTP server (SunOS 5.7) ready.
Password:
230 User res logged in.
ftp> sendport
Use of PORT cmd's off.
ftp> passive
Passive mode on.
ftp> passive
被动模式第一次再尝试时，本地时间12M1S的差异。
Passive mode off.
```

请注意这里的转发目标是 localhost，和前面建议的不一样。这没关系，因为根本没有 PORT 或 PASV 命令，当然地址就不会出错了。

2. 现在，我们要决定 FTP 客户端的缺省数据端口，这包括真正传输数据的端口和转发到的端口。在客户端运行 netstat 命令：

```
client% netstat -n | grep 2001
tcp      0      0 client:2001 client:3175 ESTABLISHED
tcp      0      0 client:3175 client:2001 ESTABLISHED
```

这表示FTP客户端到SSH的控制连接的源地址在3175端口上。服务器端也可以照此办理，找出哪一个端口连入FTP服务器（*netstat -n | egrep '^<21>'*），这样的端口可能有很多。如果有*lsof*之类的工具，那就先找出与你的客户端相关的*ftpd*或*sshd*的pid，再用*lsof -p <pid>*找到端口号。否则，可以在建立FTP连接之前和之后各运行一次*netstat*，看看能不能找到最新的那个连接。假设你是惟一一个用FTP的人，就这样做：

```
server% netstat | grep ftp
tcp      0      0 server:32714 server:ftp    ESTABLISHED
tcp      0      0 server:ftp     server:32714 ESTABLISHED
```

现在，我们知道了FTP客户端的缺省数据端口（3175），也知道服务器端被转发的控制连接其源端口在32714，后一个端口称为代理缺省数据端口（proxy default data port）；这也是从FTP服务器的角度看到的客户端缺省数据端口。

3. 将代理缺省数据端口转发到真正的数据端口：

```
# SSH1, OpenSSH
client% ssh1 -f -n -R32714:localhost:3175 server sleep 10000 &

# 仅对SSH2
client% ssh2 -f -R32714:localhost:3175 server
```

根据前面说过的，如果没有替换*sshd*，或者又运行了第二个，就得在服务器上从反方向运行修改过的*ssh*，如下所示：

```
server% ./ssh -f -n -L32714:localhost:3175 client sleep 10000 &
```

4. 用*ftp*执行数据传输命令。如果一切正常，那也只能正常一次，然后FTP服务器返回出错消息：

```
425 Can't build data connection: Address already in use.
```

（有些FTP立即返回错误信息，有些则会重试几次，因此产生延迟。）如果等到服务器的2MSL时间结束，你就能再做一次数据传输。用*netstat*可以看到问题出在哪儿，还能跟踪整个过程：

```
server% netstat | grep 32714
127.0.0.1.32714    127.0.0.1.21        32768      0 32768      0 ESTABLISHED
127.0.0.1.21       127.0.0.1.32714    32768      0 32768      0 ESTABLISHED
127.0.0.1.20       127.0.0.1.32714    32768      0 32768      0 TIME_WAIT
```

前两行表示在21端口上建立控制连接；第三行表示20端口上有旧的数据连接，现在处于TIME_WAIT状态。这个状态消失之后，就可以执行新的数据传输命令了。

现在你知道了：你已经用 SSH 实现了数据连接转发。在 SSH 与 FTP 结合的问题上，你已经达到了最高峰，不过 Sir Gawain 说：“这只是个例子而已。”，我们同意，也许你也同意吧。如果你还是非常担心你的数据连接，而且没有别的办法传送文件，也还可以忍受在每次传文件之间等上几分钟，而且运气又很好，那它就管用了。这在黑客聚会时也是很妙的把戏呢。

11.3 Pine、IMAP 和 SSH

Pine 是 Unix 上很流行的邮件程序，源于华盛顿大学（University of Washington, <http://www.washington.edu/pine/>）。除了实现本地文件以邮件方式存储和传送外，Pine 也支持用于访问远程邮箱的 IMAP（注 7）和用于发送信件的 SMTP（注 8）。

在本例中，我们将结合 Pine 和 SSH 解决两个常见问题：

IMAP 认证

在很多情况下，IMAP 允许密码在网络上以明文形式传递。我们将讨论如何用 SSH 保护你的密码，不过不是用端口转发（很奇怪吧）。

受限邮件转递

许多 ISP 只允许他们的客户访问邮件和新闻服务器。在一些场合，你可能无法合法地把你的邮件从 ISP 那里转递出来。这回 SSH 又来救急了。

我们还要谈到，把 ssh 封装在脚本里，可以避免 Pine 连接延迟，并有利于访问多个邮箱。这次讨论 Pine/SSH 集成的问题将比前一次[4.5.4] 深入很多。

11.3.1 保护 IMAP 认证

IMAP 是客户 / 服务器协议，这点与 SSH 相同。Email 程序（如，Pine）是客户端，IMAP 服务进程（如，imapd）运行在远程计算机上，称为 IMAP 主机（IMAP host），对你的远程邮箱进行访问控制。通常 IMAP 要求你在访问邮箱之间认证，典型的方法是通过用户名和密码。

注 7： Internet 消息访问协议，RFC-2060。

注 8： 简单邮件传输协议，RFC-821。

式是输入密码，这也与SSH相同。不过多数情况下这个密码以明文传至IMAP主机，这意味着安全将受到威胁（图11-8）（注9）。

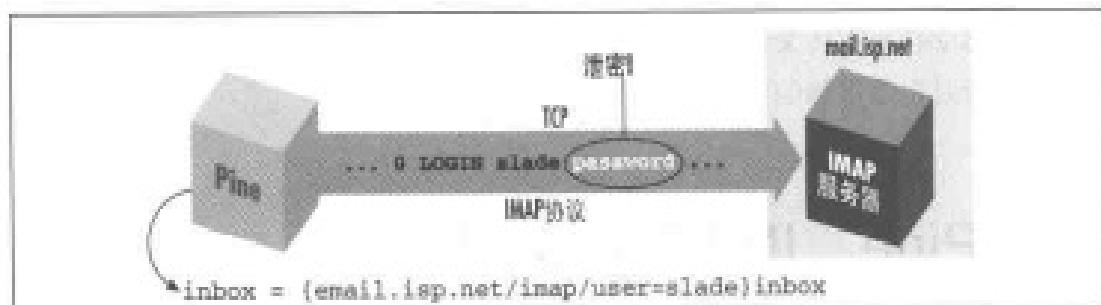


图11-8：普通IMAP连接

如果你有IMAP主机上的账号，并且这台主机上运行了SSH服务，那么你就可以对密码进行保护。因为IMAP是TCP/IP族协议，所以一种方法是在运行Pine的机器和IMAP主机之间建立SSH端口转发（见图11-9）。[9.2.1]

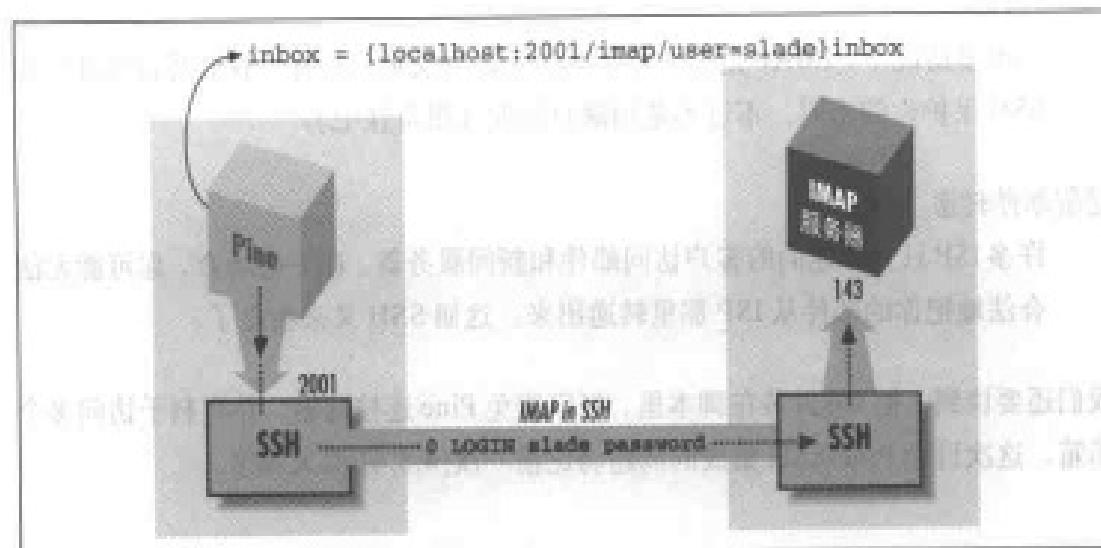


图11-9：转发IMAP连接

不过，这种技术有两个缺点：

安全隐患：从防火墙或路由器到你的IMAP服务器，所有通过转发的连接都存在安全隐患。

在多用户计算机上，其他用户不论是谁都可以连接你的转发端口。[9.2.4.3] 如果转发只是为了保护密码，那也不是什么大问题，因为大不了入侵者会建立另

注9：IMAP协议的确支持更多的安全认证方法，不过这些方法并没有得到广泛应用。

一条独立的IMAP连接，和你的连接也没什么关系。另一方面，如果端口转发能让你绕过IMAP的防火墙，那入侵者也可以用你的转发端口绕过防火墙，这个安全隐患要严重得多。

操作不便

在这种设置之下，你必须经过两次认证：第一次向IMAP主机上的SSH服务器认证（连接并创建隧道），然后用密码向IMAP认证（访问邮箱）。这既多余又烦人。

幸好我们能克服这些缺点，让你安全方便地在SSH上运行Pine。

11.3.1.1 Pine 和预认证 IMAP

IMAP协议定义了启动IMAP服务器的两种模式：普通模式和预认证模式（见图11-10）。普通模式下，服务器给每个用户邮箱都设置了特定的权限，因此需要客户端发来认证信息。如果用root启动Unix上的IMAP服务器，就进入了这种模式。

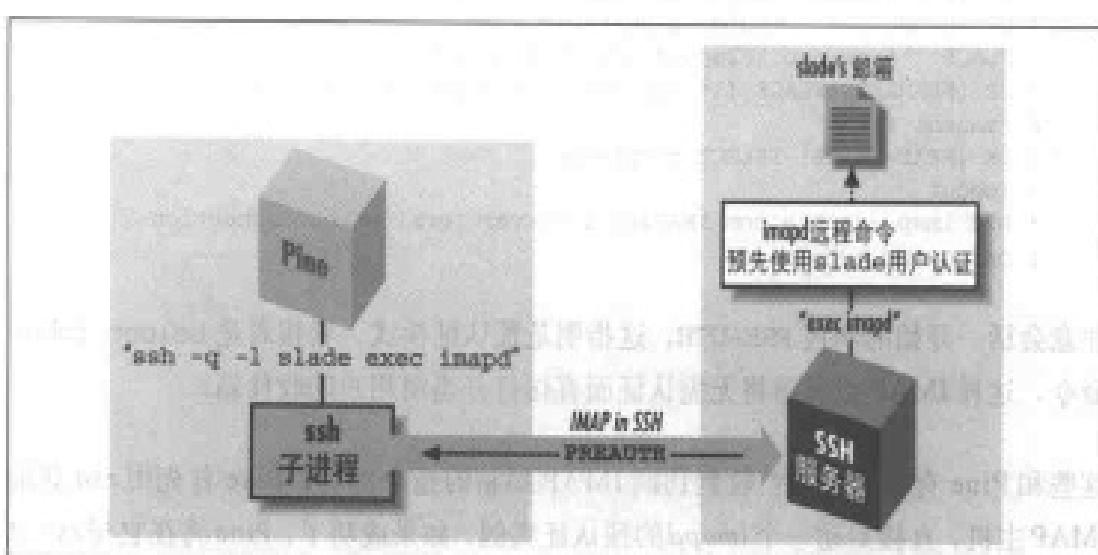


图11-10：基于SSH的预认证Pine/IMAP

下面的会话是通过inetd启动IMAP服务器imapd的例子。

```
server% telnet localhost imap
* OK localhost IMAP4rev1 v12.261 server ready
0 login res password
1 select inbox
* 3 EXISTS
```

```

* 0 RECENT
* OK [UIDVALIDITY 964209649] UID validity status
* OK [UIDNEXT 4] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)]
Permanent flags
1 OK [READ-WRITE] SELECT completed
2 logout
* BYE imap.example.com IMAP4rev1 server terminating connection
2 OK LOGOUT completed

```

在预认证模式中，IMAP服务器假定认证工作已经由启动服务器的程序完成了，因此该用户已经具有了访问其邮箱必需的权限。如果用非root用户启动*imapd*, *imapd*就认为你已经通过认证，可以打开收件箱了。然后你可以直接输入IMAP命令访问邮箱，不需要再认证：

```

server% /usr/local/sbin/imapd
* PREAUTH imap.example.com IMAP4rev1 v12.261 server ready
0 select inbox
* 3 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 964209649] UID validity status
* OK [UIDNEXT 4] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)]
Permanent flags
0 OK [READ-WRITE] SELECT completed
1 logout
* BYE imap.example.com IMAP4rev1 server terminating connection
1 OK LOGOUT completed

```

注意会话一开始的响应PREAUTH，这指明是预认证模式。紧接着是select inbox命令，这样IMAP服务器将无需认证而直接打开当前用户的收件箱。

这些和Pine有什么关系？收到访问IMAP邮箱的指令之后，Pine首先用*rsh*登录IMAP主机，直接启动一个*imapd*的预认证实例。如果成功了，Pine将在它与*rsh*之间的管道上与IMAP服务器展开对话，并自动获得访问用户远程邮箱的权限，不需要进一步认证。这是个好办法，也非常方便，惟一的问题是：*rsh*很不安全。不过可以让Pine使用SSH来代替*rsh*。

11.3.1.2 Pine与SSH集成

Pine的*rsh*登录功能受`~/.pinerc`文件中三个设置参数的控制：`rsh-path`、`rsh-command`和`rsh-open-timeout`。`rsh-path`存储打开Unix远程shell连接的程序名。

通常是 *rsh* 可执行文件的全路径（比如，*/usr/ucb/rsh*）。要让 Pine 能使用 SSH，就得将这个变量设置成 *ssh* 的客户端，而不是 *rsh*：

```
rsh-path=/usr/local/bin/ssh
```

rsh-command 表示打开远程 shell 连接的 Unix 命令行：在本例中是到 IMAP 主机的 IMAP 连接。它的值是 printf 格式的字符串，包含四个“%s”转换字符，运行时自动填充值。这四个值从前至后分别代表：

1. *rsh-path* 的值。
2. 远程主机名。
3. 访问远程邮箱的用户名。
4. 连接方式；在本例中是“imap”。

例如，*rsh-command* 的缺省值如下：

```
%s %s -l %s exec /etc/r%sd"
```

比方说这代表：

```
/usr/ucb/rsh imap.example.com -l smith exec /etc/rimapd
```

欲使之与 *ssh* 协调工作，需稍微改动原来的值，在中间加入 *-q*，启动安静模式：

```
rsh-command="%s %s -q -l %s exec /etc/r%sd"
```

这样就是：

```
/usr/local/bin/ssh imap.example.com -w -l smith exec /etc/rimapd
```

-q 是必需的，这样 *ssh* 就不发出会在 Pine 中引起混乱的提示了，比如：

```
Warning: Kerberos authentication disabled in SUID client.  
fwd connect from localhost to local port sshdfwd-2001
```

如不用 *-q*，Pine 会把这些解释成 IMAP 协议的一部分。IMAP 服务器缺省路径 */etc/r%sd* 现在是 */etc/rimapd*。

第三个变量 rsh-open-timeout 设置 Pine 打开远程 shell 连接的时间限制（秒）。缺省值 15 不必修改，不过该值可以是任何大于等于 5 的整数。

最终，Pine 的设置如下：

```
rsh-path=/usr/local/bin/ssh  
rsh-command="%s %s -q -l %s exec /etc/r%sd"  
rsh-open-timeout=
```

Pine 中的远程用户名

顺便说一句，Pine 的手册页或者配置文件注释中没有提到如何修改用户名，如果你需要用其他用户名连接远程邮箱，可用下面的语法格式：

```
{hostname/user=jane}mailbox
```

这样 Pine 调用 rsh-command 时就会用“jane”作为远程用户名（也就是说，替换第三个 %s）。

通常你用 SSH 认证时不想输密码或口令，比如用可信主机、公钥代理等。SSH 在 Pine 幕后工作，不会在终端上发出任何需要确认的东西。如果运行的是 X Window 系统，ssh 可能会弹出一个 X 界面（ssh-askpass）让你输入信息，不过你可能也不想要它。Pine 在收信过程中可能会创建多个单独的 IMAP 连接，即使在同一台服务器上也一样。这只是 IMAP 协议的工作方式而已。

设置好 `~/.pinerc` 文件，加上正确的 SSH 认证方式，就可以在 SSH 上运行 Pine 了。启动 Pine，打开远程邮箱；如果一切顺利，就不会要你输入密码。

11.3.2 邮件转递和阅读新闻

Pine 用 IMAP 读邮件，但不能发送。要想发送邮件，可以调用本机程序（比如 `sendmail`）或者用 SMTP 服务器。Pine 也可以当成新闻阅读器来用，可通过 NNTP（网络新闻传输协议，Network News Transfer Protocol, RFC-977）访问新闻服务器。

ISP 通常向接入 ISP 网络的客户提供 NNTP 和 SMTP 服务。出于安全和使用控制的原因，ISP 通常限制为只能从自己的网络之内访问（包括自己的拨号连接）。换句话

说，如果你从 Internet 上的其他地方连接进来，那就可能无法使用这些服务。对服务器的访问请求可能被防火墙隔断，即使不是这样，你发出的邮件也可能被退回来，出错消息是“无中继（no relaying）”，新闻服务器也会拒绝服务，说“用户未认证（unauthorized use）”。

你当然是经过认证的，那么该怎么办呢？SSH 端口转发！在 SSH 会话之上，把 SMTP 和 NNTP 连接转发到一台 ISP 内部的主机上，你的连接看上去就是来自那台主机，这样，就越过了基于 IP 地址的限制。可以分别用 SSH 命令转发每个端口：

```
$ ssh -L2025:localhost:25 smtp-server ...
$ ssh -L2119:localhost:119 nntp-server ...
```

另一种情况是你在 ISP 内一台运行 SSH 的机器上有账号，但不能直接登录邮件或新闻服务器，那么可以这样：

```
$ ssh -L2025:smtp-server:25 -L2119:nntp-server:119 shell-server ...
```

这是一种主机外转发，转发路径上的最后一环不在 SSH 的保护之下。[9.2.4] 不过既然这样，转发不是为了保护信息而是为了越过源地址限制，那也没什么问题。不管怎么样，你的邮件信息和发布的新闻一旦发出去就是不安全的了。（如果想保护这些东西，你需要做数字签名或者单独加密，例如，用 PGP 或 S/MIME。）

在任何情况下，都可以在 `~/.pinerc` 文件中设置 `smtp-server` 和 `nntp-server` 两个选项，以使 Pine 使用转发的端口：

```
smtp-server=localhost:2025
nntp-server=localhost:2119
```

11.3.3 使用连接脚本

Pine 设置选项 `rsh-path` 不仅可以指向 `rsh` 或 `ssh`，还可以指向任何其他程序，比如根据你的任何需求定制的脚本，这非常有用。你这样做可能出于以下几个原因：

- `rsh-path` 是全局设置，对每一个远程邮箱都起作用。也就是说，所有远程邮箱要么都用这个设置，要么都不用。如果有多个远程邮箱，而只有几个可以通过 SSH/`imapd` 访问，这就麻烦了。如果 SSH 不能获取 IMAP 连接，Point 将直

接尝试建立TCP连接，你要一直等到这个操作以失败告终。如果服务器的防火墙悄无声息地把SSH端口阻塞掉，那可就够你等的了。

- “多转发（multiple forwarding）”问题。你也许想过不用单独的SSH会话设置转发，而是在Pine的rsh-path选项中加一些参数：

```
rsh-command=%s %s -q -l %s -L2025:localhost:25 exec /etc/r%sd"
```

如果访问多个邮箱，这就难办了。因为这条命令不仅对每个邮箱都有效，而且还可能同时运行很多次。一旦转发的端口已经建立，后面的调用都会出错。更明确地说是，SSH1和OpenSSH完全失败，而SSH2发出警告后继续操作。

定制连接脚本可以解决这一类以及其他一些问题。下面的Perl脚本检验目标服务器，如果已知名主机数据库中没有这个主机名，就立刻返回错误。这意味着Pine要把rsh-path命令快速传递到其他服务器上，并尝试直接建立IMAP连接。该脚本也能发现是否有SMTP和NNTP转发存在，当且仅当它们不存在时在SSH命令中实现之。使用这个或其他类似脚本的方法是，令Pine的rsh-path指向它，并使rsh-command与之保持一致：

```
rsh-path=/path/to/script
rsh-command=%s %s %s
```

下面有一个简单的Perl脚本可以实现上面的功能：

```
#!/usr/bin/perl

# TCP/IP 模块
use IO::Socket;

# 取得 Pine 传递的参数
($server,$remoteuser,$method) = @ARGV;

die "usage: $0 <server> <remote user> <method>"
unless scalar @ARGV == 3;

if ($server eq "mail.isp.com") {
    # 在这台主机上，我必须编辑自己的imapd
    $command = 'cd ~/bin; exec imapd';
} else if ($server eq "clueful.isp.com") {
    # 在这台主机上，POP 和 IMAP 服务器都在预期位置
    $command = 'exec /etc/r${method}d';
} else {
    # 示意 Pine 继续前进
    exit 1;
}
```

```
$smtp = 25; # 对 SMTP 的知名端口
$nnntp = 119; # 对 NNTP 的知名端口
$smarty_proxy = 2025; # 转发 SMTP 连接的本地端口
$nnntp_proxy = 2119; # 转发 NNTP 连接的本地端口
$ssh = '/usr/local/bin/ssh1'; # 我要运行哪个 SSH?

# 尝试达到转发 SMTP 端口，仅在失败时转发。同样仅在未处于域“home.net”中时转发。
# 思路是这样：该网络是直接访问 ISP 的邮件和新闻服务器的主网络。

$do_forwards = !defined($socket = IO::Socket::INET->new("localhost:$smarty_proxy"))
    && `domainname` !~ /HOME.NET/i;

# 经整理
close $socket if $socket;

# 转发时设置转发选项。这样就假设分别名为“mail”和“news”的邮件和新闻服务器在你的
# ISP 域中；这是个常用而方便的惯例。

@forward = ('-L', "$smarty_proxy:mail:$smtp", '-L', "$nnntp_proxy:news:$nnntp");
if ($do_forwards);

# 准备给 ssh 的参数
@ssh_argv = ('-a', '-x', '-q', @forward, "$remoteuser@$server");

# 运行 ssh
exec $ssh, @ssh_argv, $command;
```

11.4 Kerberos 与 SSH

Kerberos 是一种认证系统，其设计目的在于，让那些可能被监控的网络和不在集中控制之下的工作站能进行安全操作。^[1.6.3] 它是 Athena 计划（Project Athena）的一部分。Athena 计划是 1983 ~ 1991 年间 MIT 进行的一项广泛的研究开发工作，起初由 IBM 和 Digital Equipment Corporation 资助。Athena 对计算机领域其他许多方面技术的发展都做出了贡献，其中包括著名的 X Window 系统。

Kerberos 与 SSH 从功能到设计都大不相同；每一个都包含另一个没有的功能和服务。本例中我们将对这两个系统详细进行比较，然后探讨如何将两者结合起来，并利用各自的优点。如果你的站点已经有 Kerberos，你可以加上 SSH，同时保持已有的账号库和认证体系。（图 11-11 表明了 Kerberos 与 SSH 的设置体系已知的地方。）如果你没用 Kerberos，由于它有相当多的优点，我们建议你装一个，就大规模计算环境而言，Kerberos 有特别的吸引力。

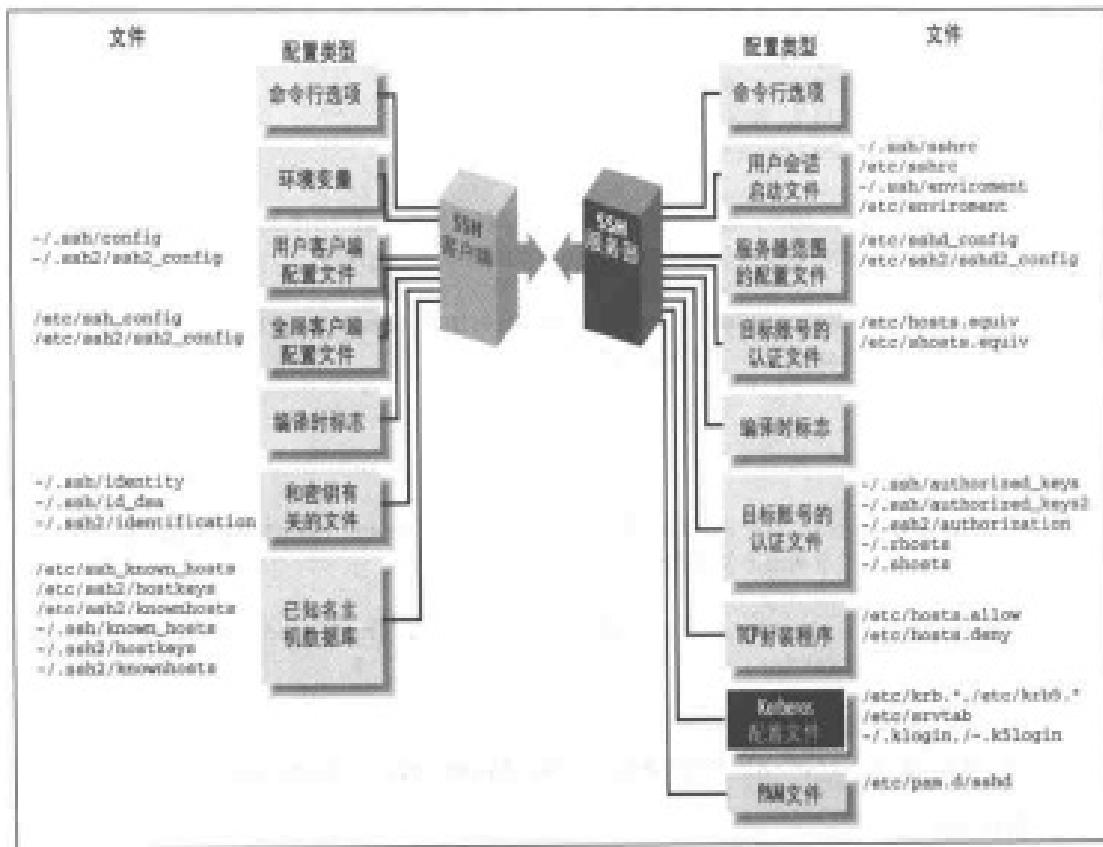


图 11-11：Kerberos 设置（高亮显示部分）

Kerberos 协议有两个版本，Kerberos-4 和 Kerberos-5，它们都可以从 MIT 免费获得：

<ftp://athena-dist.mit.edu/pub/kerberos/>

Kerberos 的当前版本是 Kerberos-5，目前 MIT 对 Kerberos-4 的开发工作已经不是很积极了。即便如此，很多情况下还会用到 Kerberos-4，尤其是一些商业系统（例如，Sun Solaris，Transarc AFS）中通常会预装 Kerberos-4。SSH1 支持 Kerberos-5，OpenSSH1 支持 Kerberos-4。目前的 SSH-2 协议还没有定义 Kerberos 认证方法，但在本书（英文版）出版时，将发布 SSH2.3.0，其中包含支持 Kerberos-5 的测试包；这方面内容我们没有提及，不过本质上应该和 SSH1 中介绍的相同。

11.4.1 SSH 与 Kerberos 比较

虽然 Kerberos 和 SSH 解决的问题有很多相同之处，但二者却有很大的不同。SSH 是一个轻量级的软件包，很容易部署，其设计目标是使它的工作给现存系统带来的变化最小。Kerberos 正好相反，它要求我们在使用之前建立重要的基础结构。

11.4.1.1 基础结构

考虑一个例子：如何让用户在两台计算机之间建立安全会话？用 SSH 很简单，在其中一台上安装 SSH 客户端，另一台上装服务器，启动服务器，就可以开始连接了。然而 Kerberos 要求完成下面这些管理任务：

- 建立至少一台 Kerberos 密钥分配中心（Key Distribution Center, KDC）主机。KDC 在 Kerberos 系统中居于核心地位，必须保证是高度安全的；通常其上除了 KDC 以外不运行任何东西，也不允许远程登录访问，并且放置在绝对安全的物理环境中（注 10）。没有 KDC，Kerberos 就不能工作，所以明智的选择是建立备份，或者将 KDC 置于从属位置，而且必须定期与主 KDC 同步。根据你的需要，某台 KDC 主机也可以运行一个远程管理服务器、一个在 Kerberos-5 中兼容 Kerberos-4 的证书转换服务器，以及其他服务器程序。
- 在 KDC 数据库中为每一位 Kerberos 用户创建账号（或者叫“用户代理，user principal”）。
- 在 KDC 数据库中为每一个打算用 Kerberos 认证其客户端的应用程序服务器创建账号（或称“服务代理，service principal”）。每一台主机上的每一个服务器都需要分别设置代理。
- 向每个服务器各自的主机分发服务代理的密钥文件。
- 为整个网络编写一个 Kerberos 配置文件 (*/etc/krb5.conf*)，并安装在所有主机上。
- 安装能识别 Kerberos 的应用程序。Kerberos 对 TCP 应用程序不透明，这一点与 SSH 不同。例如：有些版本的 *telnet* 可以与 Kerberos 结合使用，以加强认证，并给远程登录会话加密，安装这样的 *telnet* 就相当于 *ssh* 了。
- 安装时钟同步系统，如网络时间协议（Network Time Protocol, NTP）。Kerberos 要靠时间戳才能正常工作。

显然，若要部署 Kerberos，要做的工作比 SSH 更多，对现有系统的改动也更大。

注 10： 尽管如此，如果需要远程登录访问 KDC，SSH 仍然是很好的选择！

11.4.1.2 与其他应用程序集成

SSH 和 Kerberos 是为不同的应用设计的。SSH 是一组程序，凭借 SSH 协议协同工作，SSH 的设计目标是与现有应用程序结合使用，并且引起的变化最小。考虑一下 CVS [8.2.6.1] 和 Pine [11.3]，它们为执行远程程序，在内部调用了不安全的 *rsh*。如果在设置中用 *ssh* 代替 *rsh*，程序的远程连接就变成安全的了；*ssh* 的介入对程序及其远程伙伴来说都是透明的。与之对应的是，如果应用程序直接与 TCP 服务建立网络连接，只需简单地告诉应用程序使用不同的服务器地址和端口，就可以通过 SSH 端口转发保护该连接。

在另一方面，Kerberos 的设计是一种认证结构，辅以一组编程库（注 11）。这些库的作用是把 Kerberos 认证和加密机制加入已有的应用程序中；这个过程称为程序的 Kerberos 化。MIT 发布的 Kerberos 中集成了一些常用的 Kerberos 化服务，如，*telnet*、*ftp*、*rsh*、*su* 等的安全版本。

11.4.1.3 认证安全性

Kerberos 非常复杂，其中包含 SSH 所没有的属性和能力。Kerberos 的一个主要成功之处是它的认证器（就是密码、密钥之类）传输与存储机制。为了解释清楚这一优点，我们将 Kerberos 的许可证系统与 SSH 的密码和公钥认证系统做一比较。

SSH 密码认证要求每次登录都必须输入密码，每次密码都通过网络传输。当然，因为 SSH 将连接加密，在传输过程中窃取密码不容易。然而，当密码到达另一端的 SSH 服务器，等待认证之前，却是以明文形式存在的，如果敌人攻破了远程主机，就有机会得到你的密码。

另外，SSH 密钥认证会要求你把私钥存储在每一台客户主机上，对每一个你想用来登录的服务器账号都要为其建立认证文件。这样就引发安全和（密钥）分配问题。虽然存储密钥时可以用口令将其加密，但终究还得把密钥存储在很多人都可以访问的机器上，而 Kerberos 根本就不存在这种弱点。敌人能偷走你加密的私钥，用离线字典攻击它，尝试猜出你的口令。一旦成功，敌人就能访问你的账号了，你要是发现了问题，就得更换所有的密钥和认证文件。如果你在不同的机器上有多个账号的话，这可得花相当的时间，也很容易出错。如果漏掉了一个，那就麻烦了。

注 11：SSH2 最近也在向这一模式发展。其组织方式与此类似，有一组实现 SSH-2 协议的库，客户端和服务器程序可以通过 API 访问这些库。

Kerberos能保证尽可能少地传输用户的密码（注 12），而且不会将其存储于 KDC 之外。当用户在 Kerberos 系统上认证时，认证程序（*kinit*）将她的密码与 KDC 作一交换，然后立即清除，密码永远不会以任何形式在网络上传输，也不会存储在磁盘上。随后，如果客户端程序想用 Kerberos 认证，就得传送一个“许可证（ticket）”，这是 *kinit* 缓存在磁盘上的几个字节，用它向 kerberos 化的服务器证明用户身份。当然，许可证的缓存文件仅对其用户可读，但即使许可证被窃，能做的事情也有限：许可证过一段时间就会失效，一般来说是几个小时，而且特定的许可证只对特定的客户端 / 服务器 / 服务的组合有效。

如果 Kerberos 许可证缓存文件被窃，也可以用字典对其实施攻击，但情况有很大不同：里面没有用户密码。缓存的密钥属于服务器代理，而且通常随机产生，因此对字典攻击来说许可证不像密码那么脆弱。敏感的密钥只存储在 KDC 中；因为 Kerberos 的理论是：系统管理员可能几乎无法控制一大堆不同种类不同用途的服务器和工作站，而有效保护一小部分使用受到限制的机器却容易得多。Kerberos 之所以如此复杂，很大程度上是源自这一原理。

11.4.1.4 账号管理

Kerberos 有很多服务也超过了 SSH 的功能范围。它的集中式用户账号数据库将不同种类操作系统中的数据库统一起来，使你能够管理惟一的账号集合，而不必维持多个数据集的同步。Kerberos 支持访问控制列表和用户策略管理，可以严格定义哪个代理允许执行什么操作。这称为授权（*authorization*），与认证（*authentication*）相对。最后，Kerberos 的服务范围可划分成多个域（realm），每一个都有自己的 KDC 和用户账号集。这些域可以安排成层次结构，管理员可以在父子域和同级域之间建立信任关系，使其之间能进行自动交叉认证。

11.4.1.5 性能

Kerberos 认证总体上说比 SSH 公钥认证更快些。这是由于 Kerberos 通常用 DES 或者 3DES，而 SSH 用的是公钥加密，这种算法用软件实现起来比任何对称加密算法都慢得多。如果你的应用程序需要创建很多短期的安全网络连接，而硬件系统又不是最快的，那么这一差异会很严重。

注 12：实际上密钥是根据用户的口令生成的，不过在此关系不大。

总结起来，Kerberos 系统比 SSH 应用范围更广，它的功能包括认证、加密、密钥分发、账户管理以及授权服务。它需要具备大量专门的知识和系统框架才能使用，而且将给现有系统带来极大的改变。SSH 的要求较少，但是具备一些典型的 Kerberos 所没有的功能，如，端口转发。SSH 用起来更快更简单，如果要以最小的影响保护现有的应用程序，那么 SSH 更有用。

11.4.2 在 SSH 中使用 Kerberos

Kerberos 是一种授权及认证（AA）系统。SSH 是一个远程登录工具，AA 是其操作的一部分，Kerberos 也是一个它可以使用的 AA 系统（你肯定也猜到了）。如果你的站点已经用了 Kerberos，那么与 SSH 结合也是顺理成章的，因为你可以把 SSH 用到现有的代理结构和访问控制中。

即使你现在没有用 Kerberos，也可以把 Kerberos 的优点和 SSH 结合在一起，构成一个集成的解决方案。在 SSH 中，最灵活的认证方法是基于代理的公钥系统。密码认证既烦人又有限制，因为要反复输入密码，而可信主机方法在很多情况下不适用，或者说不够安全。不过公钥认证增加了大量事前的管理工作：用户必须创建、分配、维护其密钥，还要管理多个 SSH 认证文件。如果一个大规模站点中有很多非技术性用户，这就是很大的问题，也许代价会过高了。Kerberos 中具备 SSH 所缺乏的密钥管理功能。SSH 与 Kerberos 结合起来，表现得跟公钥认证很像：具有不会泄漏用户密码的加密认证；许可证缓存带来的好处和密钥代理相同，使用户只需登录一次。但是不用生成密钥，不用设置认证文件，不用编辑配置文件；Kerberos 会自动完成所有这些工作。

这也会带来一些损失。首先，只有 Unix 的 SSH 包支持 Kerberos；我们还不知道哪个 Windows 或 Macintosh 的产品包含 Kerberos 功能。目前仅有 SSH-1 协议支持 Kerberos，不过现在 SECSH 正在把 Kerberos 加入 SSH-2 中。其次，公钥认证与 SSH 其他一些重要功能密切相关，如认证文件中的强制命令，它无法与 Kerberos 的认证机制结合使用。这也是 Unix SSH 的发展过程带来的缺憾。当然了，如果需要，你还是能使用公钥认证的。你可能会发现，Kerberos 的访问控制机制能满足绝大多数需求，在少数情况下控制的粒度要更细一些，这时可以用公钥方式。

下面几部分中，我们将解释如何使用支持 Kerberos 的 SSH。要是你的站点安装了这种 SSH，这些介绍就足够让你上手了。我们不可能讨论搭建 Kerberos 框架的所有细

节，那太恐怖了，不过如果你有自己的系统，而且也想试试看的话，我们也给出了从头开始快速设置 Kerberos 的要点。不过这些只是建议，问题还没有完全说明白。如果你打算使用、安装和管理 Kerberos 化的 SSH，那么除了这里介绍的之外，你还需要对 Kerberos 有更完整的理解。从下面这个地方着手就很好：

<http://web.mit.edu/kerberos/www/>

11.4.3 Kerberos-5 简介

本节将介绍代理 (principal)、许可证 (ticket)、可更新许可证 (TGT) 等重要概念，然后给出一个实际的例子。

11.4.3.1 代理和许可证

Kerberos 可以为用户，或者提供或请求服务的软件进行认证。这些实体有个名称，叫做代理。代理由三部分组成：名字 (name)、实例 (instance)、域 (realm)，记为 *name/instance@REALM* (注 13)。具体说来：

- *name* 通常与主机操作系统中的一个用户名对应。
- *instance* 可以为空，其典型的用途是区分同一名字的不同角色。例如，用户 *res* 可以有一个常规的用户级代理 *res@REALM* (注意 *instance* 为空)，可是他还能有第二个代理 *res/admin@REALM*，这个代理具有特殊的权限，是用户 *res* 在充当系统管理员的时候用的。
- *realm* 是管理上的划分，标识一个单独的 Kerberos 代理库实例 (也就是在共同管理控制之下的代理列表)。每一个主机都给指定一个 *realm*，其标识与认证判断密切相关，我们将对此做一简短讨论。习惯上 *realm* 名永远大写。

我们已经说过，Kerberos 的基础是许可证。如果想使用某项网络服务 (比如说，用某台主机上的 *telnet* 服务远程登录)，你必须从 Kerberos KDC 得到使用那项服务的

注 13：这是 Kerberos-4 的情况。事实上在 Kerberos-5 中，代理由域和任意数量的“组件”(component) 构成。前两个组件按惯例是作为名字和实例使用的，这与 Kerberos-4 一样。

一个许可证。许可证中包含一个认证者（authenticator），它向提供服务的软件证明你的身份。由于你和这项服务都必须向 KDC 确认身份，因此各自都需要代理。

系统管理员负责建立代理，将其加入 KDC 数据库。每一个代理都有一个密钥，只有代理的所有者和 KDC 知道此密钥；Kerberos 协议的操作过程即是基于这一事实。举个例子来说，当你为一项服务申请许可证时，KDC 会传给你几个字节，这是用该服务的密钥加过密的。因此，只有此项服务能对许可证做解密和验证。此外，如果能成功解密，就说明许可证是 KDC 发出的，因为只有服务本身和 KDC 知道该项服务的密钥。

用户代理密钥是根据用户的 Kerberos 口令生成的。服务代理的密钥通常存储于服务器运行的那台主机上的 */etc/krb5.keytab* 文件中，服务中调用一个 Kerberos 例程，读取该文件，解出密钥。只要能读取这个文件，就能假扮该服务，所以显然不能让什么人都访问，必须将其保护起来。

11.4.3.2 用 kinit 获取证书

我们举个例子，看看 Kerberos 实际是怎么操作的。假设你在 Unix 主机 *spot* 上，所属的域是 FIDO，你想用 Kerberos 化的 *telnet* 登录另一台主机 *rover*。首先，运行 *kinit* 命令，获取 Kerberos 证书：

