# 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
- 跨网关连接

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，也就不应该指望无人参与的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，它在某种程度上不那么容易受到攻击。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；如果有一百个批处理，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，你对可能遇上的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号：

```
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。

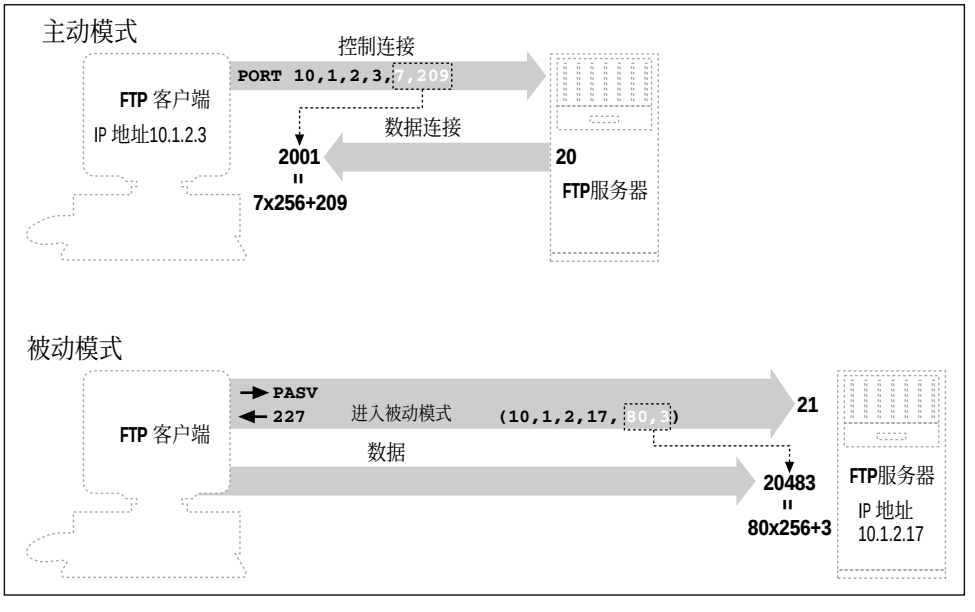


图11-1：基本FTP操作：控制连接及主动/被动模式传输对比

## 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*。

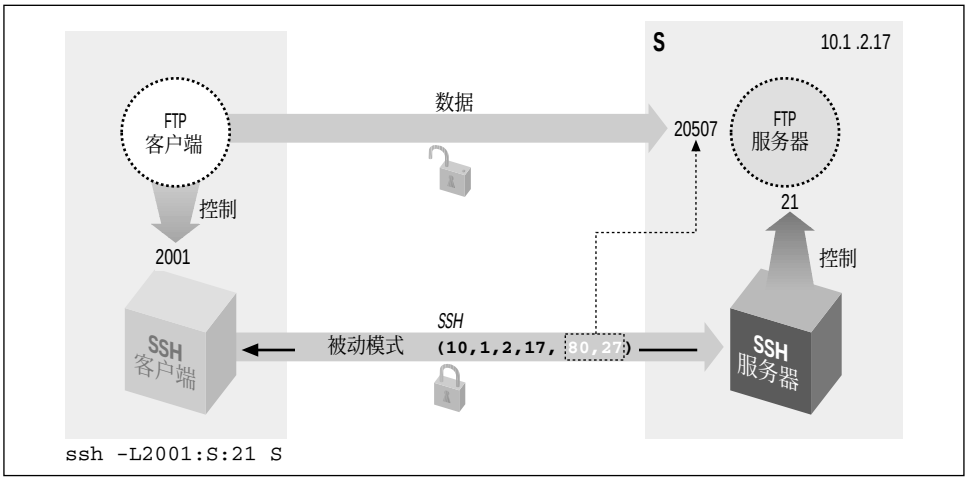


图 11-2：转发控制连接

```
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）：

```
so_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*truss*。Solaris中也有一条命令叫做*strace*，不过它与我们的问题完全无关。Solaris 1（SunOS 4及更早）中有*trace*命令，在BSD中则为*ktrace*。