```
[res@spot res]$ kinit  
Password for res@FIDO: *****
```

kinit 认为，既然你的用户名是 *res*，主机 *spot* 在 FIDO 域，那么你要的就是 *res@FIDO* 的证书。如果你请求的是其他代理的许可证，就得把名字作为参数告诉 *kinit*。

11.4.3.3 用 klist 列出证书

在用 *kinit* 成功获取证书之后，可以运行 *klist* 命令，检查一下所有当前已获得的证书：

```
[res@spot res]$ klist  
Ticket cache: /tmp/krb5cc_84629  
Default principal: res@FIDO  
Valid starting     Expires            Service principal  
07/09/00 23:35:03  07/10/00 09:35:03  krbtgt/FIDO@FIDO
```

目前你仅有一个许可证，属于krbtgt/FIDO@FIDO服务。这就是你的Kerberos TGT，是你一开始具有的许可证：等会儿会向 KDC 出示，表明你已经成功通过 res@FIDO 的认证了。注意，TGT 是有有效期的：10 小时之后将过期。之后你必须再次执行 *kinit*，重新认证。

11.4.3.4 运行 Kerberos 化的应用程序

得到证书后，现在就可以 *telnet* 远程主机了：

```
[res@spot res]$ telnet -a rover
Trying 10.1.2.3...
Connected to rover (10.1.2.3).
Escape character is '^].
[Kerberos V5 accepts you as "res@FIDO"]
Last login: Sun Jul  9 16:06:45 from spot
You have new mail.
[res@rover res]$
```

telnet 客户端的 *-a* 选项表示自动登录：这是说，与远程服务器端协商进行 Kerberos 认证。这一步成功了：远程服务器接受了你提供的 Kerberos 证书，你不用输密码就可以登录。如果回到 spot，运行 *klist*，将会看到：

```
[res@spot res]$ klist
Ticket cache: /tmp/krb5cc_84629
Default principal: res@FIDO
Valid starting     Expires            Service principal
07/09/00 23:35:03  07/10/00 09:35:03  krbtgt/FIDO@FIDO
07/09/00 23:48:10  07/10/00 09:35:03  host/rover@FIDO
```

注意，你现在有第二个许可证了，是用于 host/rover@FIDO 服务的。这个代理用于访问 rover 主机上的远程登录和命令执行服务，如，Kerberos 化的 *telnet*、*rlogin*、*rsh* 等等。当运行 *telnet -a rover* 时，*telnet* 客户端用你的 TGT 向 KDC 请求一个 host/rover@FIDO 的证书。KDC 验证你的 TGT，证实你最近曾经作为 res@FIDO 登录，然后发出你所请求的许可证。*telnet* 把这个新许可证存于你的 Kerberos 证书缓存中，下一次你连接 rover 的时候就不用再访问 KDC，可以直接用缓存里的证书（至少在证书过期之前如此）。然后，*telnet* 客户端把 host/rover@FIDO 证书提交给 *telnet* 服务器，服务器验证无误，相信这个客户端已经在 KDC 上验明身份，就是 res@FIDO 了。

11.4.3.5 授权

我们已经说明了认证的过程，那么授权又是如何进行的呢？*rover*上的*telnet*服务器相信你就是 res@FIDO，但是为什么要让 res@FIDO 登录呢？这又回到我们提过的 host 与 realm 相一致的问题上了。[11.4.3.2]你在命令中没有指定其他用户，因此 *telnet* 客户端告诉服务器你要用 res 账号登录到 *rover* 上。（改变账号的方法是 *telnet -l username*。）因为 *rover* 也在 FIDO 域中，所以 Kerberos 使用缺省的授权规则：如果主机 H 在 R 域中，那么 Kerberos 就允许代理 u@R 访问账号 u@H。这意味着系统管理员在维护操作系统用户和 Kerberos 代理之间的一致性。如果你想登录到你朋友 Bob 的账号，会发生下面的情况：

```
[res@spot res]$ telnet -a -l bob rover
Trying 10.1.2.3...
Connected to rover (10.1.2.3).
Escape character is '^'.
[Kerberos V5 认为你是 "res@FIDO"]
telnetd: Authorization failed.
```

要注意的是，上面你仍然通过了认证：*telnet* 服务器已经承认你是 res@FIDO 了。不过判断授权与否的结论却是否定的：Kerberos 决定代理 res@FIDO 不能访问账号 bob@rover。Bob 可以在 *rover* 上创建一个文件 *~bob/.k5login*，在里面加上你的代理名称 res@FIDO，这样你就可以登录到他的账号了。*.k5login* 会覆盖缺省的授权规则，所以他还得把他自己的代理也放进这个文件，不然他自己就没法登录自己的账号了。所以，这个授权文件应该是这样的：

```
rover:~bob/.k5login:
bob@FIDO
res@FIDO
```

11.4.4 在 SSH1 中使用 Kerberos-5

编译 SSH1 时加上 *--with-kerberos5* 选项，就可支持 Kerberos。[4.1.5.7] 如果 Kerberos 的支持文件（库文件与 C 的头文件）不在标准位置，*configure* 就找不到它们，此时可以这样告诉 *configure* 到哪里去找这些文件：

```
# 仅对 SSH1
$ configure ... --with-kerberos5=/path/to/kerberos ...
```

有两点需要注意：

- 在 MIT Kerberos-5 Release 1.1 中，库文件 *libcrypto.a* 的名字改成了 *libk5crypto.a*，而 SSH1 的编译文件没有据此更新。可以换掉 SSH1 的 Makefile，也可以简单地这样：

```
# cd your_Kerberos_library_directory
# ln -s libk5crypto.a libcrypto.a
```
- *auth-kerberos.c* 中使用的 *krb5_xfree()* 例程在 1.1 版中也消失了。把所有的 *krb5_xfree* 换成 *xfree* 就行了。

注意：如果在编译 SSH 时内嵌了 Kerberos 支持，即使不用 Kerberos 认证，得到的程序也只有在安装了 Kerberos 的系统中才能运行。程序中很可能连接到 Kerberos 共享库，程序运行时要求这些库必须存在。还有，SSH 启动时初始化 Kerberos，所以也需要一个有效的 Kerberos 主机配置文件 (*/etc/krb5.conf*)。

安装 SSH 之后，为了明确起见，虽然服务器范围配置关键字 *KerberosAuthentication* 缺省状态是启用，我们仍然建议在 */etc/sshd_config* 中将其设为“yes”，

```
# 仅对 SSH1
KerberosAuthentication yes
```

此外，还应确保代理 *host/server@REALM* 存在于 KDC 数据库中，并且其密钥必须存储在服务器的 */etc/krb5.keytab* 处。

运行支持 Kerberos 的 SSH，本质上和我们前面描述的 Kerberos 化的 *telnet* 一样；图 11-12 说明了这一过程。[11.4.3.4] 客户端只需要运行 *kinit*，获取 Kerberos TGT，然后运行 *ssh -v*。如果 Kerberos 认证成功，就会看到：

```
$ ssh -v server
...
server: Trying Kerberos V5 authentication.
server: Kerberos V5 authentication accepted.
...
```

服务器日志中的记录如下：

```
Kerberos authentication accepted joe@REALM for login to account joe from client_
host
```

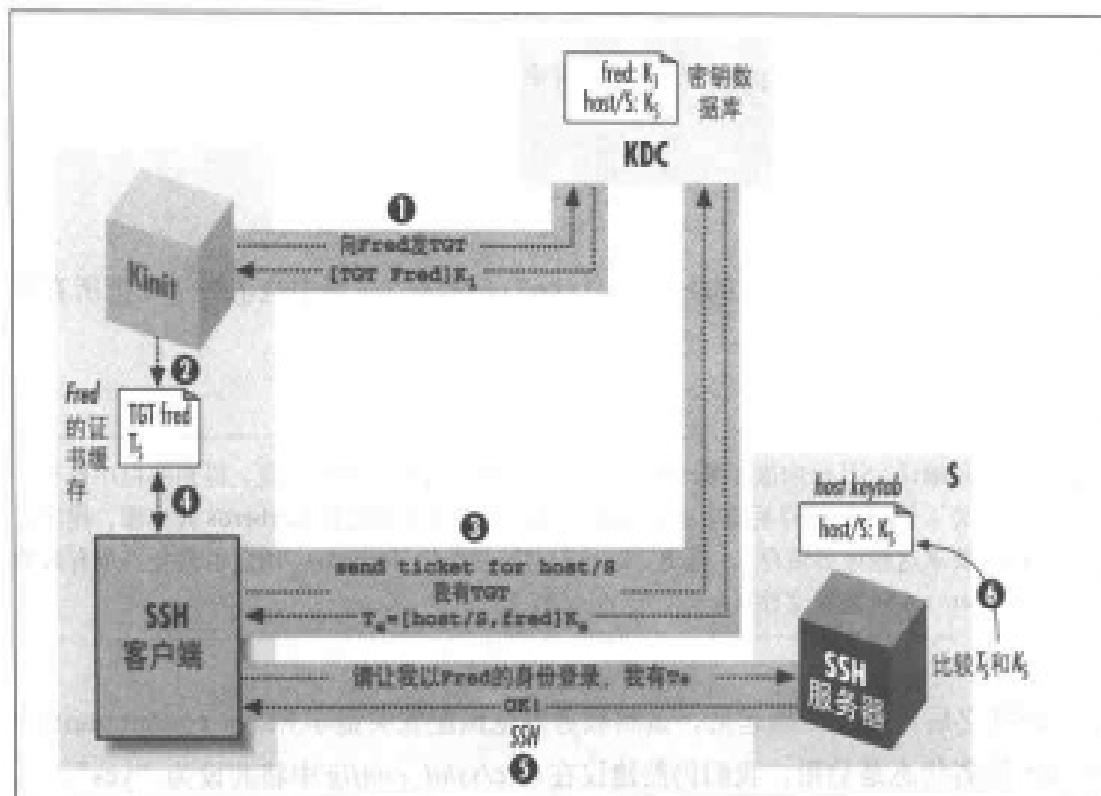


图 11-12：使用 Kerberos 认证的 SSH

如果想让其他的人用 Kerberos 和 “`ssh -l your_username`” 登录你的账号，那你所要做的工作就和用 `telnet` 一样，必须创建一个 `~/.k5login` 文件，然后把他们的代理名字写入该文件，别忘了还有你自己的。

11.4.4.1 Kerberos 密码认证

如果 SSH 服务器启用 Kerberos 认证，密码认证的行为就变了。此时，密码由 Kerberos 认证，而不是主机操作系统。在一个完全 Kerberos 化的环境中，本地口令可能根本不管用，因此这种方式是必需的。不过在混合的环境里，Kerberos 认证失败之后如果能允许 SSH 回溯到本地主机操作系统去认证，可能会很有用。SSH 控制此功能的服务器选项是 `KerberosOrLocalPasswd`：

```
* SSH1, OpenSSH
```

```
KerberosOrLocalPasswd yes
```

这种回溯机制是一种有用的故障预防手段：在 KDC 不能正常工作的情况下，仍可以通过 OS 密码对用户进行认证（虽然公钥认证在防故障方面可能做得更好）。

Kerberos化的密码认证还有一个功能，就是`sshd`在登录时可以存储TGT，因此在获取远程主机的Kerberos证书时就不用运行`kinit`，也不用再次输入密码了。

11.4.4.2 Kerberos 与 NAT

SSH经常跨防火墙使用，如今这样的环境常常牵扯到网络地址转换的问题。很不幸，Kerberos在NAT方面有个很严重的问题。Kerberos证书中通常包含一个IP地址列表，表示允许这些地址上的主机使用此证书；也就是说，客户端必须从这些地址中的某一个提交证书。缺省情况下，`kinit`请求到的TGT局限于其运行的主机IP。用`klist -a`可看到这一点：

```
[res@spot res]$ klist -a -n
Ticket cache: /tmp/krb5cc_84629
Default principal: res@FIDO
Valid starting     Expires            Service principal
07/09/00 23:35:03  07/10/00 09:35:03  krbtgt/FIDO@FIDO
                  Addresses: 10.1.2.1
07/09/00 23:48:10  07/10/00 09:35:03  host/rover@FIDO
                  Addresses: 10.1.2.1
```

(`-n`开关告知`klist`用数字方式显示地址，不用将其转换成名字) 主机的IP地址是10.1.2.1，因此KDC发出的TGT只能在这一个地址上使用。如果主机有多个网络接口或地址，将一并列出。当你基于这个TGT请求后续服务的证书时，这些证书也只能限制在同样的一些地址上使用。

现在假设这样的情形：你要连接NAT网关另一侧的SSH服务器，网关会重写你的(客户端的)IP，而KDC与你同在NAT网关的这一侧。

从KDC获得的服务证书包含你的真实IP。然而SSH服务器从连接的源地址中看到的IP是经过NAT转换的。请注意，这一地址与证书中加密过的地址并不吻合，所以会拒绝认证。在这种情况下，`ssh -v`执行结果如下：

```
Trying Kerberos V5 authentication.
Kerberos V5: failure on credentials (Incorrect net address).
```

图11-13是对此问题的说明。此刻还没有什么好办法。有一个权宜之计，就是用文档里未提及的`kinit -A`开关，这样，`kinit`返回的证书里就根本没有地址。这种做法降低了安全性，因为别人能偷走你缓存里的证书，将其用在其他地方，不过毕竟可以绕过地址不匹配的问题。

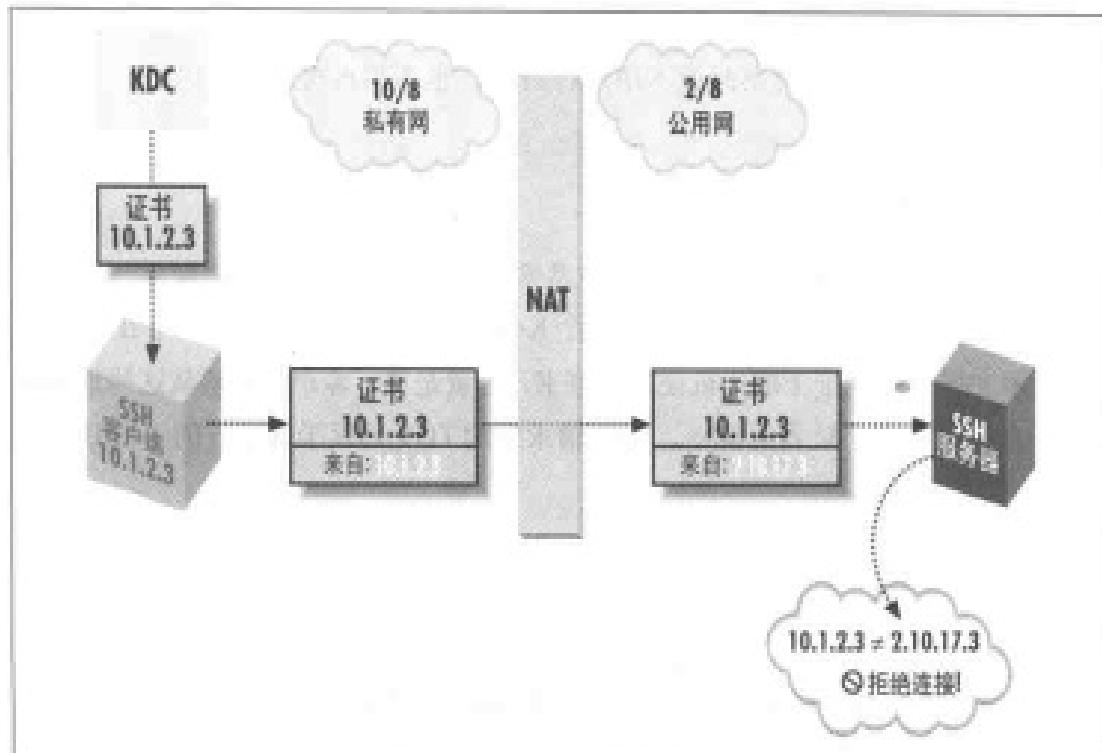


图 11-13: Kerberos 与 NAT

11.4.4.3 跨域 (cross-realm) 认证

Kerberos 域是在管理上分隔控制的不同代理的集合。例如，可能有两个部门，销售部和工程部，他们彼此不信任（当然这样说只是举例子）。销售部的人不愿意任何奇怪的工程师在他们的地盘里创建账号，工程师当然也不想让什么销售狂人登录进来闹事。所以得创建两个 Kerberos 域，SALES 和 ENGINEERING，每一个域都有各自的管理员管理账号。

两个域的用户可以跨域访问对方的资源，但双方的用户无法直接访问对方的资源。问题是销售部和工程部肯定得一起工作，销售员要登录工程部的机器试验新产品，工程部必须访问销售部的桌面系统，解决他们层出不穷的问题。假设工程师 Erin 需要销售部的 Sam 访问她在工程部机器上的账号 erin@bulwark，她可以把 Sam 的代理名写进她的 Kerberos 认证文件，如下所示：

```
bulwark:~erin/.k5login:
      这次很远行许多更妙的事，但这次我必须承认，我已
      erin@ENGINEERING
      sam@SALES
```

但这样做不行。Sam 登录时需要 host/bulwark@ENGINEERING 的服务许可证。只

有ENGINEERING域的KDC能发出这种许可证，而ENGINEERING KDC不可能知道代理名sam@SALES。一般说来，ENGINEERING域的主机没法认证来自SALES域的代理。看起来Sam在ENGINEERING域中也需要一个代理，但是这么做违反了划分独立域的整体概念，是很糟糕的办法。这样做也很麻烦，因为Sam每次想访问另一个域里面的资源时都必须再运行一次kinit。

解决此问题的办法称为跨域认证。首先，每一个域中的每一台机器上都要建立一个/etc/krb5.conf，用来描述这两个域；Kerberos只知道列在此配置文件中的域。然后，两个域的管理员在两者之间建立一个共享密钥，称为跨域密钥（cross-realm key）。每一个域中各有一个代理的名字是经过特殊设计的，跨域密钥是它们的公用密钥。这个密钥有方向性，它使一个KDC向另一个域中发送TGT成为可能；而另一个KDC可以检验该TGT，以证实它是从它的信任伙伴域中用共享密钥发出的。只要有一个跨域密钥存在，就可以在一个域中向另一个域提供认证标识。如果信任机制是对称的，也就是说，如果每一个域都应该信任另一个域，那么，就需要两个跨域密钥，一个方向上一个。

Kerberos-5的层次域结构

如果有很多域，系统马上就会变得不堪重负。如果要在每两个域之间都建立跨域信任关系，那就必须手工给每一对域都建立跨域密钥。Kerberos-5支持建立域的层次结构，可用于解决此类问题。在域名中包含点号，如，ENGINEERING.BIGCORP.COM，意味着（可能）存在BIGCORP.COM和COM域。如果要从SALES.BIGCORP.COM至ENGINEERING.BIGCORP.COM建立跨域认证，而Kerberos没有找到直接可用的跨域密钥，它就会先上行再下行，遍历层次结构中的相关域，沿着一条跨域关系链到达目标域。比如，如果从SALES.BIGCORP.COM至BIGCORP.COM已经存在跨域密钥，从BIGCORP.COM到ENGINEERING.BIGCORP.COM也有，那么从SALES到ENGINEERING就不需要单独设置也可以进行跨域认证了。当域的数目很多时，这种方法能够保持可伸缩的、完整的域间关系。

请注意目前Sam没有第二个代理sam@ENGINEERING。确切地说，ENGINEERING KDC现在就能证实Sam确实是已经通过SALES KDC认证的sam@SALES，因此，可以作为sam@SALES对其授权。在Sam通过SSH登录bulwark时，Kerberos注

意到目标机器与 Sam 的代理在不同的域，然后自动使用适当的跨域密钥，从 ENGINEERING 域中为其获取另一个 TGT。接着，Kerberos 再用这个 TGT 获取一个能让 sam@SALES 通过 host/bulwark@ENGINEERING 服务认证的证书。bulward 上的 sshd 读取 Erin 的 *~/.k5login* 文件，看到 sam@SALES 是可以访问的，就允许其登录。

这些是基本的思想。不过由于 SSH 的介入，跨域认证经常莫名其妙地出错。假设出现下面的情形：Sam 经常使用 bulwark，所以就给他在上面建立一个账号。系统管理员把 sam@SALES 写进 bulwark 的 *~sam/.k5login*，想让 Sam 可以用他的 SALES 证书登录到那里；但是这样不行。即使什么都设置对了，跨域 Kerberos telnet 也能工作，但是 SSH Kerberos 认证还是失败。更不可思议的是，所有其他的认证方式也开始出错了。Sam 的公钥认证设置是好的，以前也能工作，想让它在 Kerberos 下面工作，结果失败了，然后，再试验公钥认证，也全都失败了。如果不用密码认证，Sam 对这个问题永远也不会有头绪，最后，SSH 终于尝试用密码了：

```
[sam@sales sam]$ ssh -v bulwark
...
Trying Kerberos V5 authentication.
Kerberos V5 authentication failed.
Connection to authentication agent opened.
Trying RSA authentication via agent with 'Sam's personal key'
Server refused our key.
Trying RSA authentication via agent with 'Sam's work key'
Server refused our key.
Doing password authentication.
sam@SALES@bulwarks's password:
```

最后一条提示信息看起来完全不对：“sam@SALES@bulwark”？sshd -d 中找到的提示是这样的：

```
Connection attempt for sam@SALES from sales
```

SSH 把代理名错当成账号名了，就好像 Sam 输入的是 *ssh -l sam@SALES bulwark* 那样。当然了，没有 sam@SALES 这个账号，只有“Sam”。其实这个问题很快就能解决，只要明确指出登录的用户名是 sam 就行了，*ssh -l sam bulwark*，不过这看上去很麻烦。

出现这个怪问题的原因是 SSH 中用到的一个 Kerberos-5 功能，叫做认证名至本机名映射 (aname→lname mapping)。Kerberos 可以在很多操作系统中使用，有些操作

系统的用户命名规则无法简单地与Kerberos代理名称对应。可能有一些用户名中允许出现的字符在代理名中是非法的，或者多个操作系统的用户名语法规则相互矛盾。或者可能在合并两个已有网络时，你发现用户名有冲突，所以在某些系统中必须把代理 `res@REALM` 转换成 `res` 用户，而在另一些系统中，则要转换成 `rsilverman`。Kerberos-5的设计者们认为Kerberos最好能自动处理这个问题，所以Kerberos中增加了 `aname→lname` 机制，以便在不同的环境下把代理名转换成正确的本地账号名。

SSH1 使用了 `aname→lname` 机制。在进行 Kerberos 认证时，SSH1 客户端缺省认为代理名（而不是当前本地账号名）就是目标账号名，也就是说，实际执行的命令是 `ssh -l sam@SALES bulwark`。接着，服务器对其做 `aname→lname` 映射，将其转换成本地账号。当代理名和服务器主机在同一个域中时，此操作可以自动进行，因为有一个缺省的 `aname→lname` 规则，在 `REALM` 就是主机所在域的情况下，把 `user@REALM` 转换成 `user`。然而，Sam 正在做的是跨域认证，所以两个域不同：他的代理是 `sam@SALES`，而服务器在 `ENGINEERING` 域。所以 `aname→lname` 映射失败了，`sshd` 还是把 `sam@SALES` 当成本地账号名。既然根本没有这样的账号，那么所有的认证肯定都会失败。

`ENGINEERING` 域的系统管理员可以给 `SALES` 域设置一个 `aname→lname` 映射来解决这个问题。不过，碰巧 MIT Kerberos-5 Release 1.1.1 的 `aname→lname` 功能像是还没完成。文档几乎没有，有些调用到的功能和文件也还不存在。即便如此，我们还是找到了足够的信息，给出一个能解决问题的例子。我们从程序文件 `src/lib/krb5/os/an_to_ln.c` 的注释中得到启发，想出了下面这段“`auth_to_local`”的声明，解决 Sam 的问题：

```
bulwark:/etc/krb5.conf:  
...  
[realms]  
ENGINEERING = {  
    kdc = kerberos.engineering.bigcorp.com  
    admin_server = kerberos.engineering.bigcorp.com  
    default_domain = engineering.bigcorp.com  
    auth_to_local = RULE:[1:$1]  
    auth_to_local = RULE:[2:$1]  
    auth_to_local = DEFAULT  
}
```

这些规则让这台主机上的 `aname→lname` 函数把所有域的 `foo@REALM` 或 `foo/bar@REALM` 形式的代理名映射为“`foo`”，同时也保持了本机域的缺省转换规则。

11.4.4.4 TGT 转发

回忆一下，Kerberos发出的许可证通常只能在请求它的主机上使用。如果你在某个主机spot上执行 `kinit`，然后用SSH登录到主机rover，那么就Kerberos而言你的位置是固定的。如果你想在rover上用一些Kerberos的服务，就必须再次执行 `kinit`，因为你的证书缓存在spot上。即便把证书缓存文件从spot拷贝到rover也不行，因为spot的TGT在rover上无效；你需要一个专门为rover发出的证书。如果你再次执行 `kinit`，你的密码倒是可以通过SSH安全地在网络上传输，但这还做不到只登录一次，还是很烦人。

SSH在公钥认证问题上也有同样的麻烦，当时的解决方法是代理转发。[\[6.3.5\]](#)与此类似，Kerberos解决这个问题用的是TGT转发。SSH客户端请求KDC根据客户端目前已经持有有效TGT这一事实，发出一个在服务器主机上有效的TGT，客户端接收到这个TGT之后，就将其传给`sshd`，后者把该TGT存在远程账号的Kerberos证书缓存中。如果操作成功，你在`ssh -v`的输出信息中将看到这样的内容：

```
Trying Kerberos V5 TGT passing.  
Kerberos V5 TGT passing was successful.
```

远程主机上执行`klist`命令可显示出被转发的TGT。

必须用`--enable-kerberos-tgt-passing`开关重新编译SSH，然后才能用TGT转发。还得用`kinit -f`命令请求一个可转发的TGT；否则，会出现下面的情况：

```
Kerberos V5 krb5_fwd_tkt_creds failure (KDC can't fulfill requested option)
```

11.4.4.5 SSH1 中 Kerberos 许可证缓存的 bug

1.2.28版之前的SSH1在Kerberos证书缓存处理上有严重缺陷。在一些环境中，SSH1把远程环境变量`KRB5CCNAME`错置成字符串“none”。这个变量控制证书缓存存储的位置。在证书缓存中有很敏感的信息；任何人如果偷走你的证书缓存文件，就可以在证书的有效期之内伪装成你。通常，证书缓存文件存放在`/tmp`目录下，这对每一台机器来说都是一个可靠的本地目录。将`KRB5CCNAME`置为“none”，意味着当用户执行`kinit`时，证书缓存建立在当前工作目录下名为`none`的文件中。这个目录很容易变成NFS文件系统，证书就可能被人用网络窥探器偷走。这个文件也可能成为文件系统的薄弱环节，因为也许有人能通过继承ACL得到访问该文件的权限，SSH设置的所有权和访问权就变得毫无意义了。

警告：不要在 1.2.28 以前版本的 SSH1 中使用 Kerberos。

根据 bug report 的反映，1.2.28 版的 SSH Communications Security 消除了这个 bug。请注意，如果 SSH1 编译时加上 Kerberos 支持功能，即使当前会话没有用 Kerberos 认证，这个问题也会出现。OpenSSH 中的 Kerberos-4 没有这个 bug。

11.4.4.6 Kerberos-5 安装指南

下面我们将给出一些简明的步骤，帮助你从零开始，快速安装一个可用的单主机 Kerberos 系统，我们使用的是 MIT Kerberos-5 1.1.1 版本。这一过程还远不完整，在某些环境或者 Kerberos 版本下可能出现错误，也可能会对你产生误导。我们的本意是在你想试试 Kerberos 的前提下，给你一个着手点。假设本地主机名为 *shag.carpet.net*，你选择的域名为 FOO，用户名为“fred”：

1. 编译并安装 krb5-1.1.1。我们用的编译参数是 `--localstatedir=/var`，这样 KDC 数据库文件将建立在 `/var` 目录下。
2. 运行：
`$ mkdir /var/krb5kdc`
3. 安装一个如下的 `/etc/krb5.conf` 文件。要注意日志文件；出错的时候检查一下日志会有帮助（也可以从中知道一些信息）。

```
[libdefaults]
    ticket_lifetime = 600
    default_realm = FOO
    default_tkt_enctypes = des-cbc-crc
    default_tgs_enctypes = des-cbc-crc

[realms]
    FOO = {
        kdc = shag.carpet.net
        admin_server = shag.carpet.net
        default_domain = carpet.net
    }

[domain_realm]
    .carpet.net = FOO
    carpet.net = FOO

[logging]
    kdc = FILE:/var/log/krb5kdc.log
```

```
admin_server = FILE:/var/log/kadmin.log
default = FILE:/var/log/krb5lib.log
```

创建 /var/krb5kdc/kdc.conf 如下:

```
[kdcdefaults]
    kdc_ports = 88,750

[realms]
    FOO = {
        database_name      = /var/krb5kdc/principal
        admin_keytab       = /var/krb5kdc/kadm5.keytab
        acl_file          = /var/krb5kdc/kadm5.acl
        dict_file         = /var/krb5kdc/kadm5.dict
        key_stash_file    = /var/krb5kdc/.k5.FOO
        kadmin_port        = 749
        max_life          = 10h 0m 0s
        max_renewable_life = 7d 0h 0m 0s
        master_key_type   = des-cbc-crc
        supported_enctypes = des-cbc-crc:normal des-cbc-crc:v4
    }
```

4. 运行:

```
$ kdb5_util create -s
```

这一步在 /var/krb5kdc 处创建 KDC 代理数据库, 要求你输入 KDC 所有者密码。此密码是 KDC 运行所必需的, 存储于 /var/krb5kdc/.k5.FOO 中, 这使 KDC 软件能在无人干预的情况下启动, 不过这显然不是明智的办法, 除非你的 KDC 机器保护得非常好。

5. 运行:

```
$ kadmin.local
```

本程序对代理数据库进行修改。产生下列 kadmin 命令:

```
kadmin.local: ktadd -k /var/krb5kdc/kadm5.keytab kadmin/admin kadmin/changepw
Entry for principal kadmin/admin with kvno 4, encryption type DES cbc mode with
CRC-32 added to keytab WRFILE:/var/krb5kdc/kadm5.keytab.
Entry for principal kadmin/changepw with kvno 4, encryption type DES cbc mode with
CRC-32 added to keytab WRFILE:/var/krb5kdc/kadm5.keytab.
```

```
kadmin.local: add_principal -randkey host/shag.carpet.net
```

```
WARNING: no policy specified for host/shag.carpet.net@FOO; defaulting to no policy
Principal "host/shag.carpet.net@FOO" created.
```

```
kadmin.local: ktadd -k /etc/krb5.keytab host/shag.carpet.net
```

```
Entry for principal host/shag.carpet.net with kvno 3, encryption type DES cbc mode
with CRC-32 added to keytab WRFILE:/etc/krb5.keytab.
```

```
kadmin.local: add_principal fred
```

```
WARNING: no policy specified for fred@FOO; defaulting to no policy
```

```
Enter password for principal "fred@FOO": ****
Re-enter password for principal "fred@FOO": ****
Principal "fred@FOO" created.
kadmin.local: quit
```

6. 现在，启动 KDC 和 *kadmin* 守护进程，使用的命令是 *krb5kdc* 和 *kadmind*。

如果一切顺利，你就可以通过 *kinit* 获得一个 TGT 了，用的密码就是创建“fred”代理时向 *kadmin.local* 输入的那个，可以用 *klist* 命令查看这个 TGT，还可以用 *kpasswd* 改变 Kerberos 密码。

7. 试试 Kerberos 化的 SSH 工作情况如何。

11.4.5 OpenSSH 中的 Kerberos-4

OpenSSH 也支持 Kerberos，不过只有较老的 Kerberos-4 标准。从用户角度看，机制几乎是一样的：在正常工作的 Kerberos 域中，用 *kinit* 获取一个 TGT，然后打开 *KerberosAuthentication* 开关（缺省状态就是打开的）运行 SSH 客户端。系统管理员编译 OpenSSH 时必须加上 *--with-kerberos4* 选项，以确保生成一个 Kerberos 主机代理，并在运行 SSH 服务器的主机上安装密钥，然后在 SSH 服务器设置选项中打开 *KerberosAuthentication*。主机代理为 *rcmd.hostname@REALM*（注 14），*keytab* 文件是 */etc/srvtab*。服务器的 *KerberosAuthentication* 缺省状态是 *on*，除非其启动的时候有 */etc/srvtab* 文件存在。

账号的访问控制由 *~/.klogin* 文件完成。在 Kerberos-4 中，如果一个账号的缺省代理文件存在，就不一定非要将其包括在 *~/.klogin* 中；缺省代理总具有访问权。

表 11-1 总结出了 Kerberos-4 和 Kerberos-5 中与 SSH 有关的显著区别。

表 11-1: Kerberos-4 和 Kerberos-5 中与 SSH 有关的区别

	Kerberos-4	Kerberos-5
主机代理	<i>rcmd.hostname@REALM</i>	<i>host/hostname@REALM</i>
配置文件	<i>/etc/krb.conf</i>	<i>/etc/krb5.conf</i>

注 14: Kerberos-4 的代理也包括名字 (name)、实例 (instance, 可选) 和域 (realm) 三部分，不过书写格式是 *name.instance@REALM*，而不是 Kerberos-5 中的 *name.instance@REALM*。

表 11-1: Kerberos-4 和 Kerberos-5 中与 SSH 有关的区别

	Kerberos-4	Kerberos-5
服务器代理文件	<code>/etc/srvtab</code>	<code>/etc/krb5.keytab</code>
认证文件	<code>~/.klogin</code>	<code>~/.k5login</code>

11.4.5.1 Kerberos-5 的 Kerberos-4 兼容状态

如果你有一个 Kerberos-5 的域，就不必只为了支持 OpenSSH 就再装一个 Kerberos-4 KDC。Kerberos-5 有一种 version 4 (v4) 兼容状态，在此状态下，v5 KDC 可以响应 v4 的请求。如果进入 v4 兼容状态，你就可以安装 v4 的 `/etc/krb.conf` 和 `/etc/krb.realms` 文件，将其指向已经存在的 v5 KDC，这样，v4 `kinit` 就可以得到一个 v4 的 TGT。依照上一部分的例子，这两个文件应该是这样：

```
/etc/krb.conf:
FOO shag.carpet.net
/etc/krb.realms:
.carpet.net FOO
```

当 v4 请求 `rcmd.hostname@REALM` 的证书时，KDC 用 v5 中与之对应的 `host/hostname@REALM` 代理密钥满足其要求，因此你不必在 v5 KDC 中单独创建“`rcmd/hostname`”代理。因为只支持 v4 的服务器仍然需要使用代理的密钥，所以你得为该密钥创建 v4 版本的密钥文件 (`/etc/srvtab`)；为此，你可以用 v5 中的 `kutil`，读取已有的 `krb5.keytab`，写入 v4 的 `srvtab` 中。直接跨域认证用现有的跨域密钥仍可自动完成；不过 Kerberos-4 不支持层次域结构。

如果使用 Kerberos-5 的证书转换服务 (credentials conversion service)，你甚至不用单独运行 v4 的 `kinit`。此时在 KDC 方面，必须运行一个独立的服务器程序 `krb524b`。接着，执行 v5 `kinit`，然后用户只需要运行 `krb524init`。这样就用 v5 的 TGT 获取了一个 v4 的 TGT，这一点可以用 v4 的 `klist` 命令验证。

注意，OpenSSH 和 SSH1 的 Kerberos 认证不具有互操作性。它们使用的 SSH 协议消息相同，但是都默认应该得到的是对应版本的 Kerberos 压缩密钥。即使用 Kerberos-5 的 v4 兼容方式也无法解决此问题。我们期待着 OpenSSH 最终能支持 Kerberos-5。

还要注意，Kerberos-4 中没有一个与 Kerberos-5 的 *kinit -A* 类似的开关。我们不知道怎样用 Kerberos-4 解决 Kerberos/NAT 问题。[\[11.4.4.2\]](#) 不过我们听说 Transarc AFS KDC 可以忽略证书中的 IP 地址，这样能避免发生此问题。

11.4.5.2 Solaris 中的 Kerberos

Sun Microsystems 的 Solaris 操作系统中集成的 Kerberos-4 功能有限，只能用于特殊目的；它支持 Sun 的 NFS 和安全 RPC Kerberos 认证。就我们所谈及的，这个 Kerberos 还不足以编译或运行支持 Kerberos-4 的 OpenSSH，所以可能得安装其他版本的 Kerberos-4，如，MIT 版。要小心，这样做会引起混乱；例如，Solaris 上的 */bin/kinit* 对 MIT Kerberos-4 的操作不会有任何影响。

11.5 跨网关连接

到目前为止我们一直假设对发出的连接数没有限制，即只要你愿意，就可以发起任意数目的 TCP 连接。在讨论防火墙的时候也假设只限制可接收的连接。在需要更强的安全性（或者仅仅是管理得更严格）的环境中，这样做就不合适了。实际上我们可能根本就没有直接对外的 IP 连接。

现实中，公司通常通过一台代理服务器或网关主机连向外界。网关主机的作用就是同时连接内部和外部的网络。然而，网关并不能起到路由器的作用，两部分网络还是相互隔离的。更确切地说，网关在两个独立的网络之间提供受限的应用程序级访问。

在本例中，我们讨论 SSH 在这类环境下的一些问题：

- 用 *ssh* 提供到外界主机的透明连接。
- 建立 *scp* 连接。
- 使用端口转发运行双层 SSH。

11.5.1 创建 SSH 透明连接

假设你的公司有一台网关主机 G，这是你通向 Internet 的惟一出口。你已经登录到

一台客户主机 C 上，现在想要访问公司网络外的一台服务器 S。如图 11-14 所示。现假设所有这三台机器都已经安装了 SSH。

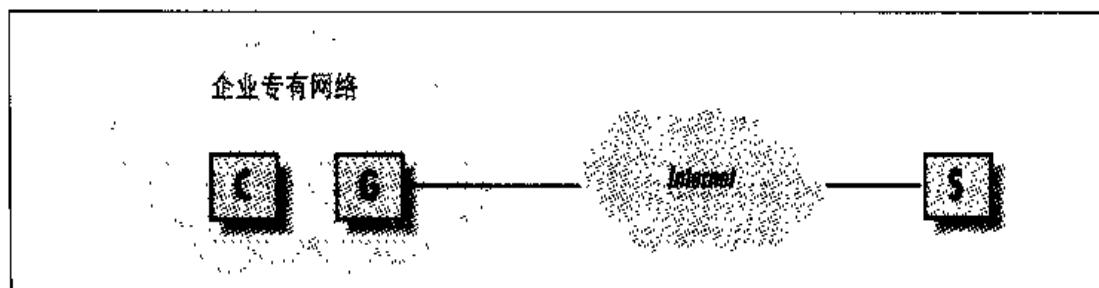


图 11-14：代理网关

从客户端 C 到服务器 S 建立连接的过程分为两步：

1. 建立从 C 到网关 G 的连接：

```
# 在客户 C 上执行  
$ ssh G
```

2. 建立从 G 到服务器 S 的连接：

```
# 在网关 G 上执行  
$ ssh S
```

这种方式当然是可行的，只不过需要在网关上做一个额外的手工操作，而人们并不关心网关上到底发生了什么。使用代理转发和公钥认证机制可以避免在网关 G 上输入口令，即便如此，最好还是让额外的操作透明化。

更糟糕的情况是，从客户端 C 向服务器 S 上执行远程命令的过程无法变得透明。所以通常的方式不是：

```
# 在客户 C 上执行  
$ ssh S /bin/ls
```

而是必须在网关 G 上运行一个远程 ssh，然后连向服务器 S：

```
# 在客户 C 上执行  
$ ssh G "ssh S /bin/ls"
```

这种方式不仅麻烦而且降低了系统的自动化程度。设想一下如果所有基于 ssh 写的脚本都得按照这种操作方式重写，会是什么情况。

幸好 SSH 还有更灵活的设置方式。下面我们将采用 SSH1 的特点和语法介绍。其中，使用公钥认证是为了利用 *authorized_keys* 文件的优势；*ssh-agent* 结合代理转发可以使认证传递过程中的第二次 SSH 连接操作变得透明（见图 11-15）。

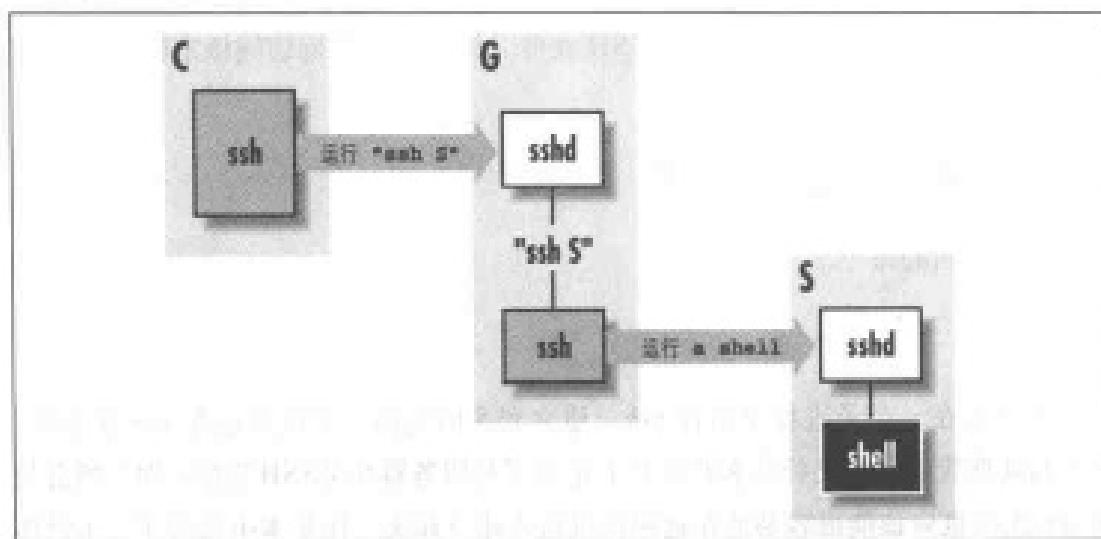


图 11-15：通过代理网关的 SSH 连接链

假设你在网关 G 上的账号是 gilligan，在服务器 S 上的账号是 skipper。首先，建立 SSH 客户端配置文件，把 S 设置为访问网关 G 的昵称：

```

# 客户 C 上的 ~/.ssh/config
host S
  hostname G
  user gilligan
  command="ssh -l skipper S ...key..."

```

现在，在 C 上执行 *ssh S* 命令，将建立到 G 的连接，然后自动运行强制命令，建立到 S 的第二条 SSH 连接。由于利用了代理转发机制，因此只要加载了正确的密钥，就可以自动完成从 G 到 S 的认证过程。这里的密钥既可以与访问 *gilligan@G* 的相同，也可以不同（注 15）。

注 15：注意如果想在交互式连接中使用这一设置，你要在 *ssh* 命令中加入 *-t*，这样可以在 G 上强制指定一个 *tty*。不这样做的原因是无法得知远程命令（在本例中是另一个 *ssh* 实例）是否需要一个 *tty*。

这种技巧不仅可以提供从客户端C到服务器S的透明连接，而且还避免了一个麻烦，即：S对于客户端C来说有可能是一个毫无意义的名字。通常在这类网络环境中，内部网的命名机制与外部是分割开的（比如说用内部分割的DNS）。SSH能让你指定那些你不能访问的主机，其关键的一点是什么呢？是在SSH客户端设置中的HOST关键字，你可以创建昵称S，借此SSH就可以经由G透明地访问该主机。[\[7.1.3.5\]](#)

11.5.2 跨网关使用SCP

回想一下下面的命令：

```
$ scp ... S:file ...
```

它实际上是在一个子进程中运行`ssh`，建立到S的连接，并调用远程`scp`服务器。[\[3.8.1\]](#)既然我们现在已经从客户端C上建立了与服务器S的SSH连接，你自然会想到`scp`连接也应该能很容易地在这两台机器上建立起来。你基本上猜对了，不过还有下面两个小问题需要解决（要没有问题也就不是软件了）：

- 由于使用了强制命令，应该如何调用`ssh`子进程。
- `tty`不可用给认证带来的困难。

11.5.2.1 如何传递远程命令

第一个问题是，客户端C上的`ssh`命令发出一条命令，它在服务器S上执行，用于启动`scp`服务；不过为了使用强制命令，这条命令现在被忽略了。必须找到合适的方式把启动`scp`服务的命令传给S。为此，需要修改网关G上的`authorized_keys`文件，告知`ssh`调用环境变量`SSH_ORIGINAL_COMMAND`中包含的命令：[\[8.2.4.4\]](#)

```
# 网关G上的~/.ssh/authorized_keys
command="ssh -l skipper S $SSH_ORIGINAL_COMMAND" ...key...
```

这样强制命令就可以正确调用服务器上与`scp`有关的命令了。不过事情还没完，因为这条强制命令破坏了现有设置。在C上通过`ssh`运行远程命令（比如，`ssh S /bin/ls`）是可以的，不过如果单独运行`ssh S`启动远程shell就会出错。这是由于只有指明了远程命令之后，才会设置`SSH_ORIGINAL_COMMAND`的值，所以在`SSH_ORIGINAL_COMMAND`尚未定义时，执行`ssh S`将会死机。

可以用 Bourne shell 及其参数替换操作符 :- 解决此问题。如下：

```
# 网关 G 上的 ~/.ssh/authorized_keys
command="sh -c 'ssh -l skipper S ${SSH_ORIGINAL_COMMAND:-}" ...key...
```

如果 \$SSH_ORIGINAL_COMMAND 的值已设置，表达式 \${SSH_ORIGINAL_COMMAND:-} 就返回这个值，否则返回空字符串。（一般说来，\${V:-D} 的意思是“返回环境变量 V 的值；如果其值没有设置就返回 D”。更多内容请参看 sh 手册页）这样就得到了预期的结果，ssh 和 scp 命令都可以从 C 向 S 正确执行。

11.5.2.2 认证

第二个与 scp 有关的问题是第二条 SSH 连接（从网关 G 到服务器 S）的认证问题。你没法把密码或者口令传递给第二个 ssh 命令，因为没有给它分配 tty（注 16）。所以你需要一种不需要用户输入的认证形式：RhostsRSA，或者公钥加代理转发。RhostsRSA 可以完成这项工作，所以如果你打算使用 RhostsRSA，就可以跳过这里的介绍，直接看下一部分。对于公钥认证来说，有一个问题需要解决：scp 用了一个 -a 开关把代理转发关掉了，[6.3.5.3] 你必须重新打开代理转发才能正常工作。不过这可需要惊人的技巧。

通常可以在客户设置文件中打开代理转发：

```
# 客户 G 上的 ~/.ssh/config，但失败了
ForwardAgent yes
```

这样做无济于事，因为命令行 -a 开关的优先级更高。你可能转而尝试 scp 命令的 -o 选项，因为它可以把诸如 -o ForwardAgent yes 之类的选项传递给 ssh。不过在这种情况下，scp 把 -a 放置在所有不如其优先级高的 -o 选项之后，因此也不行。

不过问题还是可以解决的。scp 命令的 -S 选项可以向 SSH 客户端指示其应使用的路径；这样你可以创建一个“封装（wrapper）”脚本，把需要执行的 SSH 命令嵌入其中，然后在调用 scp 时使用 -S。比方说，把下面的脚本写入客户端 C 上的某个可执行文件如 ~/bin/ssh-wrapper 中：

```
#!/usr/bin/perl
exec '/usr/local/bin/ssh1', map {$_ eq '-a' ? () : $_} @ARGV;
```

注 16：事实上你可以随意尝试一下，不过结果会很可怕，我们也不打算深入讨论。

这将调用真正的`ssh`, 如果有`-a`选项就把它从命令行中去掉。现在可以这样使用`scp`命令:

```
scp -S ~/bin/ssh-wrapper ... $file ...
```

问题就解决了。

11.5.3 独辟蹊径：双层 SSH（端口转发）

如果不用强制命令，也可以有另外一种方式实现跨网关SSH连接：从客户端C到SSH服务器S进行端口转发，先建立从C到G的SSH会话，然后在此之上运行第二个SSH会话（见图11-16）。



图11-16：通过代理网关转发SSH连接

由于`ssh -L`命令要求其后不再配置端口号`-p`，不能像刚才这样：

```
# 在客户C上执行
$ ssh -L2001:localhost:22 G
从G“(sshserv) 转发”一个端口从22端口到
# 在客户C的另一不同shell中执行
$ ssh -p 2001 localhost
```

这个C到S的过程是在第一个(C到G)连接的一个端口转发通道中携带了第二个连接。如果在客户端配置文件中创建昵称S，这一过程会变得更加透明：

```
# 客户C上的 ~/.ssh/config
host S
  hostname localhost
  port 2001
```

现在前一个命令应该改成这样：

```
# 在客户C上执行
$ ssh -L2001:S:22 G

# 在客户C的另一不同 shell 中执行
$ ssh S
```

因为这一技术需要通过执行额外的一步手工操作来实现端口转发，所以比[11.5.1]中提到的技术透明性低。不过这也有一些优点。在第一种方法之上使用端口转发或者X转发会比较复杂。*scp*不仅用`-a`开关关掉了*ssh*的代理转发，而且还用`-o "ClearAllForwardings yes"`关掉了包括端口转发在内的所有转发机制。所以你还需要在前面那个封装脚本中去掉所有不想要的选项。[11.5.2.2]然后，如果要用端口转发的话，还要建立一个接一个的端口转发链。例如，下面的命令将客户端C的2017端口转发到服务器S的143端口（IMAP端口）：

```
# 客户C上的 ~/.ssh/config
host S
  hostname G
  user gilligan

# 网关G上的 ~/.ssh/authorized_keys
command="ssh -L1234:localhost:143 skipper@S" ...key...

# 在客户C上执行
$ ssh -L2017:localhost:1234 S
```

这种方式能达到目的，却难于理解，不仅易出错，还很脆弱：如果引发了TIME_WAIT问题，[9.2.9.1]你就得修改这些文件，重新启动这条隧道，选择一个新的端口代替1234，而这个新端口的寿命也同样短暂。

双层SSH是替换这种方式的一种很好的选择。端口转发和X转发能以通常的方式直接在C和S之间运作，前面的例子就变成这样：

```
# 客户C上的 ~/.ssh/config
host S
  hostname localhost
  port 2001
```

```
# 在客户C上执行  
$ ssh -L2001:S:22 G  
  
# 在客户C的另一不同shell中执行  
$ ssh -L2017:localhost:143 S
```

最后一条命令建立到 S 的连接，将本地端口 2017 转发至 S 的 IMAP 端口。

11.5.4 安全性比较

刚才讨论的两种方法在安全性方面有所不同。我们再次强调：这里假设的网络环境是如前所述的 C、G 和 S 三台计算机。

11.5.4.1 “中间服务器” 攻击

在第一种方法中，两个 SSH 连接串成一条链。这种连接方式的弱点在于，如果中间的一台 SSH 服务器（在网关 G 上）已经被攻破，那么会话信息就暴露了。从 C 来的数据在 G 上解密，传至第二个 SSH 客户端（也在 G 中），然后该客户端将信息重新加密，再传到服务器 S。所以在 G 上可能得到会话的明文，而一台已被攻破的主机能够任意读取和替换这些信息。

第二种端口转发方法不会有这种问题。实际上 G 上的 SSH 服务器在从 C 转发至 S 的 SSH 连接中并不占什么特殊地位。任何读取或替换会话内容的企图都将会失败，这和网络嗅探器或者任何活动网络攻击失败的道理是一样的。

11.5.4.2 服务器认证

另一方面，如果用 SSH1 或者 OpenSSH 实现端口转发，就不如连接链安全，因为这两者缺少对服务器认证的支持。原因在于，当服务器地址是 127.0.0.1（本机）时，SSH 和 OpenSSH 将会进行某种特别的操作：强制接受主机密钥，而忽略实际密钥。更确切地说，是忽略了在知名主机列表中检查主机密钥的操作，认为服务器提供的主机密钥总是与表中的 localhost 相联系的。

SSH1 设计成这样是为了使用方便。如果某用户的根目录在多台计算机之间共享，那么每台机器上的 SSH 客户端看到的每个用户的已知名主机地址列表将是同一个文件。但是“localhost”这个名字是比较特殊的，在每台机器上表示的意思不一样：都

是表示本机。所以，如果该用户在多台机器上运行 `ssh localhost` 命令，就会一直被提示说主机密钥已经改变，而实际情况并非如此。已知名主机列表文件总是把“本机”映射成最后执行 `ssh` 命令的主机密钥，而不是当前活动的主机。

所以这里的问题在于，既然远程主机的IP地址实际上就是本地主机，那么忽略掉服务器认证的过程就可以提高效率，从而也导致这种方式容易受中间人或者伪装服务器攻击。

SSH2对本机没有这项特殊操作，因此不会有这个弱点。SSH2的已知名地址列表设计得更细致，其映射机制不是从主机到密钥，而是服务器套接字（[主机，端口]对）到密钥。这就意味着可以对每一个本地转发端口都使用不同的密钥。当你第一次用SSH2建立从C的转发端口2001至S的连接时，你不仅是接受了服务器的主机密钥，这样做还绕过了对第一个连接的认证。在建立第一个连接之前，你应该将S的主机密钥拷贝至C的`~/.ssh2/hostkeys/key_2001_localhost.pub`文件中。这就在S的主机密钥和套接字（`localhost,2001`）之间建立了联系，最初的转发连接也就能得到正确的服务认证。

第十二章

疑难解答及 FAQ

本章内容：

- 第一道防线：调试消息
- 疑难问题及解答
- 其他 SSH 资源
- 错误报告

SSH1、SSH2 和 OpenSSH 都是非常复杂的产品。如果出现错误的话，应按照如下的步骤解决：

1. 在调试模式下启动客户端及服务器。
2. 参考联机帮助，看看别人是不是已经遇到并解决过同一问题。
3. 寻求他人帮助。

很多人都直接跳到第 3 步，在公共论坛上把问题提出来，然后等上几个小时甚至几天，看是否有人回答；但实际上，只要一个简单的 `ssh -v` 或者 FAQ 就能很快解决问题。所以请做一个聪明、高效的技术人员吧，在求助别人之前先尝试利用手头已有的资源。（当然了，如果你已经尽了自己的全力却还没有解决，SSH 社区会很乐意帮忙的。）

12.1 第一道防线：调试消息

SSH1/SSH2 客户端及服务器都有内嵌的调试机制。通过适当的方式调用，我们就能获得程序执行过程及错误的提示消息。这些消息可用于排错。

12.1.1 客户端调试

多数客户端在使用 `-v` (详细模式) 启动之后都能打印调试消息: [7.4.15]

```
$ ssh -v server.example.com
$ scp -v myfile server.example.com:otherfile
```

在详细模式下能发现很多问题。因此无论何时, 你碰到问题之后的本能反应都应该是启动详细模式。

注意: 请深呼吸, 跟我们一起说:

“`ssh -v` 是我的好朋友……”

“`ssh -v` 是我的好朋友……”

“`ssh -v` 是我的好朋友……”

12.1.2 服务器调试

SSH1、SSH2 和 OpenSSH 服务器也能根据请求打印调试消息:

```
# SSH1, OpenSSH
$ sshd -v

# 仅对 SSH2
$ sshd2 -d
```

每种情况下服务器都进入特殊的调试模式: 它仅接受一个连接请求, 正常执行操作, 连接终止后退出。服务器不会转入后台, 也不创建子进程处理该连接, 只在进程执行过程中在屏幕 (即标准错误流设备) 上打印信息。

SSH2 的调试系统更加复杂: 它有调试级别数, 可以用 `-d` 选项指定, 数字越高表明提示信息越多。[5.8.2] 实际上 `-v` 进入的详细模式就是 `-d2` 的缩写。在更高的级别上, 调试信息非常之多, 因此只有 SSH 开发人员要跟踪很晦涩的问题时才会采用。不过你也可能需要转到 2 以上的层次上查看一些必要的信息。例如, `-d3` 可得到双方协商使用的连接算法; 若看到 “TCP/IP Failure”, 可转到 `-d5`, 获取连接过程中返回的更确切的操作系统级出错消息。

SSH 十大问题

问：第一次怎么在远程主机上安装我的公钥？

答：用密码认证方式连接服务器，在本地终端上用拷贝粘贴功能即可实现。
[12.2.2.4]

问：我已经把 SSH 密钥文件 mykey.pub 放在我的远程 SSH 目录中了，不过公钥认证还是无法工作？

答：必须在远程认证文件中指定公钥。[12.2.2.4]

问：公钥认证不工作怎么办？

答：试试 `ssh -v`，并检查密钥、文件及权限。

问：密码认证无法工作怎么办？

答：使用 `ssh -v`，引起这种情况的可能性很多。[12.2.2.2]

问：可信主机认证不能工作(SSH1 RhostsRSA, SSH2 hostbased)怎么办？

答：`ssh -v` 并检查四个控制文件、主机名、SSH 客户端程序或 `ssh-signer2` 的 setuid 状态。[12.2.2.3]

问：认证时如何才能不用输入密码或口令？

答：`ssh-agent`、非加密密钥、可信主机认证，或 Kerberos。[12.2.2.1]

问：如何用端口转发实现安全 FTP？

答：对控制连接而言，可将某个本地端口转发到 FTP 服务器的 21 端口上；数据连接则复杂得多。[12.2.5.6]

问：X 转发无法正常工作怎么办？

答：不要手工设置远程 DISPLAY 变量。（还要检查其他一些东西。）[12.2.5.6]

问：为什么 scp 命令行中不能用通配符或 shell 变量？

答：在 `scp` 运行之前本地 shell 就对这些字符进行了扩展。它将特殊字符转义了。[12.2.5.4]

问：我肯定用法没错，为什么 ssh 或 scp 的某项功能用不了？

答：`ssh -v` 也可能是系统配置将你的设置覆盖了。

当调试某个服务器时，切记要避免与其他处于正在运行的 SSH 服务器发生端口冲突。要么关闭其他服务器，要么换用其他的端口进行调试：

```
$ sshd -d -p 54321
```

客户端使用 *-p* 选项测试处于调试状态的服务器实例：

```
$ ssh -p 54321 localhost
```

这样，就不会中断或影响其他正在使用的 *sshd*。

12.2 疑难问题及解答

这一部分将分类叙述一些疑难问题，覆盖的内容比较广泛。前面列出的“SSH 十大问题”是根据我们的经验总结的FAQ中最常问到的问题。下面我们将着眼那些在不同操作系统上的SSH不同版本中都可能出现的问题。而不会介绍下列很快就将不复存在的问题：

- 某一特定操作系统的编译问题，如“HyperLinux beta 0.98 版需要 *--with-woozle* 标志”。
- 只在SSH1或SSH2的某个特定版本(尤其是较老的版本)中出现的疑问或bug。

这些问题的最好解答都在 SSH FAQ 之中，[12.3.1] 也可以与其他 SSH 使用者讨论解决。

以下问题中，我们假设你已经开始用调试或详细模式(即 *ssh -v*)解决所有的问题。(如果还没有，那就应该马上启动！)

12.2.1 一般问题

问： *ssh*、*scp*、*ssh-agent*、*ssh-keygen* 等命令的执行结果看起来和我想像的不一样。显示的帮助信息就更加奇怪了。这是怎么回事？

答： 可能你错把 SSH2 程序当成 SSH1 的了，反之亦然。找到可执行程序所在的位置，运行 *ls -l* 命令。如果是一般文件，就可能是 SSH1 或 OpenSSH 的。如果

显示的是符号链接，则应检查到底指向 SSH1 还是 SSH2。（SSH2 文件名都以“2”结尾。）

问：连接 SSH 服务器时返回信息为“Connection refused”，为什么？

答：连接时 SSH 服务器尚未启动。再检查一遍主机名及 TCP 端口号，可能服务器没有在缺省端口上运行。

问：登录时日期信息（/etc/motd）显示两遍，为什么？

答：*sshd* 及 *login* 程序都打印日期。关闭 *sshd* 打印功能的方法是在服务器范围配置中使用 PrintMotd 关键字。

问：登录时两次显示“No mail”或“You have mail”之类的消息。

答：*sshd* 和 *login* 程序都检查邮件。关闭 *sshd* 邮件检查功能的服务器范围配置关键字为 CheckMail no。

12.2.2 认证问题

12.2.2.1 一般认证问题

问：SSH1 服务器说“Permission denied”，然后就退出了，为什么？

答：所有认证方式都失败后就会发生这种现象。请使用调试模式运行客户端，读一读调试信息，从中寻找线索。也可以阅读本节后面叙述的特定认证问题及其解决办法。

问：不输入密码或口令怎么认证？

答：可以使用四种支持不用输入密码或口令的认证方式：

- 公钥认证结合 *ssh-agent*。
- 公钥认证结合磁盘存储非加密密钥（口令为空）。
- 可信主机认证。
- Kerberos（仅适用于 SSH1 和 OpenSSH）。

选择之前首先要考虑自动认证的若干重要问题。[11.1]

问：密码和口令已经确认，但我还没来得及做进一步操作，SSH服务器就把连接关闭了，为什么？

答：服务器的空闲超时时间设得过短。如果你是服务器主机的系统管理员，就在服务器范围配置文件[5.4.3.3]中将 `IdleTimeout` 设得长一些。如果你是 SSH1 或 OpenSSH 的终端用户，就在 `authorized_keys` 文件中设置一个超时值。[8.2.7]

问：`RequiredAuthentications` 不起作用，为什么？

答：SSH2 2.0.13 中此功能有问题，认证总会失败。SSH 2.1.0 中修改了这个错误。

问：`SilentDeny` 好像对所有的认证方式都无效，为什么？

答：`SilentDeny` 与认证无关，它只能控制使用 `AllowHosts` 和 `DenyHosts` 时的访问。如果连接被 `AllowHosts` 或 `DenyHosts` 的某个值拒绝，那么 `SilentDeny` 就可以控制客户端是否能看到认证失败的提示信息。

12.2.2.2 密码认证

问：口令认证无法正常工作怎么办？

答：`ssh -v`。如果所有的连接都遭拒绝，那很有可能是服务器没有运行，或者是连接的端口号不对。缺省端口为 22，但远程主机的系统管理员可能会更改使用的端口。如果返回消息为“permission denied”，那么服务器很可能关闭了密码认证。

确保服务器在其配置文件中启用密码认证（SSH1 和 OpenSSH 中是“`PasswordAuthentication yes`”，SSH2 中为“`AllowedAuthentications password`”），同时还要确认客户端配置文件中没有“`PasswordAuthentication no`”。

如果服务器让你输入密码，但是你输入的密码被拒绝，那有可能你连接的账号不对。看看本地用户名与远程用户名是否一致。如果不一致，登录时就必须指定：

```
$ ssh -l my_remote_username server.example.com  
$ scp myfile my_remote_username@server.example.com:
```

如果还不行，就再检查本地客户端配置文件（`~/.ssh/config` 或 `~/.ssh2/ssh2_config`），看看 `User` 关键字的值是否正确。特别是如果配置文件中存在包

含通配符的Host关键字，就得返回去看那条无效的命令与Host是否正确匹配。

[7.1.3.4]

有一个服务器端的常见问题和OpenSSH 及可插入式认证模块（Pluggable Authentication Modules, PAM）的配置有关。PAM是一个与应用程序无关的通用认证、授权、记帐系统。如果操作系统支持PAM（比如，Linux 和 HPUX），OpenSSH 可能在编译过程中自动选择使用PAM。但除非明确地配置PAM，令其支持SSH，否则所有密码认证操作都将莫名其妙地失败。所谓配置，只不过是将恰当的*sshd.pam*文件从OpenSSH的*contrib*目录拷贝至PAM主目录（通常为*/etc/pam.d*），并改名为“*sshd*”。*contrib*目录中包含若干不同版本 Unix 的例子。例如，在RedHat Linux 系统中：

```
# cp contrib/redhat/sshd.pam /etc/pam.d/sshd  
# chown root.root /etc/pam.d/sshd  
# chmod 644 /etc/pam.d/sshd
```

如果OpenSSH 没有使用PAM，密码认证却还不能工作，那可能与编译开关--with-md5-passwords 或 --without-shadow 有关。如果OpenSSH 支持PAM，那么这些选项就不会起作用，因为它们是决定OpenSSH 如何读取 Unix *passwd*映射文件的。有PAM时OpenSSH 不会直接读取 *passwd*文件。如果没有PAM，同时你的系统用的不是传统的*crypt*(DES) 散列加密密码，而是MD5 散列，那就必须用--with-md5-passwords。可以从*/etc/passwd*及*/etc/shadow*文件中看出系统用的是哪一种散列。经过散列的密码在每一组的第二项上；如果*/etc/passwd*的密码字段里只写了“x”，那就是说真正的内容在*/etc/shadow*中。MD5 散列的结果很长，其中包含的字符种类也多得多：

```
# /etc/shadow, MD5 散列  
test:$1$teMXcnZB$DEZbQXJzUz4g2J4qYkRh.:...  
  
# /etc/shadow, crypt 散列  
test:JGQfZ8DeroV22:...
```

最后一点，如果你怀疑OpenSSH 会访问*/etc/shadow*密码文件，可以试试--without-shadow，但你的系统不会使用它。

问：服务器不允许用空密码怎么办？

答：空密码不安全，应该避免使用。如果非用不可，可在服务器范围配置文件中设置“PermitEmptyPasswords yes”。[5.6.3]

12.2.2.3 可信主机认证

问：可信主机认证无法工作（SSH1 的 RhostsRSA 和 SSH2 基于主机的认证），为什么？

答：使用 `ssh -v`。如果看起来一切正常，则按照下面的步骤检查。在此我们假设客户端上的账号为 `orpheus@earth`，目标账号为 `orpheus@hades`，即，用户 `orpheus` 从主机 `earth` 上调用 `ssh hades`。

对于 SSH1 及 OpenSSH1：

- SSH 客户端程序必须 setuid 成 root。
- 服务器及客户端配置文件中包含“RhostsRSAAuthentication yes”。
- 客户端的公用主机密钥必须在服务器的已知名主机列表中。在本例中，`hades:/etc/ssh_known_hosts` 中必须包含能将主机名“`earth`”与该主机的公钥关联起来的一项内容。例如：

```
earth 1024 37 71641647885140363140390131934...
```
- 该项内容可能在目标账号的已知名主机列表文件中，比如 `hades:~orpheus/.ssh/known_hosts`。注意，`earth` 应该是服务器上能识别的该客户端主机的正规网络名称。也就是说，如果 SSH 连接是从 192.168.0.1 发起的，那么在 `hades` 上执行 `gethostbyname(192.168.0.1)` 的返回值就应该是“`earth`”，而不是该主机的昵称或别名（例如，如果主机名为 `river.earth.net`，那么查找结果就不仅仅是“`river`”）。这其实牵扯到多种名字服务，因为 `gethostbyname` 允许设置成根据多个来源决定地址翻译结果（比如 DNS，NIS，`/etc/hosts`）。详情需参看 `/etc/nsswitch.config`。如果系统不支持正规主机名，那使用 RhostsRSA 时就会碰到无穷无尽的麻烦。使用在已知名主机列表中增加额外的主机昵称的方式，可在某种程度上缓解这类问题，就像这样：

```
earth,gaia,terra 1024 37 71641647885140363140390131934...
```

- 设置 `hades:/etc/shosts.equiv` 或 `hades:~orpheus/shosts`，允许该用户登录。在 `shosts.equiv` 中加入 `earth` 的结果是，`earth` 上的所有非 root 用户都能访问 `hades` 上的相同名字的账号。在 `.shosts` 中增加 `earth`，意味着 `orpheus@earth` 能访问 `orpheus@hades`。

- 某些防火墙软件能拒绝特权端口向外发出连接请求。在这种情况下，RhostsRSA 认证就无法进行了，因为它是基于特权源端口的。*ssh -P* 命令可使 SSH 连接经由非特权端口建立，不过这样就不得不换用其他的认证方式。

对于 SSH2：

- 在服务器及客户端上设置“*AllowedAuthentications hostbased*”。
- *ssh2* 不需要 setuid 成 root，但 *ssh-signer2* 需要。更确切的说，后者得能读取私有主机密钥的内容，而这在一般的环境中就意味着必须将其 setuid 成 root。
- 将 earth 的公钥拷贝至 *hades:/etc/ssh2/knownhosts/earth.ssh-dss.pub* 中。（如果在服务器端指定了“*UserKnownHosts yes*”，也可以将其拷贝到 *hades:~orpheus/.ssh2/knownhosts/earth.ssh-dss.pub*。）
- 考虑正规主机名的问题。请参看有关 RhostsRSA 的讨论。

对于 OpenSSH/2

- 服务器与客户端上同时设置“*DSSAuthentication yes*”。
- *ssh* 必须 setuid 成 root（否则得要有权读取 */ect/ssh_host_dsa_key* 中的客户端主机私钥。*ssh* 并不要求一定使用特权端口）。
- 将 earth 的公用主机密钥拷贝至 *hades:/etc/ssh_known_hosts2* 或者 *hades:~orpheus/.ssh/known_hosts2* 中。
- 考虑正规主机名问题。请参看有关 RhostsRSA 的讨论。

12.2.2.4 公钥认证

问：第一次怎么在远程主机上安装我的公钥文件？

答：一般方法为：

- a. 生成密钥对。
- b. 将公钥文本拷贝到剪贴板或其他有剪切 / 粘贴功能的缓冲区中。
- c. 用密码认证方式建立到远程主机的 SSH 连接，这种认证方法不需要远程账号上具有任何特定文件。

- d. 在远程主机上编辑适当的认证及密钥文件。
 - 在 SSH1 和 OpenSSH/1 中：将公钥加入 `~/.ssh/authorized_keys`。
 - 在 OpenSSH/2 中：将公钥加入 `~/.ssh/authorized_keys2`。
 - 在 SSH2 中：将公钥粘贴到 `~/.ssh2` 下面一个新的 `.pub` 文件中（如，`newkey.pub`），并在 `~/.ssh2/authorization` 中增加一行“Key newkey.pub”。
- e. 从远程主机上退出登录。
- f. 重新用公钥认证方式登录远程主机。

在编辑远程的认证文件时，必须确保所使用的文本编辑器没在公钥的中间插入行结束符。SSH1 与 OpenSSH 的公钥非常之长，一定得存在单独一行里。

问：我已经把 SSH 公钥文件 `mykey.pub` 放在我的远程 SSH 目录中了，但是公钥认证还是无法工作，为什么？

答：只把公钥文件（例如，`mykey.pub`）放在 SSH 目录中是不够的。对于 SSH1 和 OpenSSH 而言，必须把密钥加到 `~/.ssh/authorized_keys` 中。若是 OpenSSH/2，密钥则应该加入到 `~/.ssh/authorized_keys2`。而对于 SSH2，必须在 `~/.ssh2/authorization` 中加入一行 `Key mykey.pub`。

问：无法使用公钥认证怎么办？

答：以调试模式启动客户端 (`ssh -v`)，并确认：

- 本地客户端使用的认证文件正确。
- 远程主机恰当的位置上存在正确的公钥。
- 远程主目录、SSH 目录以及其他与 SSH 有关的文件的访问权限正确。

问：登录过程中没有要求我输入密钥的口令，而是要我输入登录密码，或者，我的连接请求被拒绝，出错信息为：“进一步的认证方法不可用”(SSH2)，为什么？

答：造成这两种情况的原因有：

- 客户端与服务器必须都能用公钥认证 (SSH1/OpenSSH：“RSAAuthentication yes”；SSH2：“AllowedAuthentications publickey”)。
- 在远程用户名与本地用户名不同的情况下，用 `-l` (小写 L) 选项指定远程用户名。否则 SSH 服务器检查的用户名就是错误的。例如：

```
$ ssh -l jones server.example.com
```

- 检查服务器上账号的文件访问权限。如果某个文件或目录的所有者或访问权限设置的不对，那么 SSH 服务器就会拒绝进行公钥认证。这是提高安全性的一项功能。在详细模式下运行 *ssh* 即可揭示这一情况。

```
$ ssh -v server.example.com  
...  
server.example.com: Remote: Bad file modes for /u/smith/.ssh
```

要确保你在服务器上的账号是下列文件和目录的所有者，并且这些文件和目录不是全局可写的：`~`、`~/ssh`、`~/ssh/authorized_keys`、`~/ssh2`、`~/.rhosts` 和 `~/.shosts`。

- 对于 SSH2，如果用 `-i` 选项指定身份标识文件：

```
$ ssh2 -i my-identity server.example.com
```

就得检查一下，确保 *my-identity* 是身份标识文件，而不是私钥文件。（在 SSH1 与 OpenSSH 中情况正好相反，`-i` 选项要求指定一个私钥文件。）请记住，SSH2 身份标识文件是一些包含私钥名的文本文件。

问：要求输入口令的密钥和我用的不是一个，怎么办？

答：确认一下你希望使用的公钥是否在 SSH 服务器上的认证文件中。

检查 SSH 代理是否有问题。是否在代理运行的情况下用 `ssh -i` 或 `IdentityFile` 指定了另一个密钥？代理不能同时使用 `-i` 选项和 `IdentityFile` 关键字。请关掉代理再试一次。

在 SSH1 与 OpenSSH 中，如果 `~/ssh/authorized_keys` 中指定了任何选项，就得检查一下是否有排版错误。如果选项输错了，那么服务器就会忽略相关的密钥行。请记住，选项之间的分隔符是逗号，而不是空格。

12.2.2.5 PGP 密钥认证

问：PGP 口令确认之后，又要我输入登录密码，为什么？

答：如果输入 PGP 密钥的口令之后又要输入密码：

```
Passphrase for pgp key "mykey": *****  
smith's password:
```

而你认为输入的口令确实是正确的，那么首先要看看口令是否真的正确。（比

方，用公钥加密一个文件，再解密。）如果正确，那么有可能是客户端与服务器上的 PGP 不兼容。我们见过在客户端生成的 PGP 密钥比服务器要求的密钥位数少时出现这种情况。在这种情况下，得从服务器上生成新密钥。

问：返回信息 “Invalid pgp key id number ‘0276C297’ 是怎么回事？

答：也许你忘记在密钥 ID 前加上 0x 前缀，这样 SSH 服务器就只能把十六进制数字当成十进制数字解释了。试试 `PgpKeyId 0x0276C297`。

12.2.3 密钥和代理的问题

12.2.3.1 一般密钥 / 代理问题

问：我用 SSH1 生成了一个密钥，想在其他 SSH1 客户端（例如，NiftyTelnet SSH、F-Secure SSH Client 或者 SecureCRT）中使用，但是为什么这些客户端程序说密钥格式不对？

答：首先，看看你生成密钥时用的是 `ssh-keygen1` 还是 `ssh-keygen2`。SSH1 和 SSH2 的密钥不兼容。

其次，看看你传送密钥时使用的文件传输程序是否恰当。如果用 FTP，那么传送私钥时应该用二进制方式，否则传输程序会在其中加入一些垃圾字符。公钥应该使用 ASCII 方式传输。

问：我生成了一个 SSH1 密钥，想在 SSH2 中使用这个密钥文件。但是不行（反之亦然），为什么？

答：这很正常，SSH1 与 SSH2 的密钥不兼容。

问：我用 `-i` 或 `IdentityFile` 手工指定了密钥，但为什么似乎根本没用到？

答：你运行代理了么？如果运行了，`-i` 和 `IdentityFile` 就不起作用了。代理中使用的第一个密钥的优先权最高。

12.2.3.2 ssh-keygen

问：为什么每次运行 `ssh-keygen` 都把我缺省的身份标识文件覆盖了？

答：告知 `ssh-keygen` 将生成的密钥输出到别的文件中。SSH1 和 OpenSSH 中为 `-f`

选项。*ssh-keygen2* 中可以在命令行的最后一个参数中指定文件名，不需要选项。

问：我能不能不要用重新生成密钥就改变口令？

答：可以。SSH1 与 OpenSSH 中可以用 *-N* 选项，*ssh-keygen2* 中使用 *-p* 选项。

问：如何生成主机密钥？

答：生成空口令密钥，并将其安装在适当的位置：

```
# SSH1, OpenSSH  
$ ssh-keygen -N '' -b 1024 -f /etc/ssh_host_key  
  
# 仅对 SSH2  
$ ssh-keygen2 -P -b 1024 /etc/ssh2/hostkey
```

问：为什么生成密钥花的时间很长？

答：这是有可能的，这要看你的 CPU 速度及所要生成的密钥位数。DSA 密钥花的时间一般比 RSA 密钥长。

问：我应该生成多少位的密钥？

答：为了增强安全性，我们推荐密钥长度至少应该有 1024 位。

问：*ssh-keygen2* 输出的 oOo.oOo.oOo.oOo 是什么意思？

答：手册中称此为“进度指示”。我们猜测这是正弦函数波形的 ASCII 码表示，要么就是大猩猩滴滴咕咕的声音。可用 *-q* 标志将其屏蔽。

12.2.3.3 ssh-agent 及 ssh-add

问：我退出登录之后 *ssh-agent* 还在运行，为什么？

答：在用单 shell 方式启动代理的情况下，这种现象是正常的。此时必须自己终止代理，要么手工终止 (*bleah*)，要么在 shell 配置文件中加入特定指令。[6.3.2.1] 如果使用子 shell 方式，那么在退出登录时（实际上是关闭子 shell 时），代理就会自动终止。[6.3.2.2]

问：调用 *ssh-add* 输入口令时，为什么看到错误信息“Could not open a connection to your authentication agent.”？

答：按下列步骤调试：

a. 确保 *ssh-agent* 已在运行:

```
$ /usr/bin/ps -ef | grep ssh-agent  
smith 22719      1  0 23:34:44 ?          0:00 ssh-agent
```

否则, 必须在运行 *ssh-add* 之前运行一个代理。

b. 检查代理的环境变量是否已经设置:

```
$ env | grep SSH  
SSH_AUTH_SOCK=/tmp/ssh-barrett/ssh-22719-agent  
SSH_AGENT_PID=22720
```

如果没有设置, 那可能是你运行 *ssh-agent* 的方法不对, 比如:

```
# 错误!  
$ ssh-agent
```

在单 shell 模式下, 必须用 *eval* 命令调用 *ssh-agent*, 并用 ``'' 将其括起来:

```
$ eval `ssh-agent`
```

在子 shell 方式下, 必须指示 *ssh-agent* 激活一个 shell:

```
$ ssh-agent $SHELL
```

c. 确认代理指向的套接字是否有效:

```
$ ls -lF $SSH_AUTH_SOCK  
prwx----- 1 smith 0 May 14 23:37 /tmp/ssh-smith/ssh-22719-agent|
```

如果套接字无效, *SSH_AUTH_SOCK* 变量也许是指向的是前一次调用 *ssh-agent* 时使用的套接字, 这是用户的错误引起的。只能终止代理, 并正确的方式重新启动。

12.2.3.4 每账号认证文件

问: 服务器端的每账号配置不起作用, 怎么办?

答: 按下列步骤检查:

- 你可能搞不清楚哪个 SSH 版本应该用什么文件:
 - SSH1, OpenSSH/1: *~/.ssh/authorized_keys*
 - SSH2: *~/.ssh2/authorization*
 - OpenSSH/2: *~/.ssh/authorized_keys2* (注意, 这里的目录不是 *~/.ssh2*)

- 牢记：*authorized_keys* 和 *authorized_keys2* 文件中包含密钥，而 SSH2 的 *authorization* 文件中包含指向其他密钥文件的标识。
- 其中一个文件中可能出现排版错误。检查一下选项的拼写，再看看 SSH1 的 *authorized_keys* 中选项是否由逗号分隔开（不是空格）。例如：

```
# 正确  
no-x11-forwarding,no-pty 1024 35 8697511247987525784866526224505...  
# 不正确(将无提示失败)  
no-x11-forwarding no-pty 1024 35 8697511247987525784866526224505...  
# 也不正确(注意"no-x11-forwarding,"后的多余空格)  
no-x11-forwarding, no-pty 1024 35 8697511247987525784866526224505...
```

12.2.4 服务器问题

12.2.4.1 sshd_config、sshd2_config

问：如何能让 *sshd* 识别新的配置文件？

答：关闭并重新启动 *sshd*；还有更快的办法：向 *sshd* 发送 SIGHUP 信号，命令为 *kill -HUP*。

问：我更改了 *sshd* 配置文件，也向服务器发送了 SIGHUP 信号。但是没有得到预期结果，为什么？

答：*sshd* 启动命令行方式中的参数可能覆盖这一关键字。命令行选项都是强制生效的，优先权比配置文件关键字高。这时只能关闭 *sshd* 并重新启动了。

12.2.5 客户端问题

12.2.5.1 一般客户端问题

问：*ssh* 或 *scp* 的某项功能不能用了，不过我可以肯定我的使用方式没错，为什么？

答：可能系统管理员在编译 SSH 软件（第四章）或服务器范围配置（第五章）的时候禁用了这项功能。检查编译标志比较容易，服务器范围配置只能从 */etc/sshd_config* (SSH1, OpenSSH) 或 */etc/ssh2/sshd2_config* (SSH2) 里找了。请向系统管理员寻求帮助。

12.2.5.2 客户端配置文件

问： ssh 或 scp 使用了一些我没有请求的功能，做了一些我没想到的事情，为什么？

答： 这可能是对你客户端配置文件中的某些关键字作出的响应。[7.1.3]还有，如果命令行中指定的远程主机名可以和多个 Host 行匹配，那么配置文件中生效的段也就不止一个。

问： ssh/config 文件工作不正常，为什么？

答： 请记住， config 文件中首次出现 Host 指示符之后，所有的语句都包含在某个 Host 段中。 ssh1 手册页建议将正确的缺省配置放在 config 文件的最后； ssh1 在 config 文件中查找指示符，并使用第一个匹配的段，因此缺省配置应该在所有 Host 段之后。不过可千万别让缩排方式和空格给搞糊涂了。文件结尾处可能是这样：

```
# 最后的Host块
Host server.example.com
User linda

# 缺省
User smith
```

你的本意是登录到 server.example.com 上时用户名为 “linda”，登录到其他主机上时用户名缺省为 “smith”。但是，“User smith” 这一行依然属于 “Host server.example.com” 段内。由于前一个 User 语句指定的主机是 server.example.com，所以 “User smith” 不能与任何内容匹配， ssh 会将其忽略。正确的写法应该是：

```
# 最后的Host块
Host server.example.com
User linda

# 缺省
Host *
User smith
```

问： .ssh2/ssh2_config 文件工作不正常，为什么？

答： 请参看前面针对 SSH1 的回答。但是 SSH2 的优先级规则和 SSH1 正好相反：如果多个配置行都可以和目标主机匹配，那么最终使用的不是第一个，而是最后一个。因此，缺省值应该放在文件的最前面。

12.2.5.3 ssh

问：我想用转义字符序列挂起 ssh，但是我运行的 ssh 超过两层了（从一台主机到另一台主机再到第三台）。如何才能挂起中间一层的 ssh？

答：一种方法是使用不同的转义字符启动每一层的 ssh；否则，就只有连接链中最前面的 ssh 可以解释转义字符并挂起。

或者有个更聪明的办法。如果将转义字符输入两次，即“后转义”(meta-escape)，就能绕过其一般的功能而传递转义字符本身。所以，在存在使用缺省转义字符“~”的 ssh 会话链时，挂起其中第 n 个的方法是：输入回车，输入 n 个转义字符，然后输入 Ctrl-Z。

问：我用命令行方式在后台运行了一个 ssh，但是它自己挂起了，除非用 fg 才能激活，为什么？

答：用 -n 命令行选项指示 ssh 不要从 stdin 读数据（实际上它没用你的终端，而是在 /dev/null 重新打开了一个 stdin）。不然如果 ssh 从后台读 stdin，shell 的任务控制机制就会将其挂起。

问：ssh 显示 “Compression level must be from 1 (fast) to 9 (slow, best)”，然后就退出了。

答：可能是 ~/.ssh/config 文件中 CompressionLevel 的值不合法。该值必须是 1 至 9 之间的整数（包括 1 和 9）。

问：ssh 显示 “rsh not available” 然后就退出了，为什么？

答：是 SSH 连接失败之后客户端根据设置降级使用 rsh 连接。[7.4.5.8] 而服务器编译时就不支持降级使用 rsh，或者指向 rsh 可执行文件的路径无效。[4.1.5.12] 如果你觉得 SSH 连接不应该失败，就用调试模式找找原因。否则，只可能是 SSH 服务器的配置不支持 rsh 连接。

问：ssh1 说 “Too many identity files specified (max 100)”，然后就退出了，为什么？

答：SSH1 代码限制了每个会话最多只能有 100 个身份标识文件（私钥文件）。如果出现这种情况，要么是你运行 ssh1 的命令行里有超过 100 个 -i，要么是 ~/.ssh/config 的某一项包含超过 100 个 IdentityFile 关键字。如果 SSH 命令行和 / 或 config 文件不是由另一个程序自动生成的，你就没可能看到这条消息。

问： ssh1说“Cannot fork into background without a command to execute”然后就退出了，为什么？

答： 你使用`-f`了，对吧？这个选项的意思是客户端在认证结束之后就转到后台，接着运行远程命令。如果你没有指明远程命令，就可能出现这个错误。你输入的命令可能是这样：

```
# 这是错误的  
$ ssh1 -f server.example.com
```

只有给`ssh1`提供一个当其转到后台之后执行的命令时，`-f`标志才是有意义的：

```
$ ssh1 -f server.example.com /bin/who
```

如果SSH会话的作用只用于端口转发，那么你可能不想指定什么命令。不过还是得提供一个；因为这是SSH1协议的要求。用`sleep 100000`吧，不过别用死循环之类东西，比如`shell`命令：`while true; do false; done`。虽然对于你来说效果都一样，不过在服务器上你调用的远程`shell`将吞噬CPU的所有空闲时间。要是惹恼了系统管理员，你的账号可就危险了。

问： ssh1说“Hostname or username is longer than 255 characters”然后就退出了，为什么？

答： `ssh1`要求远程主机或远程账号名的字符长度最多为255个字符。出现这一错误的原因可能是命令行或配置文件中有一个主机名或用户名的长度超出了该限制。

问： ssh1说“No host key is known for <server name> and you have requested strict checking (or ‘cannot confirm operation when running in batch mode’),”然后退出了，为什么？

答： 客户端无法在其已知名主机列表中找到服务器的主机密钥，而设置又不允许其自动加入密钥（或者它是以批处理方式运行的，无法向你确认可否加入）。所以必须手工加到每账号或服务器范围配置的已知名主机列表文件中。

问： ssh1说“Selected cipher type ... not supported by server”然后就退出了，为什么？

答： 你要求`ssh1`使用某个特定的加密算法，不过该SSH1服务器不支持。通常，SSH1客户端和服务器通过协商来确定使用哪种加密算法，因此你可能在`ssh1`命令行中用`-c`标志或在配置文件中用`Cipher`关键字强制要求使用某个特定加密算法。解决的办法是，不要指定算法，让客户端与服务器去解决或选择其他算法。

问： ssh1 说 “channel_request_remote_forwarding: too many forwards” 然后就退出了，为什么？

答： ssh1 限制每个会话最多包含 100 个转发，可能你请求的转发数太多了。

12.2.5.4 scp

问： scp 显示如下的出错信息：“Write failed flushing stdout buffer. write stdout: Broken pipe.” 或 “packet too long”，为什么？

答： 可能是 scp 连接时 shell 启动文件（比如 `~/.cshrc`, `~/.bashrc`）正在向标准输出设备写消息。这会妨碍两个 `scp1` 程序（或 `scp2` 与 `sftp-server`）之间的交互。如果看不到明显的输出命令，试试看能否找到 `stty` 或 `tset` 命令，这些命令也可能输出一些东西。

解决的办法是从启动文件中去掉这些烦人的命令，或者将其压缩至非交互式会话中：