```
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的确在监听从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`GatewayPosts`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，连接还是可能会失败：

```
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“ 现在这样是为什么呢？ ”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-ftpd*。（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）。

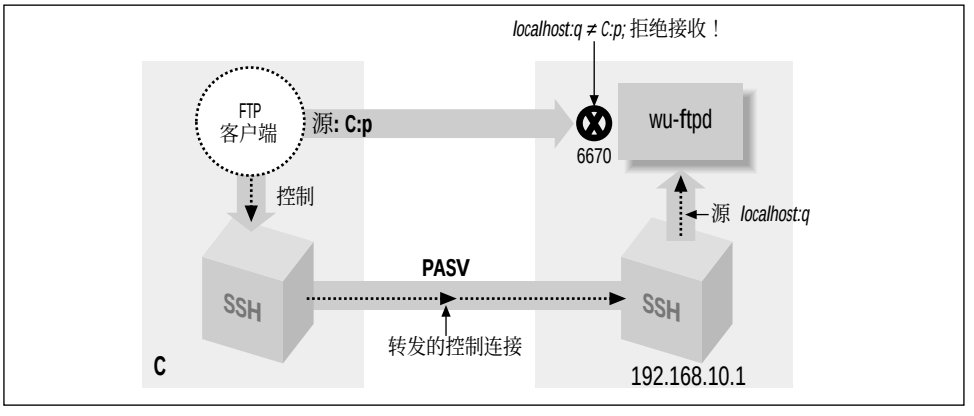


图 11-3：“PASV 端口窃听”

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。

## 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。

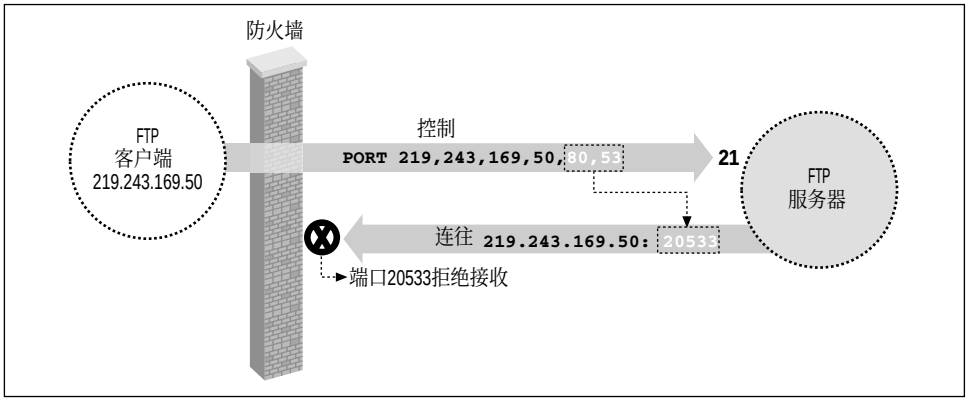


图11-4：防火墙后的FTP客户端

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，并等待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子，让特定的FTP数据连接回来时通过。这个口子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。

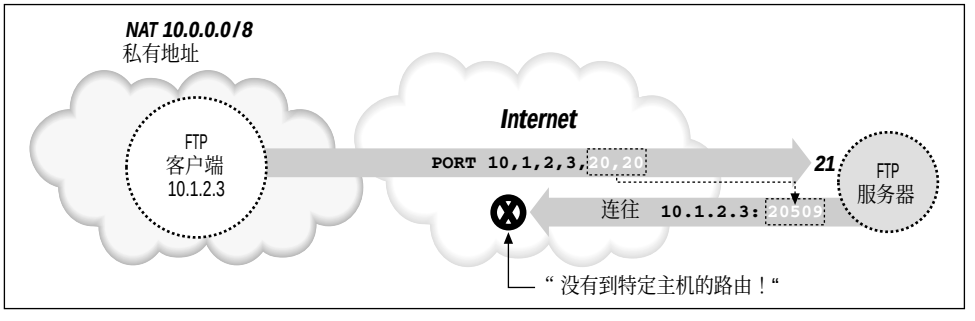


图11-5：客户端NAT阻碍了主动模式的FTP传输

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。

### 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编址机制，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）。

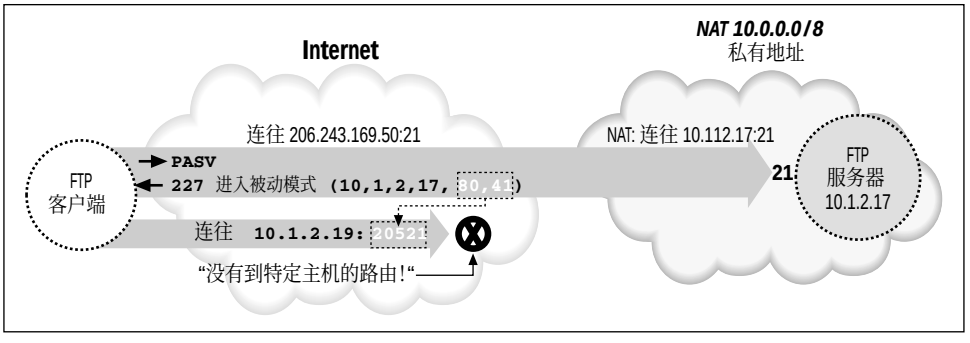


图11-6：服务器端NAT阻碍了被动模式的FTP传输

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 重组的防火墙后时仍
      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。解决这个问题的惟一办法是：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提建议。

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 命令原本不是要这样用的；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 cmds 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，这是惟一的方式。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_STATE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: %.100s", 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 &
   ```