```
if ($?prompt) then
    echo 'Here is the message that screws up scp.'
endif
```

最新版SSH2加入新的服务器设置语句`AllowCshrcSourcingWithSubsystems`，将其设为 no 可防止发生此类问题。

问： scp 显示出错消息 “Not a regular file.” 为什么？

答： 你是在拷贝目录么？用 `-r` 才能执行递归拷贝。如果没有，那你可能在拷贝某些无法拷贝的特殊文件，如设备节点、套接字或命名管道。对有问题的文件执行 `ls -l` 操作，如果文件描述中的第一个字符不是 “-”（表示常规文件）或 “d”（表示目录），可能就是出了这种问题。你不会真的想拷贝那个文件吧？

问： 为什么通配符和 shell 变量在 scp 命令行中不起作用？

答： 通配符和 shell 变量首先是由本地 shell 扩展的，然后才传到远程机器上。在 scp 运行之前，扩展工作就已经完成了。所以，如果输入：

```
$ scp server.example.com:a* .
```

本地 shell 就会试图寻找与模式 `server.example.com:a*` 匹配的本地文件。这可能不是你的本意吧。实际上你是想寻找 `server.example.com` 上与 `a*` 匹配的文件，并将其拷贝至本地。

有些 shell，特别是 C 及其派生 shell，仅仅报告一下“*No match*”便退出了。Bourne 及其派生 shell (*sh*、*ksh*、*bash*) 如果找不到匹配的文件，就会将字符串 *server.example.com:a** 传到服务器上，这正如你所愿。

同理，如果你想把远程邮件拷贝到本地机器上，下面的命令：

```
$ scp server.example.com:$MAIL .
```

可能不能达到预期的效果。*scp* 执行之前，\$MAIL 就在本地扩展了。除非本地与服务器上 \$MAIL 的内容（碰巧）相同，不然执行结果肯定和预期的不同。

我们可千万不能寄希望于 shell 和巧合，而应该将通配符和变量转义，别让本地 shell 解释它们就好了：

```
$ scp server.example.com:a\* .
$ scp 'server.example.com:$MAIL' .
```

请参看附录一中 *scp2* 的一般语法。

问：我用 *scp* 把文件从本地拷贝到远程机器上。运行没有出错。不过等我登录到远程的机器上，才发现文件没拷上去，为什么？

答：你是否漏了冒号？假设你想把文件 *myfile* 从本地机器拷贝到 *server.example.com*。正确的命令应该是：

```
$ scp myfile server.example.com:
```

如果忘记写最后一个冒号：

```
# 错误!
$ scp myfile server.example.com
```

myfile 就会给拷贝成本地文件“*server.example.com*”。看看本地是否有这个文件。

问：如何才能让别人用我的账号拷贝文件，又不让他登录？

答：这主意可不好。即使能限制其仅使用 *scp*，也无法保护你的账号。他可以运行：

```
$ scp evil_authorized_keys you@your.host:.ssh/authorized_keys
```

哦，他可是把你的 *authorized_keys* 文件换掉了呢，这下他就有完全登录的权限了。也许，一条聪明的强制命令可以帮你实现让你的朋友只能运行有限程序的目的。[8.2.4.3]

问：*scp -p* 能保护文件时间戳及访问模式，它还能保护文件的所有权吗？

答：不能。远程文件的所有权是由 SSH 认证结果确定的。假设用户 smith 在本地机 *L* 和远程机 *R* 上都有账号。如果本地的 smith 用 *scp* 将文件拷贝至远程的 smith 账号中，而此过程由 SSH 提供认证，那么远程文件的所有者就是远程的 smith。如果你想将文件的所有者换成另一个远程用户，那么 *scp* 必须能通过那个用户的认证才行。*scp* 不知道用户及其 uid 的内容，而且只有 root 可能改变文件的所有权（至少在现今大多数 Unix 系统中如此。）

问：*scp -p* 不保护文件的属主信息。不过我是超级用户，我想在两台机器间拷贝整个目录 (*scp -r*)，要拷贝的文件所有者不同。我应该如何在拷贝时保持其所有者信息呢？

答：这样就别用 *scp*，应该用 *tar* 命令，并用 *ssh* 管道传递。具体方法是在本地机器上输入：

```
# tar cpf - local_dir | (ssh remote_machine "cd remote_dir; tar xpf -")
```

12.2.5.5 sftp2

问：*sftp2* 显示 “Cipher <name> is not supported. Connection lost.” 为什么？

答：*sftp2* 在内部会调用 *ssh2* 命令连接 *sftp-server*。[3.8.2]首先搜索该用户的 PATH，定位 *ssh2* 可执行文件；这里没有使用在代码规定路径的办法。如果你装了不止一个版本的 *SSH2*，那么 *sftp2* 可能调用错误的 *ssh2* 程序，因此可能显示前面给出的错误信息。

例如，假设你同时安装了 *SSH2* 与 F-Secure *SSH2*。*SSH2* 安装在通常的位置 */usr/local*，而 F-Secure 安装在 */usr/local/f-secure*。一般情况下你使用 *SSH2*，那么 PATH 中就有 */usr/local/bin*，但没有 */usr/local/f-secure*。由于需要 CAST-128 密码，而 *SSH2* 中没有，因此你现在决定使用 F-Secure 版的 *scp2*。首先，确认 *SSH* 服务器支持 CAST-128：

```
$ /usr/local/f-secure/bin/ssh2 -v -c cast server
...
debug: c_to_s: cipher cast128-cbc, mac hmac-sha1, compression none
debug: s_to_c: cipher cast128-cbc, mac hmac-sha1, compression none
```

要求已满足，接下来执行 *scp2*，结果却是：

```
$ /usr/local/f-secure/bin/scp2 -c cast foo server:bar
FATAL: ssh2: Cipher cast is not supported.
Connection lost.
```

scp2 运行的 *ssh2* 不是 */usr/local/f-secure/bin*, 而是 */usr/local/bin/ssh2*。解决方法是在 PATH 中将 */usr/local/f-secure/bin* 置于 */usr/local/bin/ssh2* 之前, 或用 *scp2 -S* 指定 *ssh2* 的位置。

其他 SSH 程序相互调用时也会出现同样的问题。在同时安装 2.1.0 与 2.2.0 的系统上使用基于主机的认证时也会出现冲突。后一个 *ssh2* 运行了前一个的 *ssh-signer2*, 由于客户 / 签名协议已经改变, 因此该 *ssh2* 就会挂起。

问: *sftp2* 显示 “*ssh_packet_wrapper_input: invalid packet received.*” 为什么?

答: 这个错误发生得很奇怪, 不过其原因却很普通。远程账号的 shell 启动文件正在向标准输出设备输出什么东西, 而本例中 *stdout* 不是一个终端设备。*sftp2* 试图将这一输出解释成 SFTP 报文协议的一部分, 结果只能是失败了。

你已经看到, *sshd* 用 shell 启动 *sftp-server* 子系统。用户的 shell 启动文件打印出一些东西, 而 SFTP 客户端将这些信息解释成 SFTP 协议包。该操作失败后, 客户端就会显示出错信息, 然后退出; 由于包的第一个字段表示长度, 所以你看到的总是那么一条信息。

解决问题的方法是别在非交互式运行的 shell 启动文件中输出信息。例如, 如果 *stdin* 是终端, 那么 *tcsh* 将设置 *\$interactive* 的值。SSH-2.2.0 中的 *AllowCshrc-SourcingwithSubsystems* 标志可解决此类问题, 其缺省值为 *no*, 表示不让 shell 运行该用户的启动文件。[5.7.11]

12.2.5.6 端口转发

问: 我想用端口转发, 但是 *ssh* 显示 “*bind: Address already in use.*” 为什么?

答: 你试图使用的端口正在被监听端 (如果用 *-L* 转发, 就是本地主机, 如果用 *-R* 转发, 就是远程主机) 的另一个程序使用。试试 *netstat -a*, 这个命令在多数 Unix 平台和一些 Windows 平台上都有。如果看到你要使用的端口处于 *LISTEN* 状态, 那么你就知道别的程序正在使用该端口。我们应该首先检查是否还有别的 *ssh* 正在转发这个端口; 如果没有, 那就换一个尚未使用的端口作转发。

有时看起来可能没有别的程序使用你的端口, 不过还是会发这种问题。特别是在你试验转发的功能, 重复启动同一 *ssh*, 转发同一端口时更是如此。如果你启动的这些 *ssh* 中的最后一个没有正常退出 (你将其终止, 或它自己死掉,

或另一端的程序将连接强制关闭，等等），那么本地的TCP套接字可能还处于TIME_WAIT状态（用前面提到的netstat程序可看到这一点）。如果发生这种情况，你就必须等待几秒钟，让该套接字超时，退出这种状态，然后端口就又重新可用了。当然，如果你没耐心等，还可以选择其他端口。

问：我该如何用端口转发保护FTP？

答：这是一个复杂的问题。^[11.2] FTP有两种类型的TCP连接，控制连接和数据连接。控制连接传输登录名、口令及FTP命令；它处于TCP的21端口上，可以用标准的方法转发。方法为，在两个窗口中运行：

```
$ ssh -L2001: name.of.server.com:21 name.of.server.com  
$ ftp localhost 2001
```

FTP客户端可能得（通过执行passive命令）进入被动模式。FTP数据连接传输需发送的文件。这些连接建立在随机选择的TCP端口上，不能以通常方式转发，否则会很麻烦。如果网络系统中有防火墙或NAT，可能还要一些额外的操作（或者甚至根本不能实现转发）。

问：X转发不工作怎么办？

答：使用ssh -v，看看是否能通过输出信息明确断定问题所在。如果不能，则按下列步骤检查：

- 确保使用SSH之前X已经启动。检查方法是先运行一个简单的X客户端程序，如，xlogo或xterm。同时，必须已经设置好DISPLAY变量，否则SSH不会建立X转发。
- X转发必须在客户端与服务器上同时启用，且没有被目标账号禁用（即，authorized_keys文件中不包括no-X11-forwarding）。
- sshd必须能找到xauth程序，并在服务器上运行。如果不能，运行ssh -v就能发现。在服务器端可通过XAuthLocation(SSH1、OpenSSH)解决，或者在远程shell的启动文件中设置PATH变量（其中包含xauth）。
- 不要自己设置远程的DISPLAY变量。sshd会为转发会话自动设置正确的DISPLAY值。如果你的登录命令或shell启动文件中有给DISPLAY无条件赋值的语句，请将其改成仅在不存在X转发时赋值。
- OpenSSH也设置远程的XAUTHORITY变量，它将xauth证书文件放置在/tmp下。请查看你是否误将其覆盖。正确设置应该类似于：

```
$ echo $XAUTHORITY  
/tmp/ssh-maPK4047/cookies
```

一些 Unix 的标准 shell 启动文件（如，*/etc/bashrc*, */etc/csh.login*）中确实会将 *XAUTHORITY* 无条件赋值为 *~/.Xauthority*；如果是这样，必须请系统管理员帮助解决；启动文件只能在该变量没有值时才能对其赋值。

- 如果你使用了 SSH 启动文件 (*/etc/sshrc* 或 *~/.ssh/rc*)，远程的 *sshd* 就不会替你运行能加入代理密钥的 *xauth*；这个任务必须由启动文件完成，即从标准输入上接收 *sshd* 发出的代理密钥类型及内容。

12.3 其他 SSH 资源

如果本章中没有你的问题，请试试利用下面这些 Internet 资源。

12.3.1 Web 站点

遇到问题时首先应该想到查看 SSH FAQ，那里有很多常见问题的答案：

<http://www.employees.org/~satch/ssh/faq/>

SSH 主页，由 SSH Communications Security 维护。在这里可以查找一般信息及相关内容的链接：

<http://www.ssh.com/>

已经不更新的 Secure Shell Community 网站：

<http://www.ssh.org/>

SSH2 产品编译错误库：

<http://www.ssh.org/support.html>

OpenSSH 相关信息：

<http://www.openssh.com/>

当然还有本书的网站：

<http://www.oreilly.com/catalog/sshtdg/>

<http://www.snailbook.com/>

12.3.2 Usenet 新闻组

Usenet 上的新闻组 *comp.security.ssh* 讨论与 SSH 有关的技术问题。如果你不能访问 Usenet，也可以在 Deja.com 上阅读并检索其中的文章：

<http://www.deja.com/usenet/>

当然其他保存 Usenet 文章的网站也可以。

12.3.3 邮件列表

如果你是软件开发人员，有兴趣开发 SSH，或做 beta 测试，或想讨论 SSH 的安装及内部实现问题，你可以考虑加入 SSH 邮件列表。加入方法是给 *majordomo@clinet.fi* 发一封信：

```
To: majordomo@clinet.fi  
Subject: (blank)  
subscribe ssh
```

请注意，这个邮件只讨论技术问题，不能问“哪里能找到 Commodore 64 版的 SSH？”所以，订阅前请阅读最新的 SSH 消息，看看这个邮件是否适合你：

<http://www.cs.hut.fi/ssh-archive/>

在用该邮件列表发布问题之前，请用调试或详细模式运行 SSH 客户端及服务器，把全部调试信息都记录下来。

如果你无意开发或测试 SSH，只想收到 SSH 的主要声明，请加入 *ssh-announce* 邮件列表：

```
To: majordomo@clinet.fi  
Subject: (blank)  
subscribe ssh-announce
```

12.4 错误报告

如果你相信你已经发现了某个 SSH 版本的 bug:

1. 如果可能, 请先查看这个 bug 是否已经有人报告过了。
2. 将这个 bug 报告给开发商, 并同时告知你的软、硬件配置详情。

SSH1 及 SSH2 的 bug 应该报告给 ssh-support@ssh.com。此外, SSH Communication Security 也有一个 SSH2 的 bug 报告表单:

<http://www.ssh.com/support/ssh/>

下面的网站上有 OpenSSH 的 bug 报告方法的说明:

<http://www.openssh.com/>

同时还有 FAQ 和邮件订阅信息。

F-Secure Corporation 的用户支持网页在:

<http://www.f-secure.com/support/ssh/>

接收 bug 报告的邮箱是 F-Secure-SSH-Support@f-secure.com。

第十三章

其他产品概述

本章内容

- 通用功能
- 将要介绍的产品
- 产品列表
- 其他与 SSH 有关的产品

SSH 不只是 Unix 系统中的一种技术，它在 Windows、Macintosh、Amiga、OS/2、VMS、BeOS、PalmOS、Windows CE 以及 Java 上都已实现。有一些早期的产品已经完成，其他的则是志愿者们移植的软件，完成程度各不相同。

本书的剩余部分将涉及 Windows (95、98、NT、2000) 及 Macintosh 上的一些功能强大的 SSH 产品。我们认为这些产品内容完整，可用性强。我们也会提供其他产品的信息，如果你愿意，可以自己试验。

我们建了一张主页，其中包含所有我们知道的与 SSH 有关的产品的链接。从本书主页：

<http://www.oreilly.com/catalog/sshtdg/>

上的“Authors' Web Site”链接点下去，或直接访问：

<http://www.snailbook.com/>

还有一个第三方的主页，里面全是免费 SSH 产品的相关文档：

<http://www.freessh.org/>

13.1 通用功能

每一种SSH产品都有其独特的功能，但是这些产品有一个共同点：用一个客户端程序安全登录到远程系统。有些客户端是基于命令行的，其他则有图形化的终端，可以打开一个窗口，还有很多的属性供用户设置。

至于其他功能，各个产品则区别很大。只有少数产品具有安全文件拷贝工具(*scp*和*sftp*)、远程批处理命令执行工具、SSH服务器、SSH代理、精确认证及加密算法等功能。

几乎所有产品中都有一个生成公钥和私钥的工具。例如，SSH1/SSH2移植产品中的*ssh-keygen*、F-Secure SSH客户端的Keygen Wizard以及SecureCRT的Key Generation Wizard。值得注意的是，Macintosh上的NiftyTelnet SSH自己不能生成密钥，但是可以接受其他遵循SSH-1标准格式的程序生成的密钥。

13.2 将要介绍的产品

Windows平台：

- F-Secure SSH客户端，这是F-Secure Corporation发布的商用SSH客户端，支持SSH-1及SSH-2（也有Macintosh版）。
- SecureCRT，Van Dyke Technologies发布的商用SSH客户端，支持SSH-1及SSH-2。
- Sergey Okhapkin移植的Windows版SSH1。

Macintosh平台：

- NiftyTelnet SSH，Jonas Walldén制作的免费SSH客户端。它是基于免费的Telnet客户端NiftyTelnet开发的。

13.3 产品列表

我们无法列出每一种SSH产品，希望这里总结的内容能有助于你自己探索。下表按

平台顺序列出我们提到过的每一种 SSH 产品的主要功能，包括本书前面讨论过的 Unix 产品（SSH1、SSH2、OpenSSH、F-Secure SSH）。其中每个比较项的含义如第一张表所述。

比较的项目	含义
名称	产品的名称。如果后面有“(推荐)”字样，则表明我们已经评估过该程序，并推荐使用。如果某产品未列为推荐产品，那它也可能不错，只不过我们还没有对其做全面的评测
平台	程序运行在 Windows、Macintosh 还是 Unix 上？在此并不指明具体的 Windows 版本（NT、98、2000 等），因为我们无法测试全部 Windows 平台。详情请咨询供货商
版本	本书（英文版）出版时该软件的最新版本号
分发方式	该程序分发多少个？这里只有概要的许可信息；全部信息请参看产品文档
协议	该产品实现了 SSH-1 还是 SSH-2？还是两者都实现了？
远程登录	该产品能打开登录远程机器的 shell 吗？此处“ssh”表示 SSH1 或 SSH2 的命令行方式，“终端程序”表示图形界面
远程命令	该产品能像 ssh 客户端程序那样，调用远程 SSH 服务器上的独立命令吗？（即提供一个命令字符串作为最后一个参数）
文件传输	如果有程序可以在机器间安全传输文件的话，它是什么？
服务器	产品中包含 SSH 服务器吗？
认证	支持哪些认证方式？
密钥生成方法	能生成私有 / 公用密钥对吗？
代理	包含 SSH 代理吗？
转发	支持端口转发、X 转发吗？是两者都支持还是都不支持？
注释	一般信息及相关细节
联系方式	访问该软件的 URL

下表总结了很多 SSH 产品（本表跨多页）。

产品名称	AmigaSSH	SSH	JavaSSH	Java Telnet SSH Plug-in
平台	Amiga	BeOS	Java	Java
版本	3.15	1.2.26-beos	20/07/1998	2.0 RC3
发行方式	GPL	免费软件	可自由分发	GPL (GNU 公共许可)
协议	SSH-1	SSH-1	SSH-1	SSH-1
远程登录	终端程序	ssh	终端程序	终端程序
远程命令	无	ssh	无	无
文件传输	无	scp	无	无
服务器	无	无	无	无
认证	密码, 公钥	密码, 公钥, 信任主机	密码, 公钥	密码
密钥生成方法	ssh-keygen	ssh-keygen	?	无
代理	无	?	无	无
转发	无	端口 ,X	无	无
注释	NapsaTerm 与 SSH1 1.2.26 集 成; 需要 68020 及以上的 CPU 支持的	移植 SSH1 1.2.26.	需要 Java AWT 1.1	某 Java Telnet 应用程序 一部分
联系方式	http://www.lysator.liu.se/~lilja/amigassh/	http://www.bebits.com/app/703	http://www.cam.ac.uk/~fapp2/software/java-ssh/	http://www.mud.de/se/jta/doc/plugins/SSH.html

产品名称	MindTerm (推荐)	BetterTelnet	F-Secure SSHClient	NiftyTelnet SSH (推荐)
平台	Java	Macintosh	Macintosh	Macintosh
版本	1.1	2.0fc1	2.1	1.1 R3
发行方式	GPL	GPL	商用软件	免费软件
协议	SSH-1	参看注释	SSH-1, SSH-2	SSH-1
远程登录	终端程序	参看注释	终端程序	终端程序
远程命令	有	参看注释	无	无
文件传输	scp, 隧道式ftp	参看注释	隧道式ftp	图形化scp
服务器	无	参看注释	无	无
认证	密码, 公钥, 信任主机, TIS, sdii-token	参看注释	密码, 公钥	公钥
密钥生成方法	有	参看注释	有	无
代理	无	参看注释	无	无(但可记忆密码)
转发	端口, X	参看注释	端口, X	无
注释	可作为独立程序或applet使用; 已在很多操作系统下测试过	本书(英文版)出版时尚无SSH支持(这是先前的出口限制造成的), 不过应该很快就能恢复	也有Windows版	功能最小化, 很有用
联系方式	http://www.mindbright.se/	http://www.cstone.net/~rbraun/mac/telnet/	http://www.f-secure.com/	http://www.lysator.liu.se/~jonasw/freeware/niftyssh/

产品名称	SSH DOS	SSH OS2	Top Gun SSH	Ish
平台	MS-DOS	OS/2	PalmOS	Unix
版本	0.4	v03	1.2	1.0.3
发行方式	GPL	?	可自由分发	GPL
协议	SSH-1	SSH-1	SSH-1	SSH-2
远程登录	有	ssh, 终端程序	终端程序	有
远程命令	无	ssh	无	有
文件传输	无	scp	无	无
服务器	无	未完成	无	有
认证	密码	密码, 公钥, 信任主机	密码	密码, 公钥, SRP
密钥生成方法	无	有	无	有
代理	无	有	无	无
转发	无	端口, X	无	端口
注释	功能最小化; 运行于低端机 上; 基于 Putty 及 SSH1 1.2.27	基于 SSH1 1.2.13	基于 Palm Pilot 上的 Top Gun Telnet	就目前的工作进展来看 很有希望, 但仍旧不安全
联系方式	http://www.vein.hu/~nagyd/	ftp://ftp.cs.hut.fi/pub/ssh/old/os2/	http://www isaac.cs berkeley.edu/pilot/	http://www.net.lut.ac.uk/psst/

产品名称	ossh	FISH	sshexec.com	AppGate
平台	Unix	VMS	VMS	Windows, Unix, Macintosh
版本	1.5.6	0.6-1	5alpha1	
发行方式	BSD 许可证	可自由分发	免费软件	商用软件
协议	SSH-1	SSH-1	SSH-1	
远程登录	ssh	有	N/A	
远程命令	ssh	有	N/A	
文件传输	scp	无	无	
服务器	sshd	无	有	
认证	密码, 公钥, 信任主机	密码, 公钥, 信任主机, TIS (untested)	密码, 公钥	
密钥生成方法	ssh-keygen	有	有	
代理	ssh-agent	无	无	
转发	端口, X	无	X	
注释	移植了 SSH1 1.2.12		VMS 服务 器: 开发中; 初学者勿用	
联系方式	ftp://ftp.nada.kth.se/pub/krypto/ossh/	http://www.free.ln.se/fish/	http://www.er6.eng.ohio-state.edu/~jonesd/ssh/	http://www.appgate.com/

产品名称	Chaffee Port	Free FiSSH	F-Secure SSH Client (推荐)	Mathur Port
平台	Windows	Windows NT, 2000	Windows	Windows
版本	1.2.14a	?	4.1	1.2.22-Win32-beta
发行方式	?	对非商业用途 免费	商用软件	公用许可及其他
协议	SSH-1	SSH-1	SSH-1, SSH-2	SSH-1
远程登录	ssh	终端程序	终端程序及 ssh2 命令行 终端	ssh
远程命令	ssh	?	ssh2	ssh
文件传输	scp	?	scp2, sftp2, 图形化 sftp 终端	scp
服务器	无	?	无	sshd
认证	密码, 公钥	?	密码, 公钥	密码, 公钥
密钥生成方法	ssh-keygen	?	有	ssh-keygen
代理	无	?	无	?
转发	端口, X	?	端口, X	端口, X
注释	无文档; 基于 SSH1 1.2.14	我们试验过程 中表现不稳定 (原因是缺少 相关资料)	也有 Mac- intosh 版	从 1998 年起开发的 Alpha 软件, 文档很少。移 植了 SSH1 1.2.22, 带 <i>cygnus dll</i>
联系方式	http://bmrc.berkeley.edu/people/chaffee/winntutil.html , ftp://bmrc.berkeley.edu/pub/winnt-devel/ssh1.2.14a.exe	http://www.massconfusion.com/ssh/	http://www.f-secure.com/	ftp://ftp.franken.de/pub/win32/develop/gnuwin32/cygwin/porters/Mathur_Raju/

产品名称	Metro State SSH(MSSH)	Okhapkin Port	PenguiNet	PuTTY (推荐)
平台	Windows	Windows	Windows	Windows
版本	?	1.2.26, 1.2.27, 2.0.13	1.05	Beta 0.48
发行方式	GPL	与 SSH1, SSH2 相同	共享软件	可自由分发
协议	SSH-1		SSH-1	SSH-1
远程登录	参看注释	ssh	终端程序	终端程序
远程命令	参看注释	ssh	无	无
文件传输	参看注释	scp	无	scp
服务器	无	sshd (仅有 Windows NT 版)	无	无
认证	密码	密码, 公钥, 信任主机	密码, 公钥, Rhosts, RhostsRSA	密码, TIS
密钥生成方法	无	ssh-keygen	有	无
代理	无	无	无	无
转发	端口	端口, X	无	无
注释	仅进行 TCP 端 口转发, 对 Telnet 及 Email 连接有特 殊支持	包含两个 SSH1 与一个 SSH2 的移植产品		常用软件; 以包含 scp 出 名
联系方式	http://csi.mscd. edu/MSSH/	http://miracle. geol.msu.ru/ sos/	http://www. siliconcircus. .com/	http://www.chiark. greenend.org.uk/ ~sgtatham/putty/

产品名称	SSH Secure Shell (推荐)	SecureCRT (推荐)	SecureFX (推荐)	SecureKoalaTerm
平台	Windows	Windows	Windows	Windows
版本	2.1.0	3.1.2	1.0	1.0
发行方式	免费非商用分发	商用软件	商用软件	共享软件
协议	SSH-2	SSH-1, SSH-2	SSH-2	SSH-1, SSH-2
远程登录	终端程序	终端程序	无	终端程序
远程命令	无	无	无	无
文件传输	图形化的 scp2	Zmodem (安全的)	FTP (安全的)	Zmodem (安全的)
服务器	无	无	无	无
认证	密码, 公钥	密码, 公钥, TIS	密码, 公钥	密码, 公钥
密钥生成方法	有	RSA, DSA	有	有
代理	无	无	无	无
转发	端口, X 转发	端口, X 转发	无	无
注释 模拟器	这一最近发布的软件将极大地促进 SSH2 的使用; 其中 scp2 客户端尤其出色, 它模拟了 Windows 的浏览器, 可在机器间进行安全的文档拖放操作; 文档内容丰富; 其 SSH2 服务器是一个独立产品	很可靠; 我们认为在商用 Windows 客户端上很好用	基于 SSH2 的安全图形化 FTP 客户端	支持 SSH 的图形化终端
联系方式	http://www.ssh.com/	http://www.vandyke.com/	http://www.vandyke.com/	http://www.midasoft.com/

产品名称	therapy Port	TTSSH (推荐)	Zoc	sshCE
平台	Windows	Windows	Windows	Windows CE
版本	0.2	1.5.1	3.14	1.00.40
发行方式	参看注释	可自由分发	商用软件	免费软件
协议	参看注释	SSH-1	SSH-1	SSH-1
远程登录	参看注释	终端程序	终端程序	终端程序
远程命令	参看注释	无	无	无
文件传输	参看注释	Kermit, Xmodem, Zmodem, B-Plus, Quick-VAN (全部是安全的)	Kermit, Ymodem, Zmodem	无
服务器	参看注释	无	无	无
认证	参看注释	密码, 公钥, 信任主机, TIS	密码	密码
密钥生成方法	参看注释	无	无	无
代理	参看注释	无	无	无
转发	参看注释	端口, X	无	无
注释	无技术支持, 已不再开发; 基于 SSH1 1.2.20	常用软件; Teraterm Pro 的 SSH 扩展, 免费终端程序	全功能的终端程序	当前是 beta 版
联系方式	http://guardian.hfu.tuwien.ac.at/therapy/ssh/	http://www.zip.com.au/~roca/ttssh.html	http://www.emtec.com/zoc/	http://www.movsoftware.com/sshce.htm

13.4 其他与 SSH 有关的产品

SecPanel 是一个图形化、点击式 SSH 客户端连接管理器; 用 tcl 编写:

<http://www2.wiwi.uni-marburg.de/~leich/soft/secpanel/>

ssh.el 是用于建立 *ssh* 客户端连接的 Emacs 式接口:

<http://munitions.vipul.net/software/network/ssh/ssh.el>

ssh-keyscan 是 *ssh-make-known-hosts* 的替代品，据说速度更快: [4.1.6]

<ftp://cag.lcs.mit.edu/pub/dm/source/ssh-keyscan-0.3b.tar.gz>

第十四章

Sergey Okhapkin

移植的 Windows

版的 SSH1

本章内容

- 获取并安装客户端
- 客户端的用法
- 获得并安装服务器
- 配置服务
- 小结

很多程序员都尝试将 SSH1 移植到 Windows 平台上。我们看到的多数移植要么尚未完成，要么已经不再开发了，要么发布的时候没有源代码。目前所见最好的移植产品是 Sergey Okhapkin 做的，所以本章将介绍他的工作。我们将他移植的软件称为“Okhapkin 的 SSH1”，以示与 SSH1 区分。

Okhapkin 的软件功能很好，但是安装比较困难。因此，仅推荐 Windows 高级用户使用。理想的前提是熟悉 MS-DOS 环境变量、*bzip2* 压缩文件、*tar* 包、Windows NT 资源工具箱（Windows NT Resource Kit），以及最重要的是在 PC 上手工安装软件的技巧。如果你根本就没有听说过这些概念，那么应该考虑换一种 Windows 下的 SSH 程序。不过从另一个角度来看，如果你坚持走完安装的全过程，就会获得一个功能强大的基于命令行的免费 SSH。

Okhapkin 对 SSH1 1.2.26、1.2.27 以及 SSH2 2.0.13 分别做了移植。此处介绍的是他移植的 1.2.26 版，原因是我们在安装这个版本时遇到的问题最少。

14.1 获取并安装客户端

Okhapkin 的 SSH1 可从作者位于俄罗斯的网站上找到：

<http://miracle.geol.msu.ru/sos>

多数Windows用户可能并不熟悉这个软件的发布方式。软件首先打包成一个*tar*包，*tar*是 Unix 系统中常见的文件格式。接着用**bzip2**压缩这个*tar*包，这种压缩工具在 Linux 用户中很流行。举个例子，Okhapkin 移植的 1.2.26 版软件，其*tar*文档经过**bzip**压缩之后成为 *ssh-1.2.26-cygwinb20.tar.bz2*。

这个版本的 Okhpakin SSH1 客户端 (*ssh1*、*scp1*)，适用于 32 位 Windows 系统；我们将其安装在 Windows95 中。服务器 (*sshd*) 只能在 Windows NT 上运行。

按我们说的保守安装方式，需要 40MB 磁盘空间存储 SSH 及 Cygwin 支持软件，安装过程本身还需要 20MB 的空间，因此应该保证有 60MB 可用空间。SSH 本身只占 1MB，因此在安装结束后若想节省空间，可以把大部分 Cygwin 删掉。

14.1.1 准备文件夹

安装软件之前，首先得在 C: 盘创建下列文件夹：

```
C:\usr  
C:\usr\local  
C:\usr\local\bin  
C:\etc  
C:\home  
C:\home\.ssh  
C:\tmp
```

请注意这里的句点！

必须用 DOS 的 *mkdir* 命令才能创建 *C:\home\.ssh*。Windows 不能创建名字以句点开头的文件夹。

```
C:\> mkdir C:\home\.ssh
```

14.1.2 准备 autoexec.bat

我们要对 *autoexec.bat* 进行两处修改。首先要将 *C:\usr\local\bin* 加入 MS-DOS 搜索路径。具体方法是在该文件中加入如下的行：

```
PATH=%PATH%;C:\usr\local\bin;C:\Cygwin\bin
```

其次要把环境变量 CYGWIN 设置成“tty”：

```
SET CYGWIN=tty
```

这一步是必需的，这样 *ssh1* 客户端才能以交互模式运行。最后应该保存 *autoexec.bat*，打开 MS-DOS 命令窗口，使上述改动生效：

```
C:\> C:\autoexec
```

14.1.3 创建密码文件

Unix 上登录名、密码及其他与用户有关的信息都保存在 */etc/passwd* 中。其中一行包括七个字段，相互由冒号隔开：

1. 一个登录名，可以是字母和数字串。
2. 一个星号。
3. 一个大于 0 的整数。
4. 一个大于 0 的整数。
5. 全名。
6. */home* 文件夹，你的 SSH 文件夹创建的地点。请注意斜线的方向；这不是 MS-DOS 的文件夹分隔符，而是问号键上的那个斜线。
7. */command.com* 程序。请再次留意斜线。

这就是 Unix 中 *passwd* 文件的格式。Okhapkin 的 *SSH1* 中只有字段 1 和 6 实际有用。其他字段的值可能以后会用到。下面是一个例子：

```
smith:*:500:50:Amy Smith:/home:/command.com
```

14.1.4 安装 Cygwin

Cygwin 是一组非常优秀的命令行程序。这些程序是从 GNU (<http://www.gnu.org>) 移植到 Windows 上的，其基础是被称为 Cygwin DLL 的代码库 (*cygwin1.dll*)。Okhapkin 的 *SSH1* 需要使用这个 DLL，也就是说，安装完 Cygwin 后，可将其他绝大多数文件删除。Cygwin 可从以下地址获得：

<http://sourceware.cygnus.com/cygwin>

我们用不着源代码，只安装二进制版本即可。正式的下载和安装过程花费时间较长，因此可以考虑仅下载 *cygwin1.dll*，不用下载其他程序。本书出版时，Cygwin 镜像机的 */pub/cygwin/latest/cygwin* 文件夹中都有这个文件（可从上面那个 URL 寻找），其分发格式是 gzip 压缩的 tar 文档（文件后缀为 *.tar.gz*），在 Windows 上使用 WinZip 即可解压。然后将 *cygwin1.dll* 复制到前面已经创建的文件夹 *C:\usr\local\bin* 中。

14.1.5 安装 bzip2

bzip2 是一个压缩 / 解压文件的程序。

<http://sourceware.cygnus.com/bzip2>

处有其 Windows 版。先将程序下载到 *C:\usr\local\bin*，这个程序不用安装就能运行。解出来的可执行文件名为 *bzip2095d_win32.exe*，不过今后的版本可能会用不同的名字。

将 *bzip2* 可执行文件重命名为 *bzip2.exe*:

```
C:\> cd \usr\local\bin  
C:\usr\local\bin> rename bzip2095d_win32.exe bzip2.exe
```

14.1.6 安装 Okhapkins 的 SSH1

首先下载 Okhapkin 的 SSH1 1.2.26，下载地址为：

<http://miracle.geol.msu.ru/sos>

要下载的文件名是：*ssh-1.2.26-cygwinb20.tar.bz2*。由于文件中包含多个句点，因此下载时系统可能会自动消除最后一个以外的所有句点，比如 *ssh-1_2_26-cygwinb20_.tar.bz2*。

然后用 *bzip2* 解压，生成一个 *tar* 文档：

```
C:\temp> bzip2 -d ssh-1_2_26-cygwinb20_tar.bz2
```

将这个 tar 文档释放到 C: 盘根目录下。通过这一步操作，C:\usr 中就得到了解包后的文件：

```
C:\temp> cd \
C:\> tar xvf \temp\ssh-1_2_26-cygwinb20.tar
```

如果没有安装完整的 Cygwin 包，[14.1.4] 系统中就可能没有 tar 程序。Windows 上流行的 WinZip 软件也能解包 tar 文档（当然是在运行了 bzip2 之后）。请确认解包的目的地址是否为 C: 盘根目录。

现在 SSH1 客户端软件就安装好了。

14.1.7 生成密钥对

在运行 Okhapkin SSH1 客户端之前，首先要设置 SSH 文件夹，并生成公钥认证使用的密钥对。具体方法是运行 ssh-keygen1：

```
C:\> ssh-keygen1
```

ssh-keygen1 会在 C:\home\ssh 文件夹中生成私钥文件 identity 和公钥文件 identity.pub。生成过程中输出的信息如下所示。“w: not found” 那一行可以忽略，它是由 Unix 与 Windows 的差异造成的，并没有什么危害。

```
Initializing random number generator...
w: not found
Generating p: .....++ (distance 352)
Generating q: .....++ (distance 140)
Computing the keys...
Testing the keys...
Key generation complete.
```

然后会让你确认一个保存密钥的文件名。要接受缺省值，请键入回车：

```
Enter file in which to save the key (/home/.ssh/identity): [此处键入回车]
```

然后让你输入私钥的口令。请选择适当的口令，这里要求你输入两次。口令不会在屏幕上显示。

```
Enter passphrase: *****
Enter the same passphrase again: *****
```

这样，密钥对就生成好了，它保存在 C:\home\ssh 中。如果打算连接某个 SSH 服务器，就将公钥 (*identity.pub*) 拷贝到那台主机上，并在 *~/.ssh/authorized_keys* 中加入该密钥。[2.4.3]

14.1.8 通过 ssh1 登录远程主机

现在，连接的准备工作都做好了。请运行 *ssh1* 客户端，同时提供远程登录名。假设要用“smith”登录 SSH 服务器 *server.example.com*。

```
C:\> ssh1 -l smith server.example.com
```

第一次登录某个远程主机时，*ssh1*会在向你确认之后将这个主机名加入其已知名主机数据库中。[2.3.1] 请回答 yes，然后继续：

```
Host key not found from the list of known hosts.  
Are you sure you want to continue connecting (yes/no)? yes  
Host 'relativity.cs.umass.edu' added to the list of known hosts.
```

最后需要输入口令：

```
Enter passphrase for RSA key 'You@YourPC': *****
```

如果一切顺利，现在你就已经通过 SSH 登录到这台远程主机上了。也可以使用普通方式通过 SSH 运行其他命令，即在命令行的末尾加上一条命令：

```
C:\> ssh1 -l smith server.example.com /bin/who
```

14.1.9 使用 scp1 实现文件的安全拷贝

用 *scp1* 应该也能实现安全拷贝。请试试看能否将一个文件拷贝到远程机器上：

```
C:\> scp1 C:\autoexec.bat smith@server.example.com:
```

14.2 客户端的用法

Okhpakin 的 SSH1 支持第五、六、七、九章中提到的绝大多数 SSH1 功能。区别就是把所有的 *~/.ssh* 的全都替换成 *C:\home\ssh*。例如，可在 *C:\home\ssh\config* 处创建一个客户端配置文件。[7.1.3]

不过 SSH 代理不能在 Windows 上运行，因此第六章提到的有关 *ssh-agent1* 和 *ssh-add1* 的所有内容，在此都不适用。

14.3 获取并安装服务器

Okhapkins 的 SSH1 服务器是 *sshd*，它可以被安装成 Windows NT 的一个服务来运行。它支持第五章除公钥认证之外的所有服务器配置功能。在 NT 登录认证中，用户名和密码是必需的，因此 SSH 无法绕过这一屏障去提供基于公钥的认证方式。

14.3.1 获取 *sshd*

Sergey Okhapkin 的网站上的 *sshd* 1.2.26 有两种格式：预编译好的可执行文件和源代码补丁。我们使用可执行文件。此外，还有人维护着一些包含 Sergey 的可执行文件及相关支持文件在内的包，我们比较喜爱的网站之一是：

http://www.gnac.com/techinfo/ssh_on_nt

14.3.2 获取 NT 资源工具箱

sshd 要想作为 NT 的服务运行，必须要使用 NT 资源工具箱中的三个程序：*instsrv.exe*、*srvany.exe* 和 *kill.exe*。前两个是将普通 NT 程序转变成 NT 服务的工具，第三个可以删除 NT 任务管理器删不掉的进程。

14.3.3 创建一个管理员级用户

sshd 必须由管理员级用户账号将其作为一个 NT 服务调用，因此现在我们得创建一个管理员级用户。运行用户管理器，执行下列操作：

1. 创建一个本地用户，比如说叫 *root*。
2. 将 *root* 加入管理员组。
3. 在“Options/User Rights”中选中复选框“Show Advanced User Rights”。

现在，给 root 赋予下列权限：

- Act as part of the operating system.
- Increase Quotas.
- Log on as a service.
- Replace a process level token.

然后关闭用户管理器，继续执行以下步骤。

14.3.4 安装服务器

首先将服务程序 *sshd.exe* 拷贝到选定的某个目录下，比如 *C:\bin*。完成安装过程之前，还得把 *sshd* 转换成一项 NT 服务，并用新创建的管理员用户启动该服务，然后创建若干注册表项：

1. 要把服务器作为一个 NT 服务安装，就需要执行下面的命令。这里假设使用的管理级用户名为 *root*，NT 资源工具箱位于 *C:\reskit*，计算机名为 *mypc*（这是一条命令，写在同一行里）。

```
C:\> C:\reskit\instsrv.exe SecureShellDaemon  
C:\reskit\srvany.exe -a mypc\root -p root
```

2. 创建下列注册表项。其中 HKLM 表示 HKEY_LOCAL_MACHINE：
 - 在 *HKLM\SYSTEM\CurrentControlSet\Services\SecureShellDaemon* 中创建一个字符串 “ObjectName”，其值是 “LocalSystem”。
 - 在 *HKLM\SYSTEM\CurrentControlSet\Services\SecureShellDaemon\Parameters* 中创建一个字符串 “Application”，其值是 “*C:\Bin\sshd.exe*”；再创建一个字符串 “AppParameters”，其值为 “*-f /etc/sshd_config*”。

14.3.5 生成主机密钥

服务器需要使用一主机密钥向 SSH 客户端惟一标识自己的身份。[\[5.4.1\]](#) 请用 *ssh-keygen1* 生成密钥，并将其存储在 *C:\etc* 中：

```
C:\> ssh-keygen1 -f /etc/ssh_host_key -N "" -C ""
```

14.3.6 编辑 sshd_config

你的服务器已经快运行起来了。现在要根据系统安全策略创建 *sshd* 的服务器范围配置文件。^[5.3.1] NT 上该文件位于 *C:\etc\sshd_config*。请参看第十章推荐的设置。

请确认是否正确给出了主机密钥等文件的位置。Cygwin 中 “/” 表示启动盘根目录，例如：

```
HostKey /etc/ssh_host_key  
PidFile /etc/sshd.pid  
RandomSeed /etc/ssh_random_seed
```

警告：如果 SSH 正在运行的时修改了 *sshd_config*，必须终止 *sshd* 并重新启动它，这样这些改变才能生效；^[14.3.9] 只在控制面板中的服务中停止并重启服务是不够的。

14.3.7 运行服务器

要运行 *sshd*，请打开服务控制面板，查找 SecureShellDaemon 服务，选中它，单击“启动”按钮。就是它！NT 任务管理器中就会显示出进程 *sshd.exe*。

14.3.8 测试服务器

如果在本地 PC 上同时安装了 *sshd* 和 *ssh1*，就可以试试自己连接自己了：

```
C:\> ssh1 localhost  
smith@127.0.0.1's password: *****
```

否则需连接其他站点：

```
c:\> ssh1 -l smith mypc.mydomain.org  
smith@mypc.mydomain.org's password: *****
```

如果不能连接，可用 *ssh1 -v* 打印诊断信息，据此判断问题出在何处。

14.3.9 终止服务

我们通常都用控制面板中的服务上的“停止”按钮来终止 NT 服务。不过对 *sshd* 而

言，即使从控制面板上看到服务已经停止了，实际上它还没有停止。所以必须手工删除进程。可以用 NT 资源工具箱中的 *kill.exe* 实现。先在 NT 任务管理器上找出 *sshd.exe* 的 pid (假设是 392)，然后：

```
C:\> kill 392
```

14.4 疑难解答

若 *ssh1* 或 *scp1* 的行为与预期的不同，则应使用 *-v* (详细模式, verbose) 让客户端在运行过程中打印调试信息。这些信息也许就是我们解决问题的线索。现在回答若干具体问题：

问：运行 *ssh1* 时提示 “You don't exist, go away!”，这是为什么？

答：也许你没有按照指示创建 *C:\etc\passwd*。还有，*passwd* 文件中不能包括盘符 (C:)，因为冒号会被错认为字段分隔符。

问：我无法创建 “.ssh” 或其他以句点开头的文件夹，为什么？

答：这在 Windows 图形界面中无法实现。应该在 MS-DOS 方式下执行 *mkdir* 命令 (*mkdir .ssh*)。

问：为什么 *ssh1* 说 “Could not create directory /home/.ssh” ？

答：你忘记先创建 *C:\home* 了。

问：*scp1* 说它找不到 *ssh1*，为什么？

答：你把 *C:\usr\local\bin* (即 *ssh1.exe* 的存放路径) 加入 MS-DOS 的 PATH 中了么？

问：*ssh-agent1* 不工作。它返回一条 “Bad modes or owner for directory '/tmp/ssh-smith'” 然后就退出了，为什么？

答：这样是对的。SSH 代理在 Windows 下不能运行，它要求有 Unix 域套接字。

问：我无法用 *sshd* 连接我在另一个域中的 NT 账号，为什么？

答：是的，肯定不能连接。NT *sshd* 只能连接本地账号，即连接目标必须与本地机器位于同一个域中 (注 1)。

注 1： 我们已经听说过 NT *sshd* 会认证来自任意可信域的账户，但我们尚未证实这一点。

问：还有问题怎么办？

答：下面这些网站不错，去看看吧：

<http://miracle.geol.msu.ru/sos/ssh-without-cygwin.html>

<http://marvin.criadvantage.com/caspian/Software/SSHD-NT>

http://www.gnac.com/techinfo/ssh_on_nt

<http://www.onlinemagic.com/~bgould/sshd.html>

<http://v.iki.fi/nt-ssh.html>

14.5 小结

Sergey Okhpkin 的 SSH1 1.2.26 是目前我们看到的在 Windows 平台上移植地最好的 SSH1。其中可用的程序包括 *ssh1*、*scp1* 及 *ssh-keygen1*，在典型 SSH 环境中这些就足够了；此外还有 *sshd*。不过请留意，1.2.26 不是 SSH1 最新版本，因此其中可能存在一些安全漏洞，不过以后的正式版本可能已经修正了这些问题。

本章内容

- 获取及安装
- 客户端的基本用法
- 密钥管理
- 客户端的高级用法
- 配置
- 脚本语言
- 安全性

第十五章

SecureCRT (Windows)

Van Dyke Technologies 开发的 SecureCRT 是 Microsoft Windows 9x、NT、2000 上的商用 SSH 客户端软件。该软件以终端程序的面貌出现；而实际上，其基础正是 Van Dyke Technologies 的另一个产品——终端程序 CRT。因此，SecureCRT 终端的可配置性很强。它可模拟多种类型的终端，既可以使用 Telnet 登录，也可以使用 SSH 登录，它有一个脚本语言、一个键盘映射编辑器，并支持 SOCKS 防火墙，还有聊天等很多功能。不过我们只着眼于它的 SSH 功能。

SecureCRT 可以同时支持 SSH-1 及 SSH-2。其他重要功能包括端口转发、X11 包转发，而且可以支持多个 SSH 实例。它不支持代理。安全拷贝并非用 *scp* 之类的程序实现，而是使用 ZModem——一种古老的文件上传/下载协议（当然远程的机器必须也安装 ZModem）。当通过 SSH 登录时用了 ZModem，那么文件传输就是安全的。

本章的结构与这本书前半部分讨论 Unix 上 SSH 实现的结构一致。我们会适时引导你到前面的章节中查看详细内容。

我们讨论的 SecureCRT 版本号为 3.1.2，2000 年 9 月发布。

15.1 获取及安装

可从 Van Dyke Technologies 公司网站上购买并下载 SecureCRT：

<http://www.vandyke.com/>

上面也有免费试用版，安装后有30天的试用期，所以你可以先行试用，然后再购买。安装过程非常简单，不易出错。该软件发布形式是单独一个exe；仅需运行这个exe即可安装程序。解包过程中需要一个序列号和许可证，Van Dyke 会给每个注册用户提供这些信息。请根据屏幕提示将软件安装在任意路径中。我们就用缺省路径了。

15.2 客户端的基本用法

程序安装好之后，可以配置一个新的Session（SecureCRT用这个词称呼一组设置项）（注1）。从File菜单中选择“Connect...”，在弹出的窗口中单击Properties按钮，打开如图15-1所示的Session Options窗口。选择Connection，输入图中显示的信息。现在先选择密码认证。单击OK按钮，关闭窗口，然后在Connect窗口中单击Connect按钮。此时应该要求输入远程主机的登录口令，这样SSH登录过程就完成了。

一旦登录成功，就可以像一个普通终端程序那样使用SecureCRT。SSH的端到端加密机制对用户应该是透明的，实际上也正是透明的。

15.3 密钥管理

SecureCRT支持基于RSA密钥的公钥认证。既可以用内置的向导生成新密钥，也可以使用已有的SSH-1和SSH-2密钥。SecureCRT还可区分两种不同类型的SSH身份标识：全局标识和会话特定的标识。

15.3.1 RSA密钥生成向导

SecureCRT的RSA密钥生成向导可创建用于公钥认证的密钥对。向导的入口依次为Session Options窗口、Advanced按钮、General标签、Create Identity File。

注1： 在这里使用“Session”这个术语并不明智。通常“Session”都是指一个活动的SSH连接，而不是一组静态选项集合。

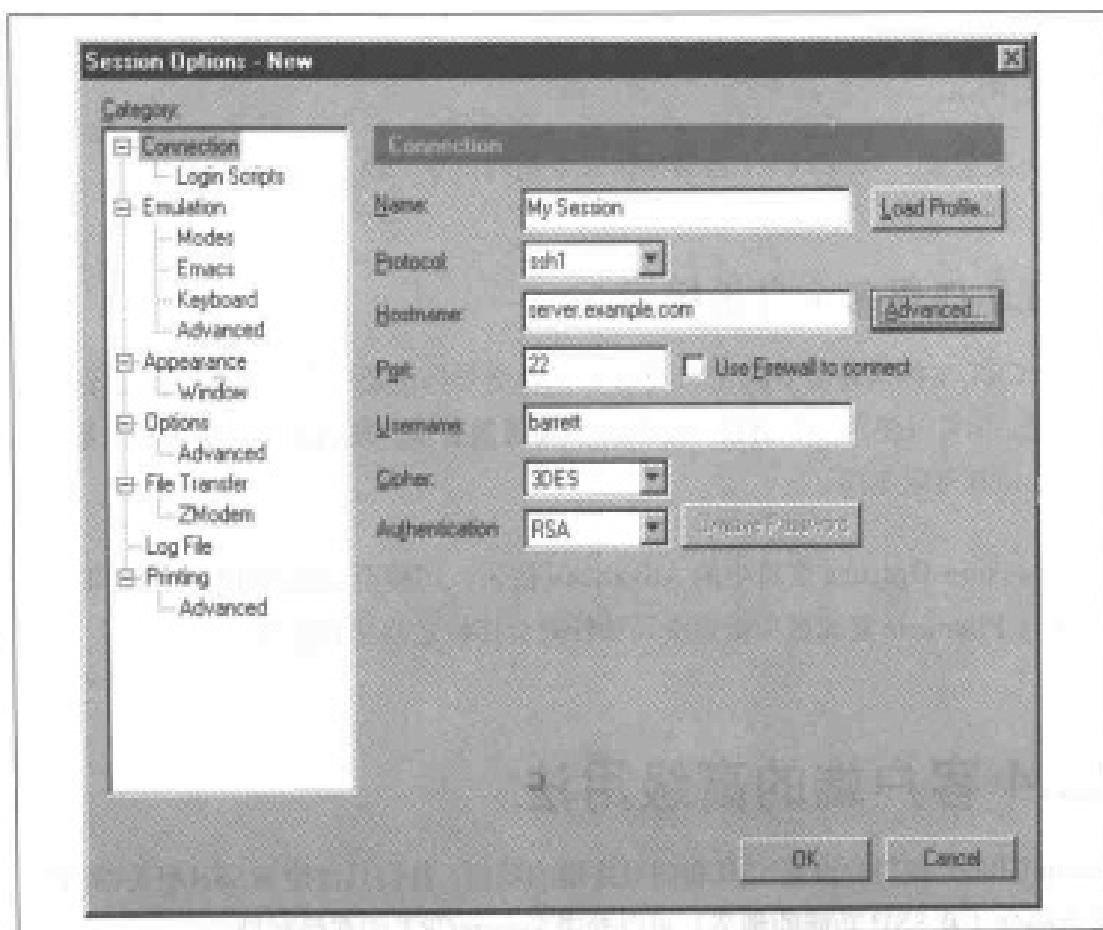


图 15-1：SecureCRT 的 Session Options 窗口

操作过程很直观，需要输入的只有口令和密钥位数，还要请你随便移动几下鼠标，生成一些随机信息。然后，RSA 密钥生成向导创建一对密钥，保存在两个文件中，私钥文件名是你起的，相应的公钥文件是在私钥文件名之后加.pub，这一点与 Unix 上是一样的。

密钥对生成好后，就得把公钥拷贝到 SSH 服务器所在的主机上，并将其存储于你账号的认证文件中。具体的方法是：

1. 用 SecureCRT 的密码认证方式登录 SSH 服务器。
2. 打开公钥文件，将密钥的全部内容拷贝到 Windows 剪贴板中。
3. 在 SSH 服务器上你的远程账号中安装公钥（从剪贴板中将密钥粘贴过来）。
4. 退出登录。

5. 在 Session Options 窗口中选择 Connection，把 Authentication 从 Password 改成 RSA。
6. 再次登录。SecureCRT 会请你输入公钥的口令，输入后就能登录了。

15.3.2 使用多个身份标识

SecureCRT 支持两种类型的 SSH 身份标识。全局标识是所有 SecureCRT 会话的缺省设置。你可以使用一个会话特定的标识覆盖缺省值；顾名思义，你定义的每个会话其标识都可以不同。

单击 Session Options 窗口中的 Advanced 按钮，切换至 General 标签页，即可在 Identity Filename 复选框中指定全局和针对会话特定的密钥文件。

15.4 客户端的高级用法

SecureCRT 允许用户设置 SSH 和图形终端的功能。我们只讨论与 SSH 有关的内容。其余部分（及 SSH 功能的细节）可以参考 SecureCRT 的在线帮助。

SecureCRT 的每一个会话都可以调用一组配置参数。它可以区分仅影响当前会话的参数和影响所有会话的全局参数。

启动 SSH 连接前可以修改会话参数。一些参数可以在连接建立之后修改，比如说远程 SSH 服务器的 name 选项。从 Options 菜单中选择 Session Options，或单击工具栏上的 Properties 按钮，都可以打开 Session Options 窗口（见图 15-1）。

15.4.1 强制性参数

要建立 SSH 连接，必须填写 Session Options 窗口中的所有 Connection 参数。包括：

Name

记录设置参数集的名字。它可以为任何值，缺省值是 SSH 服务器的名字。

Protocol

SSH-1 或 SSH-2。

Hostname

要连接的远程 SSH 服务器的主机名。

Port

SSH 连接的 TCP 端口号。实际上所有 SSH 客户端及服务器都在 22 端口上运行。

除非你要连接非标准服务器，否则不必修改这个值。[7.4.4.1]

用户名

你在远程 SSH 服务器上的用户名。如果使用公钥 (RSA) 认证，该用户名必须是一个拥有你公钥的账号。

Cipher

使用的加密算法。如果你不清楚应该使用那种加密算法，就用缺省的吧 (3DES)。

Authentication

你如何向 SSH 服务器证明你的身份。可以是 Password (使用远程登录密码)、RSA (公钥) 或 TIS。[15.4.3] SecureCRT 不支持可信主机认证。

15.4.2 数据压缩

SecureCRT 可对 SSH 连接上传输的数据透明地进行压缩和解压。这提高了连接使用速度。[7.4.11]

依次选择 Session Options 窗口、Connection、Advanced 按钮和 General。选中复选框 “Use Compression” 即启用数据压缩。你还可选择压缩等级 (Compression Level)。此项功能与 SSH1 的 CompressionLevel 一样。该值越高，压缩效果越好，不过 CPU 的负载也越重，会使你的计算机变慢。

15.4.3 TIS 认证

SecureCRT 可以使用 TIS (Trusted Information System 公司) 的 Gauntlet 防火墙工具包对用户进行认证。[5.5.1.8] 这可以通过在 Session Options 窗口中的 Connection 中把 Authentication 设置成 TIS 来实现。

15.4.4 使用防火墙

SecureCRT 可使连接穿越多种类型的防火墙，如 SSH1 和 SSH2 服务器支持的 SOCKS4 及 SOCKS5 防火墙。这只需在 Global Options 的 Firewall 中填入所要求的条目，包括防火墙的主机名或 IP 地址及要连接的 TCP 端口号即可。

15.5 转发

SecureCRT 支持 SSH 的转发功能（第九章），即其他网络连接可以从 SSH 中通过，同时进行加密。这也称为隧道方式，因为 SSH 给另一个连接提供了一条安全的隧道。SecureCRT 同时支持 TCP 端口转发和 X 转发。

15.5.1 端口转发

端口转发可利用一个 SSH 连接，对任意 TCP 连接透明地进行加密和路由选择。^[9.2]这样，不安全的 TCP 连接，如 Telnet、IMAP 或 NNTP（Usenet 新闻），就变得安全了。SecureCRT 支持本地端口转发，即本地的 SSH 客户端（SecureCRT）将连接转发到某台远程 SSH 服务器上。

你创建的每一个 SecureCRT 会话都可能有各自不同的端口转发设置。如果你已经连接到某台特定的远程主机上，就必须先关闭连接，然后打开 Session Options 窗口，才能设置转发。然后单击 Advanced 按钮，选择 Port Forwarding，就看到创建端口转发的界面（参看图 15-2）。

创建新转发时，首先单击 New 按钮。然后填写运行有你要使用的 TCP 服务（IMAP、NNTP 等）的那台远程主机的名字、该远程服务器的端口号和一个用于转发的本地端口号（在你的 PC 上）。理论上后一个端口号可以是任何值，但按照惯例一般用 1024 及以上的数字。同时，所选端口号不能已被本地其他 SSH 客户端使用。

设置好之后，单击 Save 按钮将这个转发保存下来。接着重新打开 SSH 连接，此时，只要连接还存在，你指定的端口就会一直被转发。

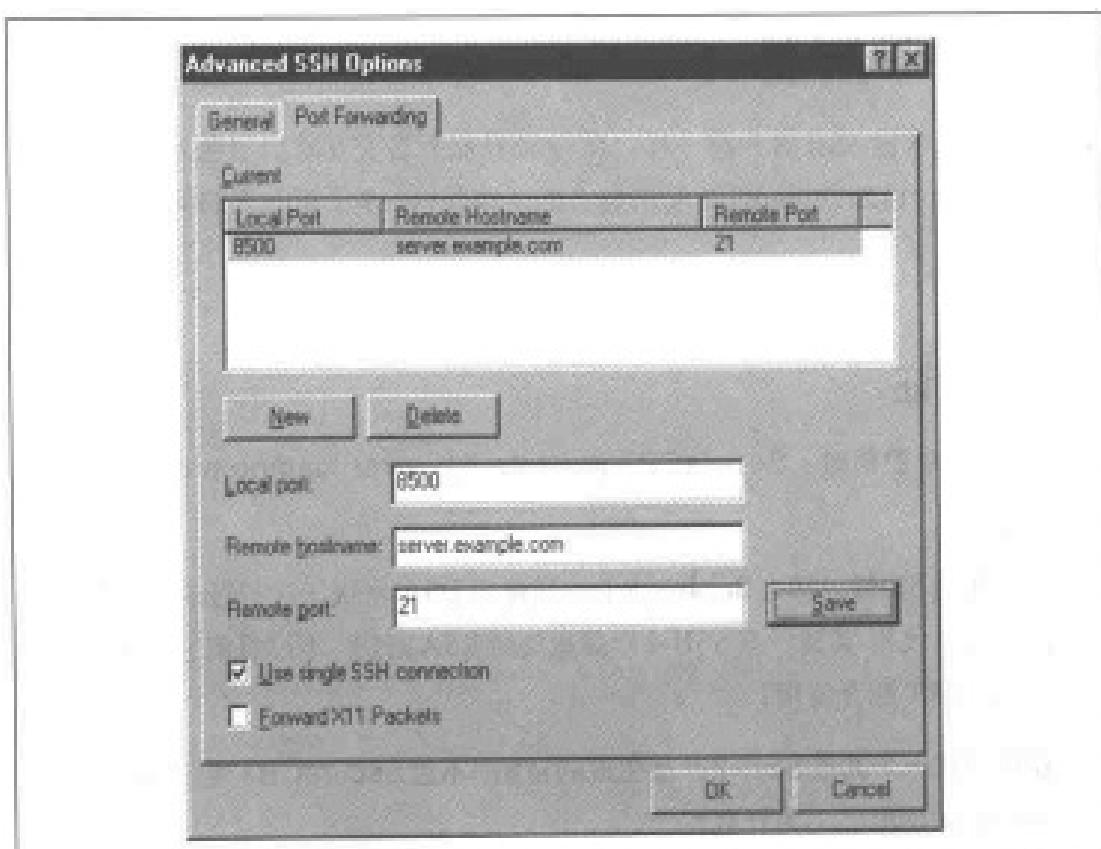


图 15-2：SecureCRT 端口转发

15.5.2 X 转发

X Window 系统是 Unix 机上最流行的窗口软件。如果你想运行远程 X 客户端，而在你自己的 PC 上打开窗口，你需要：

- 一台远程主机，上面有 X 客户端程序，并运行 SSH 服务器。
- 你的 PC 上有一个 Windows 下的 X 服务器，如 Hummingbird 的 eXceed。

SSH 通过一种称为 X 转发的机制来保护你的 X 连接。^[9.3] 开启 X 转发的操作在 SecureCRT 上非常简单。你只需打开 Session Option 窗口，单击 Advanced 按钮，选择 Port Forwarding，然后在“Forward X11 Packets”复选框前打勾即可。

为实现用 SSH 转发保护 X 连接的目的，你首先应该运行 SecureCRT，建立起到 SSH 服务器的安全终端连接，然后运行你 PC 上的 X 服务器，并关闭 XDM 之类的登录功能。现在只需在服务器上调用 X 客户端即可。

15.6 疑难解答

SecureCRT 与其他 SSH 客户端一样，在与 SSH 服务器交互时会出现无法预期的问题。本节我们讨论 SecureCRT 本身所特有的一些问题，至于更一般的问题请参看第十二章。

15.6.1 认证

问：我用 RSA 密钥时，SecureCRT 说 “Internal error loading private key.”，为什么？

答：如果使用 SSH2，那么 SecureCRT 就只接受 DSA 密钥（这与当前的 SSH-2 标准草案一致）。其他一些 SSH-2 产品也支持 RSA 密钥；不过目前它们尚不能与 SecureCRT 配合使用。

问：我想加载一个其他 SSH-2 产品生成的密钥，不过 SecureCRT 说 “The private key is corrupt.”，为什么？

答：SSH 协议标准草案仅仅指定了密钥在 SSH 会话过程应该如何描述，而并未说明存储密钥的文件格式。这导致开发者随意创造各种不同的、互不兼容的密钥格式，给使用者增加了负担。SSH1 与 SSH2 都是如此，不过 SSH1 中问题少些。这种情形也许会改变，所以我们也只能说：“请与供应商联系”。

问：我输入口令之后看到一个对话框“SSH_SMSG_FAILURE: invalid SSH state.”，随后我还没有登录进去会话就断开了，这是为什么？

答：没有分配伪 tty 终端。可能你远程账号的 *authorized_keys* 文件中设置了 no-pty 选项。[8.2.9]

问：SSH-2 认证失败，返回信息是 “SecureCRT is disconnecting from the SSH server for the following reason: reason code 2”，这是为什么？

答：本书出版时，来自不同厂商的若干 SSH-2 客户端和服务器由于所用 SSH-2 标准草案版本的不同而不能一起工作。希望你看到这里时问题已经解决。目前你还是得确认一下，你在 SecureCRT 的 Properties 窗口的 Connection 中所选的 SSH 服务器是否正确。*telnet* 到服务器的 SSH 端口（通常为 22），阅读出现的版本

字符串，即可确定你要连接的服务器的类型。具体方法是，在MS-DOS模式下输入 *telnet* 命令：