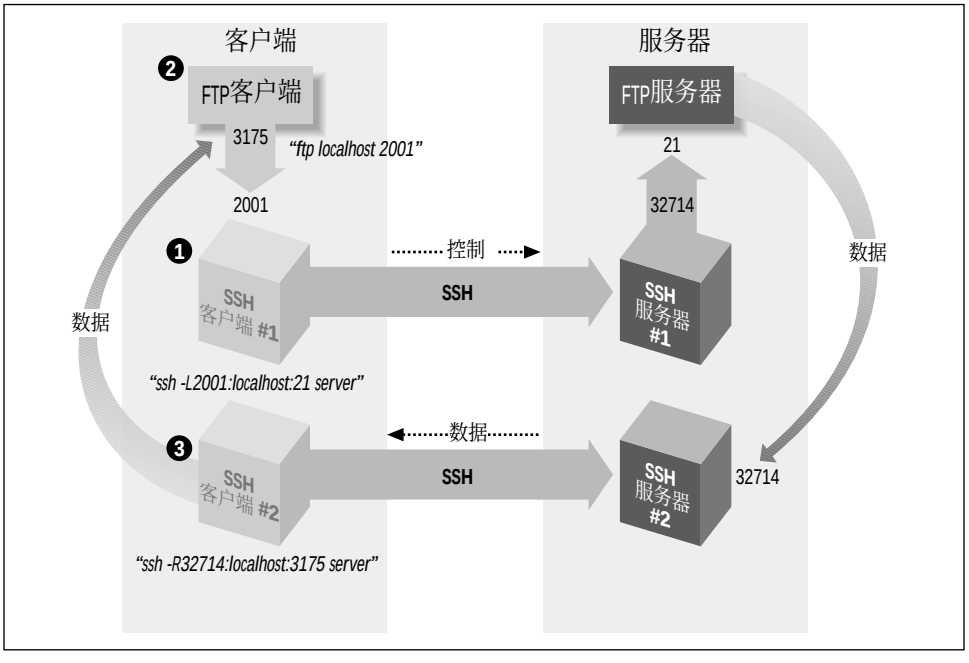


图 11-7：转发 FTP 数据连接

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 cmds off.
ftp> passive
Passive mode on.
ftp> passive
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，那也只可能正常一次，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SMTR（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

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）。

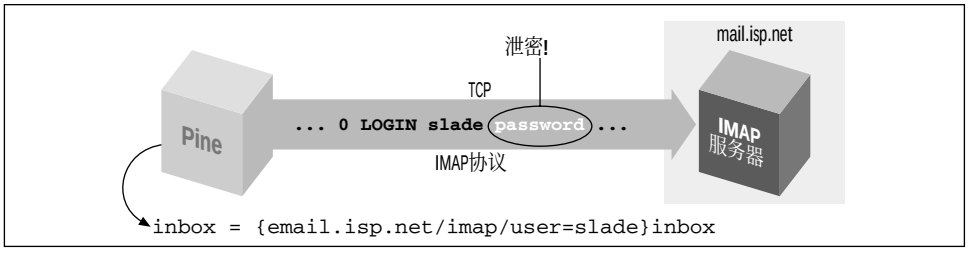


图 11-8：普通 IMAP 连接

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]

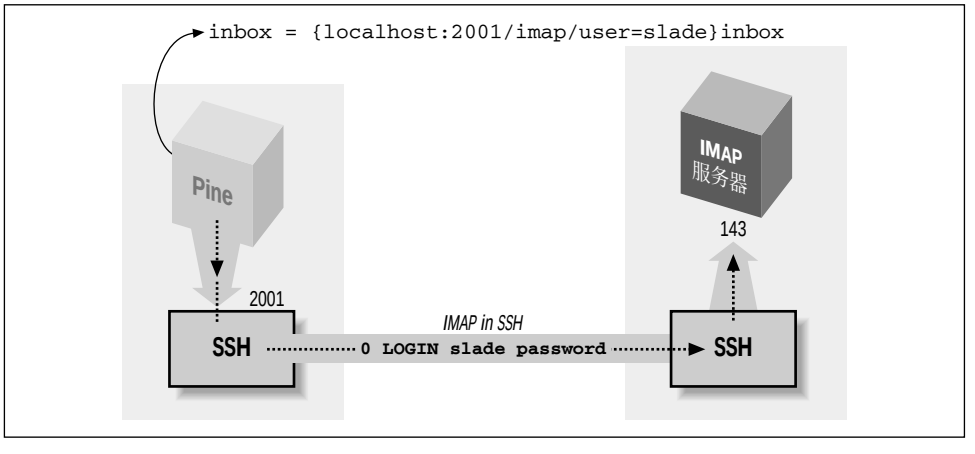


图 11-9：转发 IMAP 连接

不过，这种技术有两个缺点：

*安全隐患*

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。

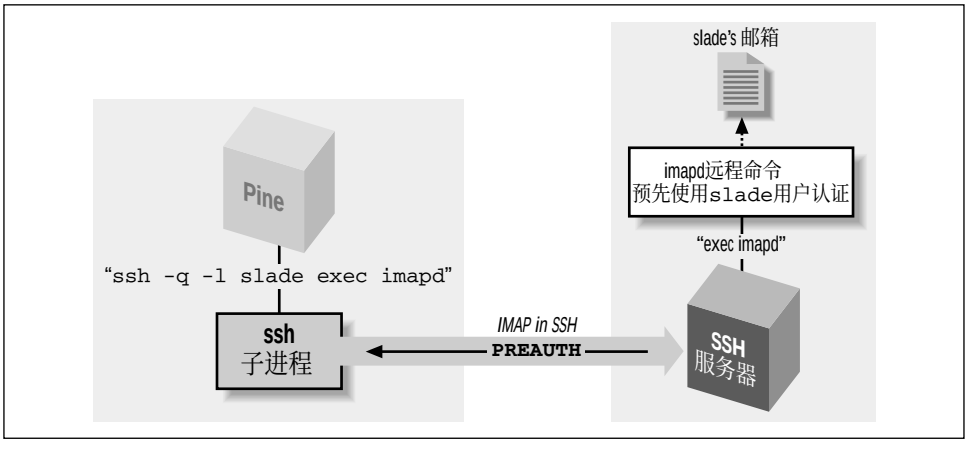


图11-10：基于SSH的预认证Pine/IMAP

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`rch-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 %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
$nntp = 119; # 对 NNTP 的知名端口
$smtp_proxy = 2025; # 转发 SMTP 连接的本地端口
$nntp_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:$smtp_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',"$smtp_proxy:mail:$smtp",'-L',"$nntp_proxy:news:$nntp");
    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。