```
telnet server.example.com 22
```

系统会出现一个窗口，其中包含了以下的响应信息：

```
SSH-1.99-2.0.13 F-SECURE SSH
```

表明（本例中）使用了 F-Secure SSH 服务器，版本号 2.0.13。

15.6.2 转发

问：我用不了端口转发，返回的信息说端口已使用，这是为什么？

答：你是否还打开了一个SecureCRT窗口？这个窗口里也是用同样的端口转发设置进行连接的吗？不能同时建立两个连接来转发同一本地端口。解决这个问题的方法是：复制第一个会话的设置，修改本地端口号，另存为第二个会话。现在，你就能同时使用两个会话了。

15.7 小结

我们使用 SecureCRT 已经有超过一年的时间了，我们发现这是一个可靠、稳定、功能强大的产品，对用户的技术支持也很好。它的缺点是不支持代理和 *scp*。

本章内容

- 安装与安装
- 远端的基本用途
- 密钥管理
- 客户端的高级用途
- 域名
- 调用命令
- 小结

第十六章

F-Secure SSH Client (Windows、Macintosh)

F-Secure Corporation 是 Unix 上商用 SSH1 和 SSH2 的发行商，同时也为 Windows 和 Macintosh 平台制作 SSH 客户端软件。^[1.5] 它的客户端产品线有个很好听的名字：F-Secure SSH Client，这是一个 VT100 风格的窗口式终端程序，可以使用 SSH-1 和 SSH-2 两种 SSH 协议登录。

Windows 平台上我们评估的是 4.1 版，这一个产品就可以同时支持两种协议。而更早的版本使用不同的产品分别支持 SSH-1 和 SSH-2。Macintosh 平台上我们评估了 1.0 (SSH-1) 和 2.0 (SSH-2)，不过直到本书出版时，这两个软件在我们的系统中的运行情况依然不可靠，所以本书对其不作讨论。在测试过程中，我们遇到了一些无法解释的死机现象，这可能是我们的 Macintosh 配置造成的原因，也可能是 F-Secure 本身的原因。不过 Windows 客户端就不存在这些问题。

F-Secure SSH Client 的功能相当丰富。对于 SSH-1，它支持远程登录、端口转发、X 转发、密钥对生成以及一些终端程序的特性。对于 SSH-2，它还提供 *scp2* 和 *sftp2*，可以支持安全文件拷贝。

16.1 获取与安装

F-Secure 可以在线购买：

<http://www.f-secure.com/>

此处也有免费测试版，可用传统的Windows安装程序完成安装：只需选择目标路径，并作其他几个简单的决定即可。

16.2 客户端的基本用法

客户端的行为是在Properties窗口中设置的：选择Edit菜单中的Properties项（见图16-1）。选中Connection，输入如图16-1所示的信息。单击OK按钮关闭窗口，然后选择File菜单中的Connect。目前程序还不知道任何公钥，因此使用密码认证。如果一切顺利，远程服务器会让你输入密码，然后，你就成功地通过SSH登录了。

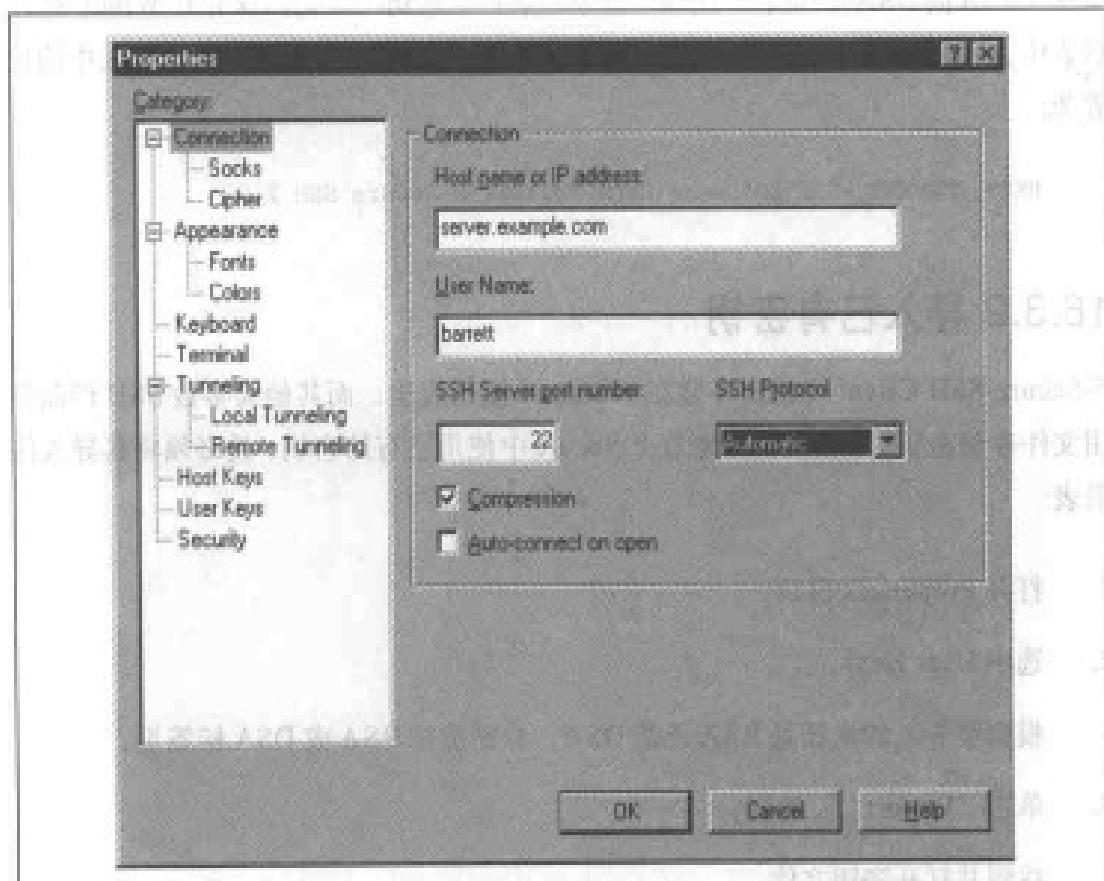


图16-1：F-Secure SSH Client的Connection Properties窗口

登录一旦成功之后，该程序的操作与普通终端程序相同。SSH的端到端加密操作对用户而言是透明的，而且也应该是透明的。

用户可以创建一个会话文件来保存喜欢的设置。会话文件的后缀名为`.ssh`，可通过File菜单中的Save和Open命令保存和打开（惊喜吧）。

16.3 密钥管理

F-Secure SSH Client 支持使用 RSA 或 DSA 密钥的公钥认证。可用内建的 Key Generation Wizard 生成密钥，也可使用现有的 SSH-1 或 SSH-2 密钥。

16.3.1 生成密钥

Key Generation Wizard 可从 Tools 菜单中打开。这个向导可让用户选择密钥的生成算法（RSA 或 DSA）、位数、注释、口令和名称。密钥生成之后保存在 Windows 注册表中，可在 Properties 窗口的 User Keys 里找到。密钥在 Windows 注册表中的位置为：

```
HKEY_CURRENT_USER\Software\Data Fellows\F-Secure SSH 2.0
```

16.3.2 导入已有密钥

F-Secure SSH Client 将密钥存储在 Windows 注册表里。而其他大多数 SSH 产品是用文件存储密钥的，因此如果想在 F-Secure 中使用已有的密钥，就必须将其导入注册表：

1. 打开 Properties 窗口。
2. 选中 User Keys。
3. 根据要导入的密钥是 RSA 还是 DSA，分别选择 RSA 或 DSA 标签页。
4. 单击“Import...”按钮。
5. 找到并打开密钥文件。
6. 输入该（SSH-1 格式的）密钥的口令。

现在，密钥已导入到 F-Secure 中，可以使用了。

16.3.3 安装公钥

F-Secure SSH Client 还单独为 SSH-2 公钥提供了 Key Registration Wizard，可向远程账号所在的 SSH-2 服务器自动上传并安装公钥。这可太好了！此过程用密码认证方式通过 SSH-2 连接到远程账号上，因此当然是安全的。

SSH-1 的公钥必须手工在服务器上安装。先用密码认证连接远程主机，然后打开 Properties 窗口，选择 User Keys。此处有两种选择：

- 用“Export...”按钮将公钥导出到一个文件中，然后把这个文件传到远程服务器主机上，接着把文件的内容拷贝至 *authorized_keys* 中。
- 单击 Copy To Clipboard，将公钥拷贝至 Windows 剪贴板，然后粘贴到远程的 *authorized_keys* 文件中。

16.3.4 使用密钥

F-Secure SSH Client 与多数 Unix 上的 SSH 产品的不同之处在于，它不让用户指定哪个会话使用哪个密钥，而是依次尝试每个密钥。如果有一个密钥可以和服务器上的公钥匹配，就会提示输入口令。如果不用 F-Secure 选择的密钥，而想使用其他密钥，可以按 Escape 键或者单击 Cancel 按钮，F-Secure 就会尝试下一个密钥。如果所有的密钥都不匹配，程序将转用密码认证。

16.4 客户端的高级用法

要建立 SSH 连接，必须在 Properties 窗口中填写下列 Connection 字段。包括：

Host Name

欲连接的远程 SSH 服务器主机的名字。

User Name

你在远程 SSH 服务器上的用户名。如果使用公钥 (RSA) 认证，那么这个用户名必须属于拥有你公钥的某个账号。

Port Number

SSH连接使用的TCP端口号。实际上所有SSH客户端及服务器都是在22端口上运行的。除非你打算连接非标准SSH服务器，否则不必修改此值。[7.4.4.1]

SSH Protocol

你可能要选SSH-1或SSH-2，要么就选Automatic，这样程序就可以根据服务器的响应替你做出选择。

你可以随意选择加密算法及认证方式。在Properties窗口的Cipher中选择你允许客户端使用的加密算法集。（对大多数应用而言缺省设置就足够了。）SSH服务器会与客户端协商，选用两者都支持的加密算法。

认证方式可以是公钥或密码方式，这可以在login窗口中进行选择。程序会自动尝试用User Keys中的每一个密钥进行认证。[16.3.4]

16.4.1 数据压缩

F-Secure SSH可以对SSH连接上传输的数据透明地进行压缩和解压，这可以加速连接的传输。[7.4.11]

在Properties窗口中，选择Connection，选中Compression复选框。F-Secure与SSH1一样无法设置不同的压缩等级。

16.4.2 用详细模式调试

你的SSH会话不能像预期的那样工作了吗？打开详细模式，这样会话运行时窗口上就能打印出状态消息。这有助于问题的定位和解决。

在Properties窗口中选择Appearance，再选中Verbos Mode。下次连接时就可看到这样的信息：

```
debug: connecting ...
debug: addresses 219.243.169.50
debug: Registered connecting socket: 12
debug: Connection still in progress
debug: Marked name resolver 1 killed
debug: Replaced connected socket object 12 with a stream
```

详细模式与 Unix 的 SSH 中相应的部分很类似。[7.4.15] 这是诊断连接问题时不可或缺的工具。

16.4.3 SOCKS 代理服务器

F-Secure SSH Client 支持 SOCKS 4 代理服务器。[4.1.5.8] 在 Properties 窗口中选择 Socks，填写代理服务器的主机名或 IP 地址，以及端口号（通常 SOCKS 端口为 1080）。

16.4.4 接受主机密钥

每个 SSH 服务器都有一个主机密钥唯一标识自己的身份，这样 SSH 客户端才能证实和它对话的是实际的服务器，而不是冒名顶替的。[2.3.1] F-Secure SSH 客户端会记录它遇到的所有主机密钥。这些密钥保存在 Windows 注册表中。

如果想让 F-Secure SSH Client 忽略以前没见过的主机密钥，就得在 Properties 窗口的 Security 中选中与此相关的一个复选框。

16.4.5 附加的安全特性

通常 F-Secure SSH Client 记录主机名、用户名、文件名以及它遇到的终端输入输出信息。如果想从程序中清除这些信息（比方说，不想让第三方在你的计算机上看到这些内容），就在 Properties 中选择 Security。窗口中的按钮可实现这一删除功能。

16.4.6 用 SFTP 实现安全文件传输

程序中还有一个图形化的文件传输工具：F-Secure SSH FTP。如果你用过其他图形化的 FTP 客户端软件，那么对这个程序的界面应该很熟悉，只有 SSH 认证设置那一部分略有不同。该程序的文档在联机帮助里，不过我们还是想在这儿提一下。

16.4.7 命令行工具

F-Secure SSH Client 既有一个图形化的终端程序，也有支持 SSH-2 协议的命令行

程序。其中包括`ssh2`、`scp2`和`sftp2`。这些程序与SSH2的对应程序几乎完全一样（见第二章），但是以下几点不同：

- 不支持某些 Unix 命令行选项。输入程序名（如`ssh2`）即可看到当前支持的选项列表。
- 不支持密钥文件；这些程序与 F-Secure SSH Client 一样，是从 Windows 注册表中读取密钥的。

命令行程序在脚本、批处理以及在服务器主机上执行远程命令几方面都非常有用：

```
C:\> ssh2 server.example.com mycommand
```

16.5 转发

F-Secure SSH Client 支持转发（第九章），即其他网络连接可以从 SSH 中通过，同时由 SSH 对其进行加密。这也称为隧道方式，因为 SSH 给另一个连接提供了一条安全的隧道。F-Secure SSH Client 同时支持（本地的和远程的）TCP 端口转发和 X 转发。

16.5.1 端口转发

每个 F-Secure SSH Client 配置都可以有多个不同的端口转发设置。要使设置的转发对特定的远程主机生效，应该先关闭到该主机的连接（如果已经连接上了的话），再打开 Properties 窗口，切换至 Tunneling。选择 Local Tunneling，即可建立本地转发（参看图 16-2）；选择 Remote Tunneling 即可建立远程转发。[9.2.3] 每种情况下需要输入的信息都相似：

Source Port:

源端口号。

Destination Host:

目的主机。

Destination Port:

目的端口号。

Application to Start:

使用该端口转发时启动的外部应用程序。

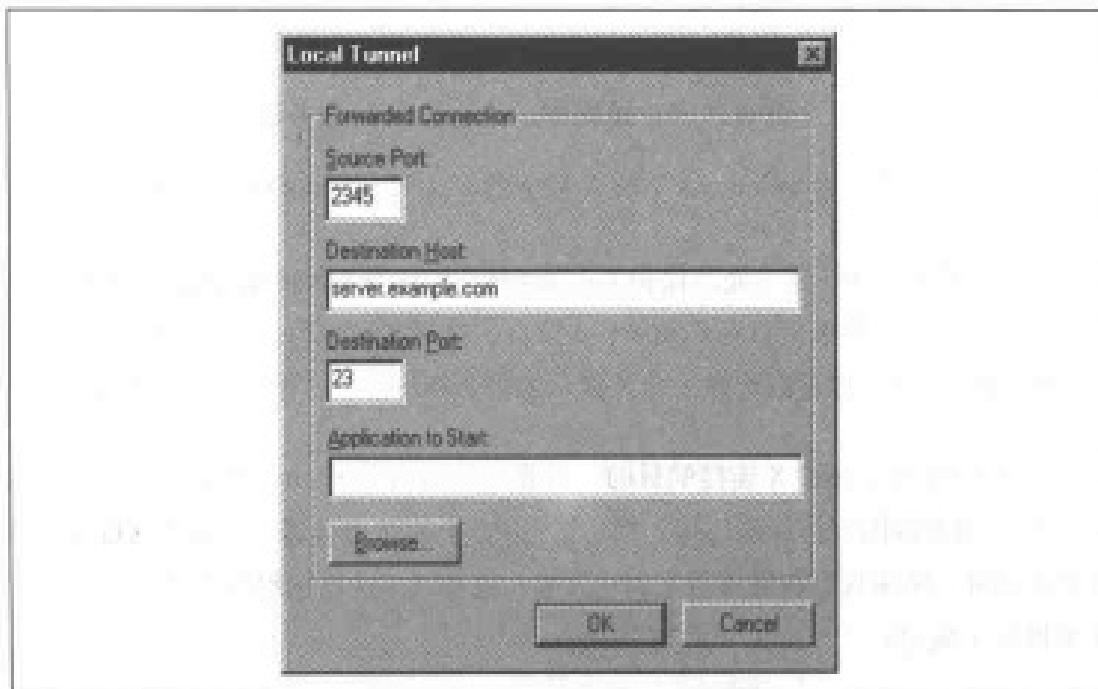


图 16-2: F-Secure SSH Client 本地端口转发选项

例如，若要使用 SSH 隧道方式使 telnet 连接 *server.example.com* (TCP 端口 23)，则必须指定：

Source port: 8500 (任意随机口号)

Destination Host: *server.example.com*

Destination Port: 23

Application to Start: c:\windows\telnet.exe

设置好之后，重新打开 SSH 连接，此后该连接就一直使用端口转发。

请注意，F-Secure SSH Client 禁止远程连接访问本地转发端口。这一安全特性与 "GatewayPorts no" 类似。[9.2.1.1]

16.5.2 X 转发

X Window 系统是 Unix 机上最流行的窗口软件。如果想运行远程 X 客户端，而在你自己的 PC 上显示窗口，需要：

- 一台远程主机，上面有 X 客户端程序，并运行 SSH 服务器。
- 你的 PC 上有一个 Windows 下的 X 服务器，如 Hummingbird 的 eXceed。

SSH 通过一种称为 X 转发的处理保护你的 X 连接。^[9.3] 开启 X 转发的操作在 F-Secure SSH Client 上非常简单：打开 Properties 窗口，选择 Tunneling，选中 Enable X11 Tunneling 复选框。你可以选择一个 X 显示编号，SSH 会话过程中也能修改这个值。

为实现用 SSH 转发保护 X 连接的目的，首先应该运行 F-Secure SSH Client，建立起到 SSH 服务器的安全终端连接。然后运行 PC 上的 X 服务器，并关闭 XDM 之类的登录功能。现在只需在服务器上调用 X 客户端即可，这样远程的 X 窗口就可以在本地机器上显示。

16.6 疑难解答

F-Secure SSH Client 与其他 SSH 客户端一样，在与 SSH 服务器交互时会出现无法预期的问题。本节我们将讨论 F-Secure SSH Client 本身特有的一些问题，至于更一般的问题请参看第十二章。

问：为什么 F-Secure SSH for Windows 滚屏的速度那么慢呢？

答：本书出版时 F-Secure 的滚屏速度的确非常之慢，不过有一个跳滚（jump scrolling）功能，缺省是关闭的，启用之后可在一定程度上提高滚屏速度，不过它只能通过 F-Secure 的键盘映射机制启用。下面是在给定会话文件中打开跳滚的步骤：

- a. 在 F-Secure 安装目录下找到 *Keymap.map*。
- b. 以编辑方式打开这个文件，找到有“enable fast-scroll-mapping”的那行。
- c. 去掉这一行前面所有的注释标志符“#”。
- d. 保存文件然后关闭。

- e. 在 Edit 菜单中选择 Properties。
- f. 在 Properties 窗口中选择 Keyboard。
- g. 在 Map Files 下的 Keyboard 输入框中，输入编辑过的那个 *Keymap.map* 文件的路径。
- h. 单击 OK 按钮。

现在，用 Ctrl-Alt-F3 组合键可在平滑滚动和跳滚两种状态之间切换。只要在 F-Secure SSH 会话中按下这个组合键，即可提高滚屏速度。

问：为什么看到这样的出错消息：“Warning: Remote host failed or refused to allocate a pseudo tty”？

答：服务器的 SSH-1 *authorized_keys* 文件中可能对相应的公钥指定了 no-pty 选项。

问：我无法进行端口转发，我得到的消息是该端口已使用，为什么？

答：你是否还打开了一个 F-Secure 窗口？这个窗口里也是用同样的端口转发设置进行连接的吗？不能同时建立两个连接来转发同一本地端口。解决这个问题的方法是：复制第一个会话的设置，修改本地端口号，另存为第二个会话。现在，你就能同时使用两个会话了。

问：我试用了 Key Registration Wizard，但是不行。Wizard 显示“Disconnected, connection lost”，为什么？

答：首先查看主机名、远程用户名和远程密码是否正确。如果确信没有出错，就单击 Advanced 按钮，检查一下那里面的信息。SSH-2 服务器是在缺省端口 22 上还是在其他端口上运行？你填的 SSH2 目录和认证文件名与服务器上的一致吗？

另一种技术性更强的可能是服务器 /etc/sshd2_config 文件中缺少 *sftp-server* 子系统设置项，或信息不完整。F-Secure 推荐的方式是写明 *sftp-server2* 的完整且符合规范的路径，不然 SSH 服务器可能无法定位它。Key Registration Wizard 用 SFTP 协议传输密钥文件。服务器的系统管理员必须据此修改设置。

问：我如何才能不看到 F-Secure SSH2 启动时显示的欢迎信息？

答：在 Windows 的注册表中修改：

HKEY_CURRENT_USER\Software\DaFa Fellows\F-Secure SSH 2.0\InT\Settings>ShowSplash

缺省值是 yes；将其设为 no。下次运行程序的时候就看不到欢迎窗口了。（这一点文档中没有提到。）

问：运行 F-Secure 时弹出一个 Logon Information 窗口，如何去掉？因为我用的是公钥认证，没必要出现这个窗口。

答：在 Properties 窗口中选择 Connection，选中 Auto-Connect On Open 即可。

16.7 小结

从总体上看，F-Secure SSH Client for Windows 是一个运行良好、稳定可靠的产品。目前的版本确实存在一些长期性的弱点：没有代理、基于 SSH-1 的产品不支持安全文件拷贝（基于 SSH-2 的产品有 *sftp*）、文档不完整（至少没提到键盘映射文件），以及无法在 Preference 窗口中开启跳滚功能。不过我们一年多以来已使用过多个版本，发现它确实很可靠。F-Secure SSH 目前仍在积极的开发之中，因此可能会突破现有的局限。

本章内容

- 安装与安装
- 基本客户端应用
- 配置服务器
- 小结

第十七章

NiftyTelnet SSH (Macintosh)

Jonas Walldén 的 NiftyTelnet SSH 是 Macintosh 上的最小 SSH 客户端，可以免费发布。其基础是 Chris Newman 的图形化终端 NiftyTelnet，在其中加入了对 SSH-1 的支持。NiftyTelnet SSH 支持远程登录和安全文件拷贝。如果打开多个终端窗口，它还能帮助用户记住公钥的口令（即在内存中缓冲）。不过这可不是代理。

NiftyTelnet SSH 最大的优点是免费，且性能很好。缺点是不支持任何形式的转发，无法生成 SSH 密钥对。要使用公钥认证，必须由其他 SSH 程序（如 SSH1 的 *ssh-keygen1*）为账号生成密钥对。

我们对 NiftyTelnet SSH 的讨论基于 1.1 R3 版本。

17.1 获取与安装

可以从下面的网址下载 NiftyTelnet SSH：

<http://www.lysator.liu.se/~jonasw/freeware/niftyssh/>

然后使用 *Stuffit Expander* 将其解到某个目录中。也可以将整个目录拷贝到 Macintosh 中的任何位置。

NiftyTelnet SSH 首次运行时，会显示如图 17-1 所示的 New Connection 对话框。单

击 New 按钮，设置一个 SSH 客户端/服务器连接。图 17-2 显示了重要的 SSH 配置字段：Hostname、Protocol 和 RSA Key File。Hostname 中应该填写运行 SSH 服务器的那台远程主机的名字。Protocol 应该设置为某种 SSH 加密算法（DES、3DES 或 Blowfish）。如果打算用密码认证，就要把 RSA Key File 那一行空着。否则，如果 Mac 中已安装了一个私钥文件，就填入那个文件的位置。这里必须包含该文件的完整路径，文件夹间的分隔符是冒号“：“。举例来说，如果要依次打开磁盘 MyDisk，文件夹 SSH 和 NiftyTelnet 才能找到密钥文件 Identity，那么就应该输入：

```
MyDisk:SSH:NiftyTelnet:Identity
```

设置完成后，用密码认证方式连接远程主机。将公钥文件拷贝到远程主机上，退出登录，然后用公钥认证重新连接。

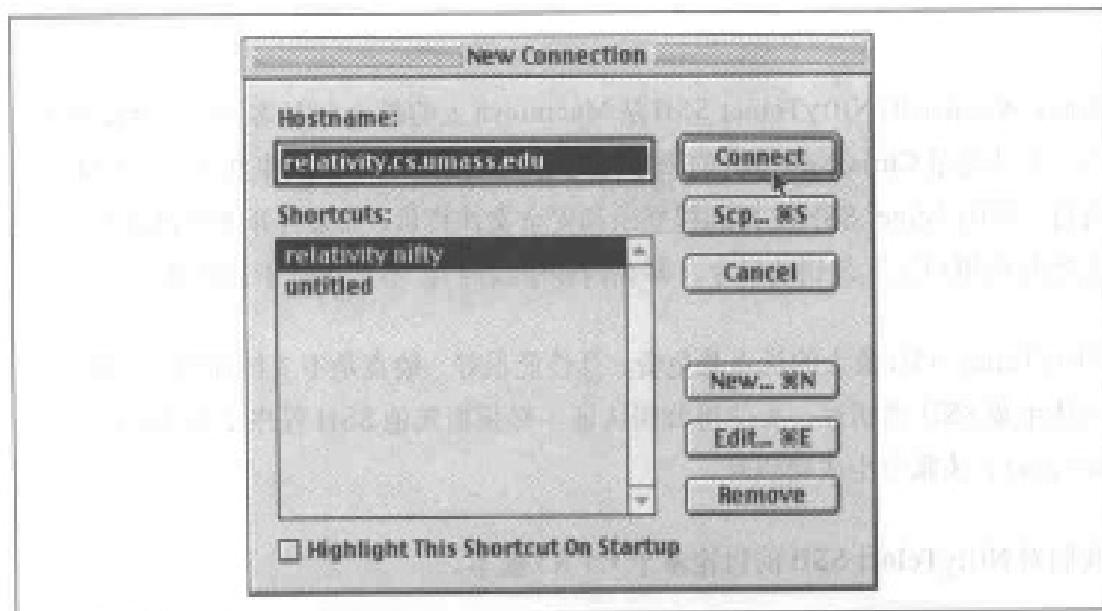


图 17-1：NiftyTelnet 的 New Connection 对话框

17.2 基本客户端应用

NiftyTelnet SSH 起源于 Macintosh 上的一个 Telnet 工具 NiftyTelnet，然后另外一个程序员为其增加了 SSH 支持，因此它的大多数可配置参数都与 Telnet 有关，对此我们就不再进行讨论了，现在只讨论那些 SSH 特有的参数。

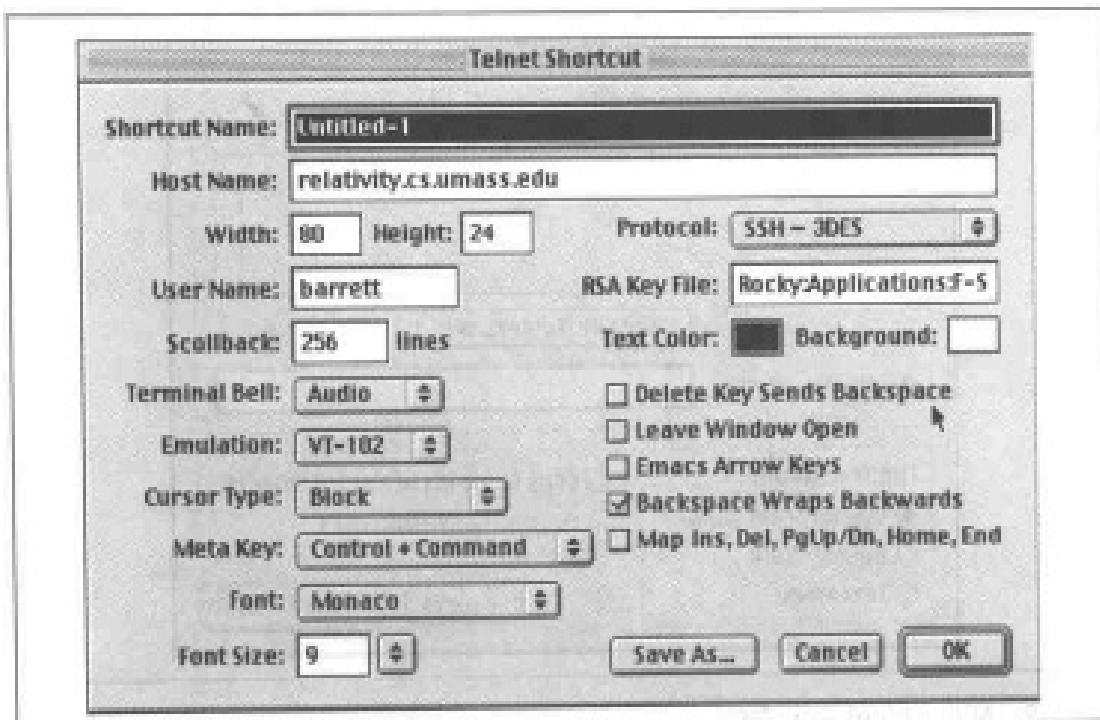


图 17-2: NiftyTelnet SSH 的设置窗口

17.2.1 认证

如图 17-2 所示, 与 SSH 有关的参数只有加密算法 (“Protocol” 设置的值) 和私钥文件的路径 (“RSA Key File” 设置的值), 缺省认证方式为公钥认证, 不过如果没有密钥文件, 认证就会失败, 并降级使用密码认证。

惟一需要点儿技巧的地方是私钥文件的路径。这里只能输入, 不能在 Macintosh 的文件选择器 (file selector) 中选择。(17.3) 是如何设置对连接到远程计算机的私钥文件的?

17.2.2 Scp

单击 New Connection 对话框中的 Scp 按钮, 就可以在 Mac 和某台远程计算机之间执行安全的文件和目录拷贝操作。此项功能与 SSH1 的 scp1 客户端程序很相似, 不过前者使用了图形界面 (见图 17-3)。

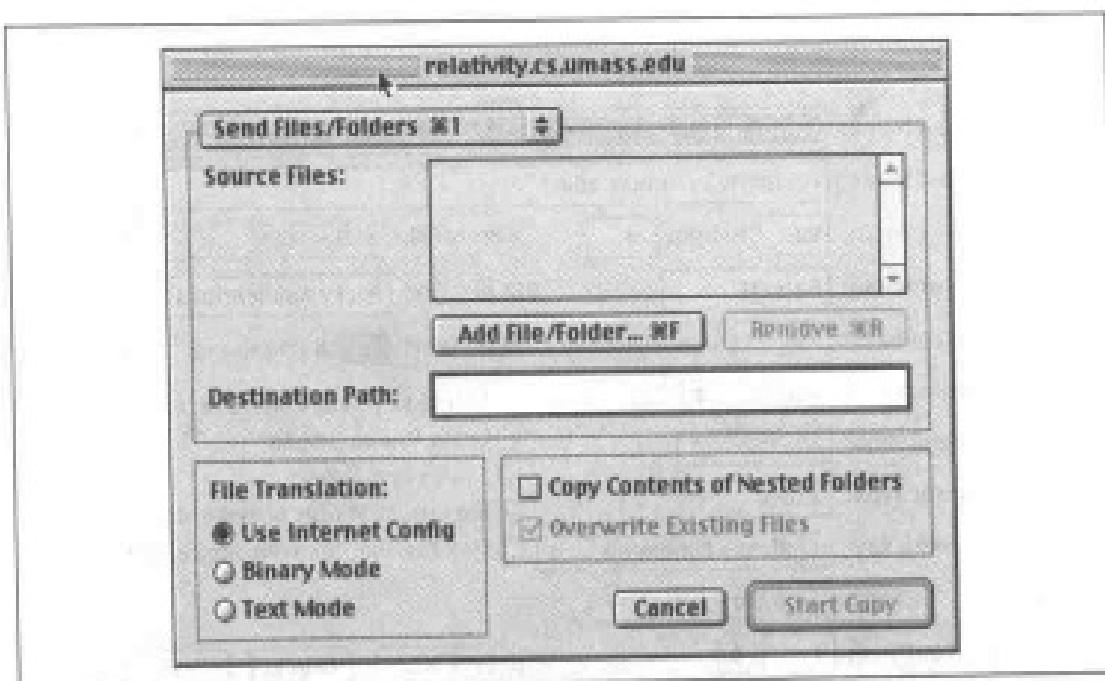


图 17-3: NiftyTelnet Scp 窗口

本地文件及文件夹可以用浏览的方式进行选择，而远程文件及文件夹就必须手工输入。如果用过 Fetch 那样的 Mac FTP 工具，现在这个界面也许会显得很简陋。不过它还是能工作的，而且如果让 NiftyTelnet SSH 记住密码的话，就不用在每传输一个文件时重复输入了。

17.2.3 主机密钥

每个 SSH 服务器都有一个主机密钥来表示自己的身份，这样诸如 NiftyTelnet SSH 之类的客户端就能证实和它对话的是实际的服务器，而不是冒名顶替的。[\[2.3.1\]](#)

NiftyTelnet SSH 客户端会记录它遇到的所有主机密钥。这些密钥保存在 Macintosh System 文件夹的 *Preferences* 中，文件名为 *NiftyTelnet SSH Known Hosts*。这个文件的格式与 SSH1 已知名主机列表文件的格式相同。[\[3.5.2.1\]](#)

17.3 疑难解答

问：我想使用公钥认证。用 NiftyTelnet 怎么生成密钥对呢？

答：这没法实现。生成密钥对必须得用其他SSH程序，比如Unix上SSH1中的*ssh-keygen1*。[\[2.4.2\]](#)

问：我应该在“RSA Key File”那个输入框里填什么？

答：私钥文件的完整路径，包括前面每一层文件夹的名字，中间以冒号分隔。比如说，如果要依次打开磁盘*MyDisk*、文件夹*SSH*和*NiftyTelnet*才能找到密钥文件*Identity*，那么就应该输入：

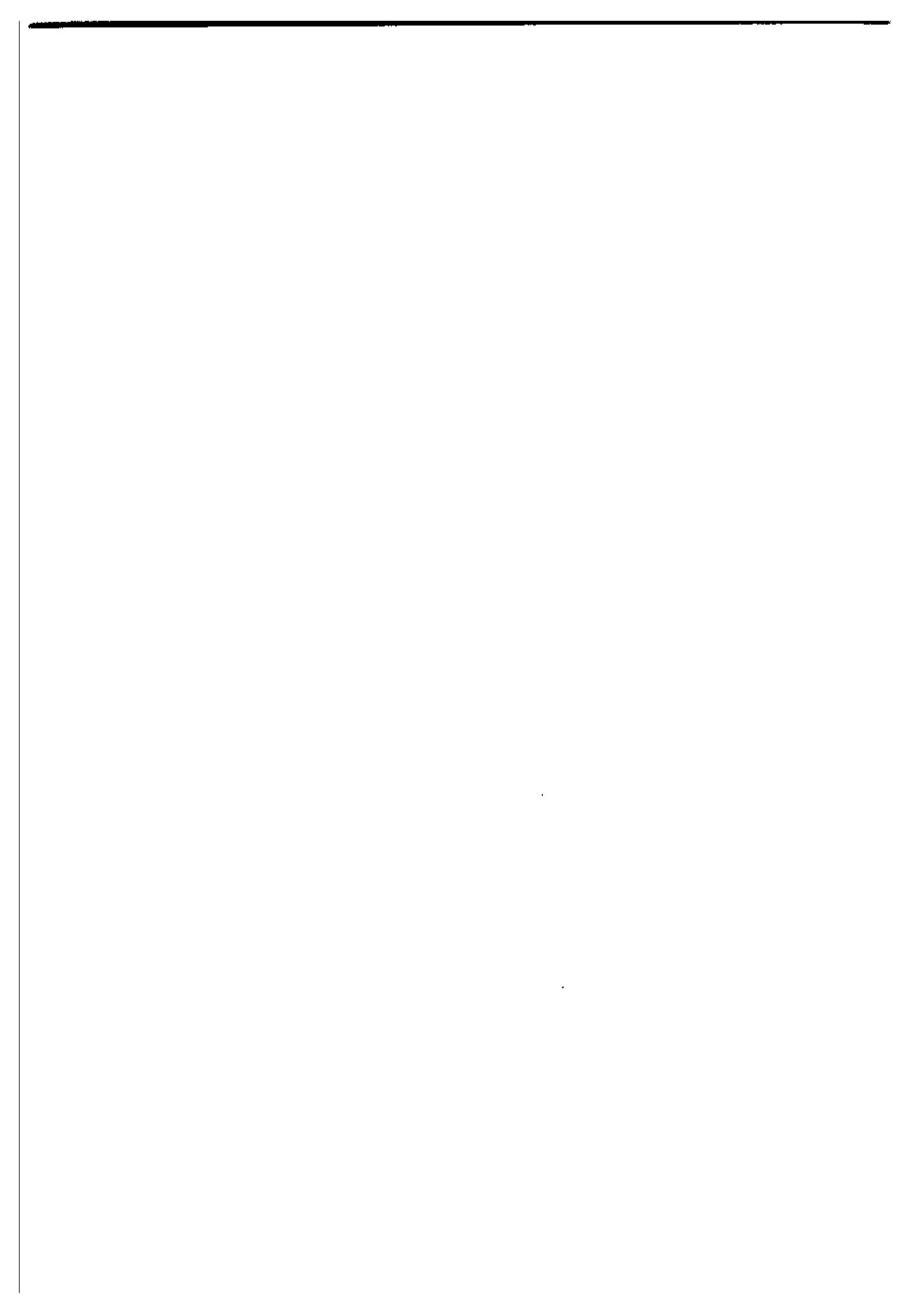
`MyDisk:SSH:NiftyTelnet:Identity`

问：我用不了Scp功能。我传输文件时，会出现一个窗口，上面写着“File: Waiting For Connection”，过几秒之后就消失了。文件也没有传过去。

答：据软件的作者说，如果与NiftyTelnet SSH通信的是某种较老的SSH服务器，有时就会出现这种情况。据说*sshd* 1.2.25以上的版本没有问题。我们最初是在一台1.2.27服务器上看到这一问题的，但是后来它自己消失了，因此无法找出原因。

17.4 小结

在Macintosh上的SSH客户端程序中，NiftyTelnet SSH是一个很好的选择。它支持*scp*，这在PC及Mac机的SSH-1实现产品中很少见；它还能在程序退出之前记忆口令。换个角度来说，NiftyTelnet SSH是我们见过的功能最少的SSH软件，甚至不能生成密钥对。如果需要更多的功能，如端口转发和更多的设置选项，请考虑选用Mac版的F-Secure SSH Client（第十六章）。



附录一

SSH2 手册页， sshregex 部分

SSHREGEX(1)

SSH2

简介

本文档介绍了 *scp2* 和 *sftp2* 在文件名中使用的正则表达式（或称为模式）。

模式

转义字符为反斜线（“\”）。使用这个转义字符，我们可以对要使用的元字符进行转义，使用该字符的原意。

在下面的例子中，“E”和“F”都表示任意表达式，该表达式可以是一个模式，也可以是一个字符。

* 可以和由零个或多个字符组成的任意字符串匹配，其中的字符可以是除斜线（“/”）之外的任意字符。但是，如果一个字符串的第一个字符是小数点（“.”），或者该字符串中包含一个“/.”（斜线后面紧跟着一个小数点）子字符串，那么星号则不能与之匹配。即星号不能用来匹配第一个字符是小数点的文件名。

如果星号的前一个字符是斜线，或者用星号表示一个字符串的开始部分，那么星号就可以匹配小数点。

即星号（“*”）的作用和 UNIX shell 中的通常用法相同。

? 问号 ("?") 可以和除斜线 ("/") 之外的任意一个字符匹配。但是，如果一个字符串的第一个字符是问号，或者问号的前一个字符是斜线，那么问号就不能匹配小数点。

即问号的作用和 Unix shell 中的通常用法相同（至少在 ZSH 中就是如此，尽管不能匹配小数点的用法并非标准）。

/ 可以和任意由空字符或以斜线结尾的字符序列匹配。但是，该字符串中不能包含子字符串 "/."。这种用法模仿了 ZSH 独创的 “/” 结构。（显然 “**” 和 “*” 相同。）

E# 用作 Kleene star 操作，可以匹配 E 零次或多次。

E##

闭集，可以匹配 E 一次或多次。

(子表达式开始

) 子表达式结束

E|F

连接，可以匹配 E 或 F。如果 E 和 F 都可以匹配，则 E 优先。

[字符集开始。（请参看下文）。

字符集

字符集以 “[” 开始，以 “]” 结束，它不是 POSIX 标准的字符集标识符，后面不能立即跟上 “[”。

以下的字符集都具有特殊意义，如果要使用原意，需要使用转义字符进行转义：

- (减号)

范围操作符，后面不能立即跟上 “[”，否则就失去了特殊意义。

^ 或 !

如果紧跟在 “[” 之后，则表示注释：整个字符集都被注释掉。否则就是原意 “^”。

[:alnum:]

isalnum 返回真的字符所组成的字符集（请参看 ctype.h）。

[*:alpha:*]

`isalpha` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:cntrl:*]

`iscntrl` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:digit:*]

`isdigit` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:graph:*]

`isgraph` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:lower:*]

`islower` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:print:*]

`isprint` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:punct:*]

`ispunct` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:space:*]

`isspace` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:upper:*]

`isupper` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

[*:xdigit:*]

`isxdigit` 返回真的字符所组成的字符集（请参看 *ctype.h*）。

例子

`[[:xdigit:]XY]`通常等价于
`[0123456789ABCDEFabcdefXY]`。

作者

SSH Communications Security 公司。

更多信息，请参看 <http://www.ssh.com/>。

还可以参看

scp2(1), sftp2(1)

附录二

SSH 快速参考

图例

标记	含义
✓	包括: 支持或包含某项功能
1	仅对 SSH-1 有效, 不包括 SSH-2
2	仅对 SSH-2 有效, 不包括 SSH-1
F	仅对 F-Secure SSH 有效
N	F-Secure SSH 中没有

sshd 选项

SSH1	SSH2	Open SSH	选项	含义
		✓	-4	仅使用 IPv4 格式的地址
		✓	-6	仅使用 IPv6 格式的地址
✓		✓	<i>-b bits</i>	服务器密钥的长度
✓		✓	<i>-d</i>	详细模式
	✓		<i>-d level</i>	允许显示 debug 信息
	✓		<i>-d "module=level"</i>	允许某个模块显示 debug 信息

SSH1	SSH2	Open SSH	选项	含义
✓	✓	✓	<i>-f filename</i>	使用另外的配置文件
✓	✓	✓	<i>-g time</i>	设置可以登录的时间
✓	✓	✓	<i>-h filename</i>	使用别的主机密钥文件
✓	✓	✓	<i>-i</i>	用 <i>inetd</i> 调用 <i>sshd</i>
✓		✓	<i>-k time</i>	重新生成密钥的周期
	✓		<i>-o "keyword value"</i>	给配置关键字赋值
✓	✓	✓	<i>-p port</i>	选择 TCP 端口号
✓	✓	✓	<i>-q</i>	安静模式
		✓	<i>-Q</i>	如果不支持 RSA 就进入安静模式
	✓		<i>-v</i>	详细模式
✓			<i>-V</i>	打印版本号
		✓	<i>-V id</i>	OpenSSH 的 SSH2 兼容模式

sshd 关键字

SSH1	SSH2	Open SSH	关键字	值	含义
✓	✓	✓	#	任意文字	注释行
✓			AccountExpireWarningDays	# 天数	提醒用户离账号到期还有多少天
		✓	AFSTokenPassing	Yes/no	将 AFS 令牌转发到服务器
N			AllowAgentForwarding	Yes/no	启用代理转发
	✓		AllowedAuthentications	认证方式	允许使用何种认证技术
N			AllowCshrcSourcingWith-Subsystems	Yes/no	源 shell 启动文件
F			AllowForwardingPort	端口号 列表	允许转发的端口号
F			AllowForwardingTo	主机 / 端口号列表	允许转发到达的主机

SSH1	SSH2	Open SSH	关键字	值	含义
✓	N	✓	AllowGroups	用户组列表	基于 Unix 用户组的访问控制机制
✓	✓		AllowHosts	主机列表	基于主机名的访问控制机制
✓	✓		AllowSHosts	主机列表	用 <i>.shosts</i> 实现的访问控制机制
✓	N	✓	AllowTcpForwarding	Yes/no	启用 TCP 端口转发
	N		AllowTcpForwardingFor-Users	用户列表	允许哪些用户转发
	N		AllowTcpForwardingFor-Groups	组列表	允许哪些组转发
✓	N	✓	AllowUsers	用户列表	基于用户名的访问控制
	N		AllowX11Forwarding	Yes/no	启用 X 转发
	✓		AuthorizationFile	文件名	认证文件的位置
✓	✓	✓	CheckMail	Yes/no	登录时检查新邮件
	N		ChRootGroups	组列表	登录时运行 chroot()
	N		ChRootUsers	用户列表	登录时运行 chroot()
	✓	2	Ciphers	加密算法列表	选择加密算法
F			DenyForwardingPort	端口列表	禁止转发的端口
F			DenyForwardingTo	主机 / 端口列表	禁止哪些主机做转发的目的地址
✓	N	✓	DenyGroups	组列表	基于 Unix 用户组的访问控制
✓	✓		DenyHosts	主机列表	基于主机名的访问控制
✓	✓		DenySHosts	主机列表	通过 <i>.shosts</i> 进行的访问控制

SSH1	SSH2	Open SSH	关键字	值	含义
	N		DenyTcpForwardingFor- Users	用户列表	禁止哪些用户转发
	N		DenyTcpForwardingFor- Groups	组列表	禁止哪些组转发
✓	N	✓	DenyUsers	用户列表	基于用户名的访问 控制
		2	DSAAuthentication	Yes/no	允许使用 SSH2 的 DSA 认证
✓	✓		FascistLogging	Yes/no	详细模式
✓			ForcedEmptyPasswdChange	Yes/no	若密码为空则强制 更改
✓			ForcedPasswdChange	Yes/no	首次登录时需更改 密码
	✓		ForwardAgent	Yes/no	允许使用代理转发
	✓		ForwardX11	Yes/no	允许使用 X 转发
		✓	GatewayPorts	Yes/no	允许远程主机连接 本地的转发端口
		2	HostDSAKey	文件名	DSA 密钥的位置
✓		✓	HostKey	文件名	主机密钥文件的位 置
		✓	Hostkeyfile	文件名	主机密钥文件的位 置
✓			IdleTimeout	时间	设置空闲超时时间
✓	✓	✓	IgnoreRhosts	Yes/no	忽略 .rhosts 文件
✓	✓		IgnoreRootRhosts	Yes/no	忽略 .rhosts 文件
✓	✓		IgnoreUserKnownHosts	Yes/no	忽略用户的已知主 机密钥
✓	✓	✓	KeepAlive	Yes/no	发送 keepalive 包
✓		✓	KerberosAuthentication	Yes/no	允许使用 Kerberos 认证
✓		✓	KerberosOrLocalPasswd	Yes/no	Kerberos 降级认证
✓		✓	KerberosTgtPassing	Yes/no	支持可授权许可证

SSH1	SSH2	Open SSH	关键字	值	含义
		✓	KerberosTicketCleanup	Yes/no	登出时清除许可证缓存
✓		✓	Key_regeneration_interval	时间	重新生成密钥的间隔
✓	✓	✓	ListenAddress	IP 地址	在指定接口上监听
✓	✓	✓	LoginGraceTime	时间	认证的时间限制
		✓	LogLevel	Syslog 等级	设置系统日志等级
N			Macs	算法	选择 MAC 算法
N			MaxBroadcastsPerSecond	# 广播数	监听 UDP 广播
✓			MaxConnections	# 连接数	允许同时进行的最大连接数
✓			NoDelay	Yes/no	允许使用 Nagle 算法
✓	✓	✓	PasswordAuthentication	Yes/no	允许使用口令认证
	✓		PasswordGuesses	猜 # 次 口令	允许输入口令 # 次
✓			PasswordExpireWarningDays	# 天数	口令过期前提醒用户
✓	✓	✓	PermitEmptyPasswords	Yes/no	允许密码为空
✓	✓	✓	PermitRootLogin	Yes/no/ nopwd	允许超级用户登录
N			PGPPublicKeyFile	文件名	PGP 公钥文件的缺省位置
✓		✓	PidFile	文件名	Pid 文件的位置
✓	✓	✓	Port	端口号	选择服务器的端口号
✓	✓	✓	PrintMotd	Yes/no	打印日期
		✓	Protocol	1/2/1,2	允许使用 SSH-1 和 / 或 SSH-2 协议
✓			PubkeyAuthentication	Yes/no	允许使用公钥认证
✓			PublicHostKeyFile	文件名	公用主机密钥的位置

SSH1	SSH2	Open SSH	关键字	值	含义
✓	✓		QuietMode	Yes/no	安静模式
✓			RandomSeed	文件名	随机数种子文件的位置
	✓		RandomSeedFile	文件名	随机数种子文件的位置
N			RekeyIntervalSeconds	秒数	重新生成密钥的频率
	✓		RequireReverseMapping	Yes/no	进行反向DNS查找
	✓		RequiredAuthentications	认证方式	要求具备的认证技术
✓	✓	✓	RhostsAuthentication	Yes/no	允许使用.rhosts 认证
	✓		RhostsPubKey- Authentication	Yes/no	允许联合认证
✓	✓	✓	RhostsRSAAuthentication	Yes/no	允许联合认证
✓	✓	✓	RSAAuthentication	Yes/no	允许使用公钥认证
✓			ServerKeyBits	# bits	服务器密钥的位数
		✓	SkeyAuthentication	Yes/no	允许使用S/Key认证
		✓	Ssh1Compatibility	Yes/no	启用 SSH1 兼容模式
	✓		Sshd1Path	文件名	ssh1 的路径
✓			SilentDeny	Yes/no	DenyHosts 不打印任何消息
✓	✓	✓	StrictModes	Yes/no	严格控制文件 / 目录的访问权限
✓	✓	✓	SyslogFacility	Syslog 等级	设置系统日志记录的信息等级
✓			TISAuthentication	Yes/no	允许进行TIS认证
✓			Umask	Unix umask	设置登录 umask
✓		✓	UseLogin	Yes/no	选择登录程序

SSH1	SSH2	Open SSH	关键字	值	含义
	✓		UserConfigDirectory	目录名	用户的 SSH2 目录的位置
	✓		UserKnownHosts	Yes/no	参照 <code>~/.ssh2/knownhosts</code> 中的内容
	✓		VerboseMode	Yes/no	详细模式
✓	N	✓	X11Forwarding	Yes/no	启用 X 转发
✓		✓	X11DisplayOffset	# offset	限制 SSH 使用的显示区号
✓		✓	XAuthLocation	Filename	<code>xauth</code> 的位置

ssh 和 scp 关键字

SSH1	SSH2	Open SSH	关键字	值	含义
✓	✓	✓	#	任意文字	注释行
		✓	AFSTokenPassing	Yes/no	将 AFS 令牌传给服务器
N			AllowAgentForwarding	Yes/no	启用代理转发
	✓		AllowedAuthentications	认证方式	允许使用的认证技术
N			AuthenticationNotify	Yes/no	在 <code>stdout</code> 上显示认证成功的信息
N			AuthenticationSuccessMsg	Yes/no	在 <code>stderr</code> 上显示认证成功的消息
✓	✓	✓	BatchMode	Yes/no	无提示
		✓	CheckHostIP	Yes/no	检测是否存在 DNS 欺骗
✓		1	Cipher	加密算法	请求加密算法
	✓	2	Ciphers	加密算法	列表 支持的加密算法

SSH1	SSH2	Open SSH	关键字	值	含义
✓			ClearAllForwardings	Yes/no	忽略任何特定的转发
✓	✓	✓	Compression	Yes/no	启用数据压缩
✓		✓	CompressionLevel	0-9	选择压缩算法
✓		✓	ConnectionAttempts	# attempts	客户端重试的次数
	N		DefaultDomain	域	指定域名
	✓		DontReadStdin	Yes/no	stdin 重定向为 /dev/null
	2		DSSAAuthentication	Yes/no	允许使用 SSH-2 的 DSA 认证
✓	✓	✓	EscapeChar	字符	设置转义字符 (^=Ctrl)
✓	✓	✓	FallBackToRsh	Yes/no	若 ssh 失败，转用 rsh
	✓		ForcePTYAllocation	Yes/no	分配一个伪终端
✓	✓	✓	ForwardAgent	Yes/no	启用代理转发
✓			ForwardX11	Yes/no	启用 X 转发
✓	✓	✓	GatewayPorts	Yes/no	允许远程主机连接本地的转发端口
✓		1	GlobalKnownHostsFile	文件名	全局已知主机文件的位置
		2	GlobalKnownHostsFile2	文件名	全局已知主机文件的位置
	✓		GoBackground	Yes/no	转到后台运行
✓	✓	✓	Host	模式名	该主机的设置内容中最开始的一段 (对 SSH2 来说，则表示主机的真实地址)
✓		✓	HostName	主机名	主机的真实名字
✓		1	IdentityFile	文件名	私钥文件的名字 (RSA)
		2	IdentityFile2	文件名	私钥文件的名字 (DSA)

SSH1	SSH2	Open SSH	关键字	值	含义
✓	✓	✓	KeepAlive	Yes/no	发送 keepalive 包
✓		✓	KerberosAuthentication	Yes/no	允许进行 Kerberos 认证
		✓	KerberosTgtPassing	Yes/no	支持 TGT
✓	✓	✓	LocalForward	端口, socket	本地端口转发
	N		Macs	算法名	选择 MAC 算法
	✓		NoDelay	Yes/no	允许使用 Nagle 算法
✓		✓	NumberOfPasswordPrompts	# 重试次数	允许重试的次数
✓	✓	✓	PasswordAuthentication	Yes/no	允许使用密码认证
	✓		PasswordPrompt	字符串	密码确认
✓			PasswordPromptHost	Yes/no	需确认密码的主机名
✓			PasswordPromptLogin	Yes/no	需确认密码的用户名
✓	✓	✓	Port	端口号	选择服务器的端口号
✓		✓	ProxyCommand	命令	连接代理服务器
	✓		QuietMode	Yes/no	安静模式
	✓		RandomSeedFile	文件名	随机数种子文件的位置
✓	✓	✓	RemoteForward	端口号, socket	远程端口转发
✓	✓	✓	RhostsAuthentication	Yes/no	允许使用 .rhosts 认证
	✓		RhostsPubKey-Authentication	Yes/no	允许使用组合认证
✓	✓	✓	RhostsRSAAuthentication	Yes/no	允许使用组合认证
✓	✓	✓	RSAAuthentication	Yes/no	允许使用公钥认证
	N		PGPSecretKeyfile	文件名	用于认证的 PGP 私钥文件的默认位置

SSH1	SSH2	Open SSH	关键字	值	含义
		✓	SkeyAuthentication	Yes/no	允许使用 S/Key 认证
N			SocksServer	服务器	指定 SOCKS 服务器
✓			Ssh1AgentCompatibility	Yes/no	启用 SSH1 代理兼容模式
✓			Ssh1Compatibility	Yes/no	启用 SSH1 兼容模式
✓			Ssh1Path	文件名	<i>ssh1</i> 的路径
✓			SshSignerPath	文件名	<i>ssh-signer2</i> 的路径
✓	✓	✓	StrictHostKeyChecking	Yes/no/ask	主机密钥不匹配时的对策
✓			TISAuthentication	Yes/no	允许使用 TIS 认证
✓		✓	UsePrivilegedPort	Yes/no	允许使用特权端口
✓	✓	✓	User	用户名	远程用户
✓		1	UserKnownHostsFile	文件名	用户的已知主机文件的位置
		2	UserKnownHostsFile2	文件名	用户的已知主机文件的位置
✓	✓	✓	UseRsh	Yes/no	用 <i>rsh</i> 代替 <i>ssh</i>
	✓		VerboseMode	Yes/no	详细模式
		✓	XAuthLocation	文件名	<i>xauth</i> 的位置

ssh 选项

SSH1	SSH2	Open SSH	选项	含义
		✓	-2	仅使用 SSH-2
		✓	-4	仅使用 IPv4 格式的网络地址
		✓	-6	仅使用 IPv6 格式的网络地址
✓			-8	无意义；仅在退到 <i>rsh</i> 时将此值传给它

SSH1	SSH2	Open SSH	选项	含义
✓	✓	✓	-a	禁用代理转发
	✓	✓	+a	启用代理转发
		✓	-A	启用代理转发
✓	✓	✓	-c 加密算法名	选择加密算法
✓		✓	-C	启用压缩
	✓		-C	禁用压缩
	✓		+C	启用压缩
	✓		-d debug 等级	启用 debug 模式
	✓		-d “模块名 =debug 等级”	对特定模块启用 debug 模式
✓	✓	✓	-e 字符	设置转义字符 (^ = Ctrl)
✓	✓	✓	-f	转到后台
	✓		-F 文件名	使用其他配置文件
✓	✓	✓	-g	允许远程主机连接本地的转发端口
	✓		-h	打印帮助信息
✓	✓	✓	-i 文件名	选择身份标识文件
✓		✓	-k	禁止转发 Kerberos 许可证
✓	✓	✓	-l 用户名	远程用户名
✓	✓	✓	-L 端口 1:主机 2:端口 2	本地端口转发
	N		-m algorithm	选择 MAC 算法
✓	✓	✓	-n	将 stdin 重定向为 /dev/null
	2		-N	不执行远程命令
✓	✓	✓	-o “关键字名 值”	设置配置关键字
✓	✓	✓	-p 端口号	选择 TCP 端口号
✓	✓	✓	-P	使用非特权端口
✓	✓	✓	-q	安静模式
✓	✓	✓	-R 端口 1:主机 2:端口 2	远程端口转发
	✓		-s 子系统	调用远程子系统
	✓		-S	没有会话通道
✓	✓	✓	-t	分配 tty
	2		-T	不分配 tty

SSH1	SSH2	Open SSH	选项	含义
✓	✓	✓	-v	详细模式
✓	✓	✓	-V	打印版本号
✓	✓	✓	-x	禁用 X 转发
	✓		+x	启用 X 转发
		✓	-X	启用 X 转发

scp 选项

SSH1	SSH2	Open SSH	选项	含义
	✓		-I	启用 <i>scp1</i> 兼容模式
		✓	-4	尽使用 IPv4 格式的网络地址
		✓	-6	仅使用 Ipv6 格式的网络地址
✓		✓	-a	不逐个统计文件
✓		✓	-A	逐个打印文件的统计信息
✓	✓	✓	-B	无需确认
✓	✓	✓	-c 加密算法	选择加密算法
✓		✓	-C	启用压缩
✓	✓	✓	-d	拷贝单个文件时要求目标参数是一个目录
	✓		-D “模块名 =debug 等级”	允许某个特定模块打印 debug 信息
✓	✓	✓	-f	指定拷贝源（内部用法）
	✓		-h	打印帮助信息
✓	N	✓	-i 文件名	选择身份标识文件
✓			-L	使用非特权端口
	✓		-n	打印操作步骤，但不拷贝
✓	N	✓	-o “关键字值”	设置配置关键字的值
✓	✓	✓	-p	保持文件属性
✓	✓	✓	-P 端口号	设置 TCP 端口号
✓		✓	-q	不打印统计信息

SSH1	SSH2	Open SSH	选项	含义
	✓		-q	安静模式
✓			-Q	打印统计信息
	✓		-Q	不打印统计信息
✓	✓	✓	-r	递归拷贝
✓	✓	✓	-S 文件名	ssh 可执行文件的路径
✓	✓	✓	-t	指定拷贝的目标（内部使用）
	✓		-u	删除原文件
✓	✓	✓	-v	详细模式
	✓		-V	打印版本信息

ssh-keygen 选项

SSH1	SSH2	Open SSH	选项	含义
	✓		-l 文件名	SSH1 密钥文件转换为 SSH2 格式
✓	✓	✓	-b 位数	生成的密钥的位数
N			-B 正整数 ^a	指定密钥显示时的基数
✓		✓	-c	更改注释（与 -C 合用）
	✓		-c 注释内容	更改注释 :
✓		✓	-C 注释内容	指定新的注释（与 -c 合用）
		✓	-d	生成 DSA 密钥
N			-D 文件名	根据私钥文件生成公钥
	✓		-e 文件名	交互式环境下编辑密钥文件
✓	b	✓	-f 文件名	输出文件的名字
	✓		-F 文件名	打印公钥的指纹
	✓		-h	打印帮助然后退出
		✓	-l	打印公钥的指纹
✓		✓	-N 密码短语	指定新的密码短语
	✓		-o 文件名	输出文件的名字
✓		✓	-p	更改密码短语（与 -P 及 -N 合用）

SSH1	SSH2	Open SSH	选项	含义
✓	F N	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	-p 密码短语	更改密码短语
			-P 密码短语	指定原来的密码短语(与-p合用)
			-P	使用空的密码短语
			-q	安静模式下的压缩过程指示
			-r	搅动随机池
			-R	监测 RSA (exit code 0/1)
			-t 算法	选择密钥生成算法
			-u	更换加密算法
			-v	打印版本信息, 然后退出
			-V	打印版本信息, 然后退出
✓	F N	✓ ✓ ✓ ✓ ✓	-x	将 OpenSSH 公钥转换到 SSH2
			-X	将 SSH2 公钥转换到 OpenSSH
			-y	根据私钥文件生成公钥
			-? ^a	打印版本信息, 然后退出

- a. 文档中未提及。
 b. 输出文件名是 *ssh-keygen2* 的最后一个参数。
 c. 在您的 shell 中可能需将问号转义, 如 `^?`。

ssh-agent 选项

SSH1	SSH2	Open SSH	选项	含义
✓	✓ ✓	✓ ✓ ✓ ✓	-I	SSH1 兼容模式
			-c	打印 C shell 风格的命令
			-k	杀死已有的代理
			-s	打印 sh 风格的命令

ssh-add 选项

SSH1	SSH2	Open SSH	选项	含义
	✓		-l	有限兼容 SSH1
✓	✓	✓	-d	卸载密钥
✓	✓	✓	-D	卸载全部密钥
	✓		-f 步数	有限步的代理转发
	✓		-F 主机列表	有限步的代理转发
	✓		-I	由 ID 标识 PGP 密钥
✓	✓		-l	列出已加载的密钥
		✓	-l	列出已加载密钥的指纹
	✓		-L	锁定代理
		✓	-L	列出已加载的密钥
	✓		-N	由名字标识 PGP 密钥
✓	✓		-P	从 stdin 读取密码短语
	✓		-P	由指纹标识 PGP 密钥
	✓		-R 文件名	指定 PGP 密钥环
✓			-t 超时值	超时值一到密钥就过期
✓			-U	代理解锁

身份标识及授权文件

~/.ssh/authorized_keys (SSH1, OpenSSH/1) 及 ~/.ssh/authorized_keys2 (OpenSSH/2): 每行中包含一个公钥，选项在密钥之前。

选项	含义
command="Unix shell 命令"	指定一条强制命令
environment=" 变量 = 值 "	设置环境变量
from= 主机名或 IP 地址	限制接入的主机数
idle-timeout= 时间值	设置超时时间

选项	含义
no-agent-forwarding	禁用代理转发
no-port-forwarding	禁用端口转发
no-pty	不分配 TTY

~/.ssh2/authorization (SSH2): 每行中包括一个关键字 / 值的组合。

关键字	含义
Command Unix 命令	指定强制命令
Key filename.pub	公钥文件的位置
PgpPublicKeyFile 文件名	PGP 公钥文件的位置
PgpKeyFingerprint 指纹	根据指纹选择 PGP 密钥
PgpKeyId id	根据 ID 选择 PGP 密钥
PgpKeyName 名字	根据名字选择 PGP 密钥

~/.ssh2/identification (SSH2): 每行中包括一个关键字 / 值的组合。

关键字	含义
IdKey 文件名	私钥文件的位置
IdPgpKeyFingerprint 指纹	根据指纹选择 PGP 密钥
IdPgpKeyId id	根据 ID 选择 PGP 密钥
IdPgpKeyName 名字	根据名字选择 PGP 密钥
PgpSecretKeyFile 文件名	PGP 私钥文件的位置

环境变量

变量	赋值者	来源	含义
SSH_AUTH_SOCK	ssh-agent	SSH1, OpenSSH	指向代理套接字的名称
SSH2_AUTH_SOCK	ssh-agent	SSH2	指向代理套接字的名称
SSH_CLIENT	sshd	SSH1, OpenSSH	客户端套接字的信息
SSH2_CLIENT	sshd	SSH2	客户端套接字的信息

变量	赋值者	来源	含义
SSH_ORIGINAL_COMMAND	<i>sshd</i>	SSH1	客户端的远程命令字符串
SSH SOCKS_SERVER	<i>sshd</i>	SSH2	SOCKS 防火墙信息
SSH_TTY	<i>sshd</i>	SSH1, OpenSSH	已分配的 TTY 名
SSH2_TTY	<i>sshd</i>	SSH2	已分配的 TTY 名