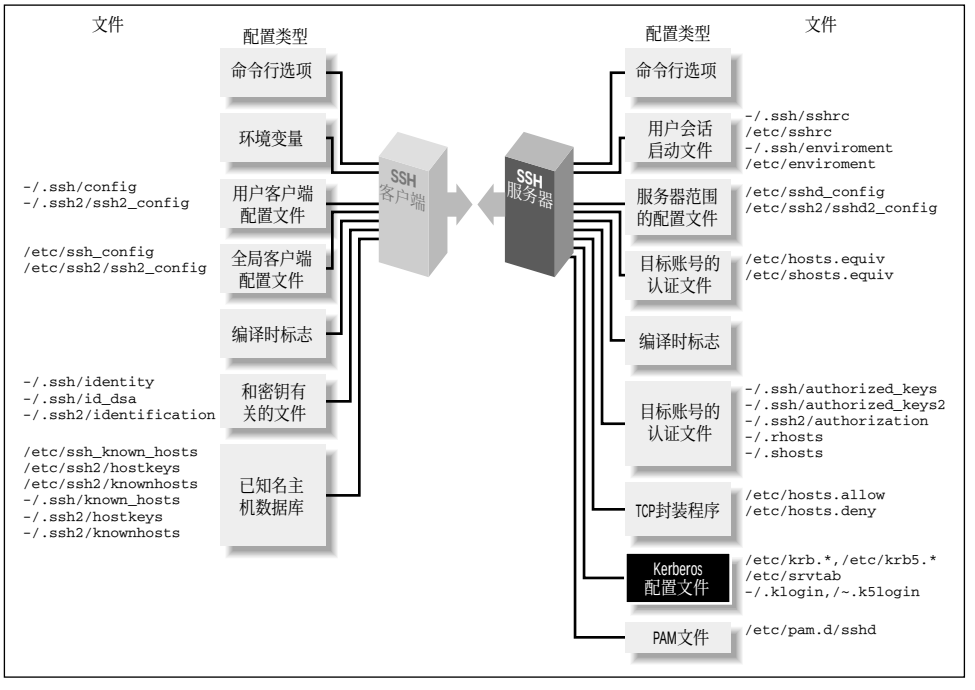


图 11-11：Kerberos 设置（高亮显示部分）

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，OpenSSH/1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并不容易。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，那可就有麻烦了。

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知道这项服务的密钥。

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*libk-5crypto.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
```

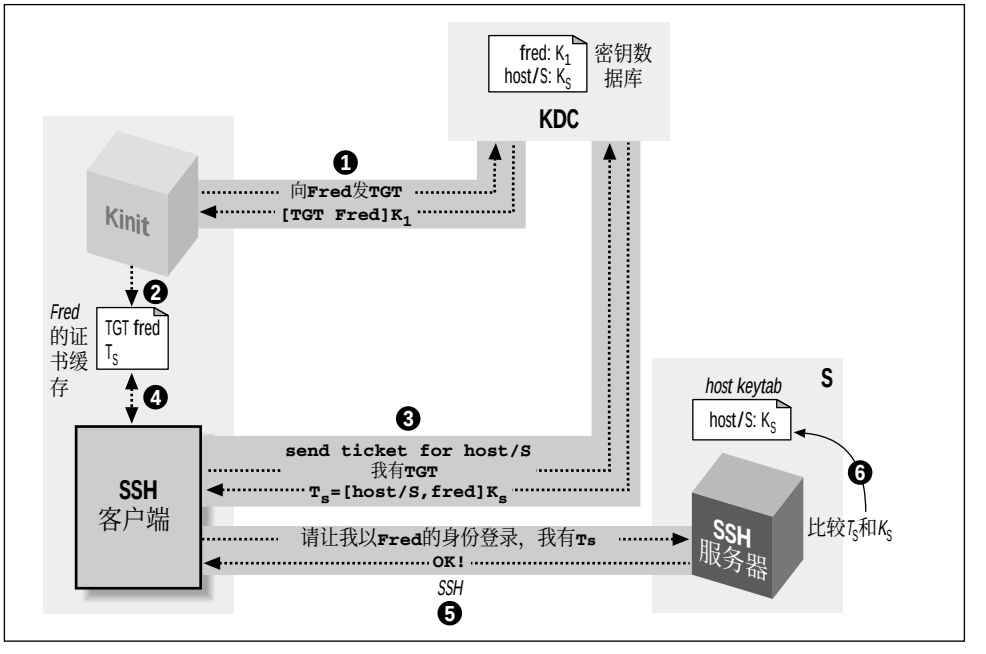


图 11-12：使用 Kerberos 认证的 SSH

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。

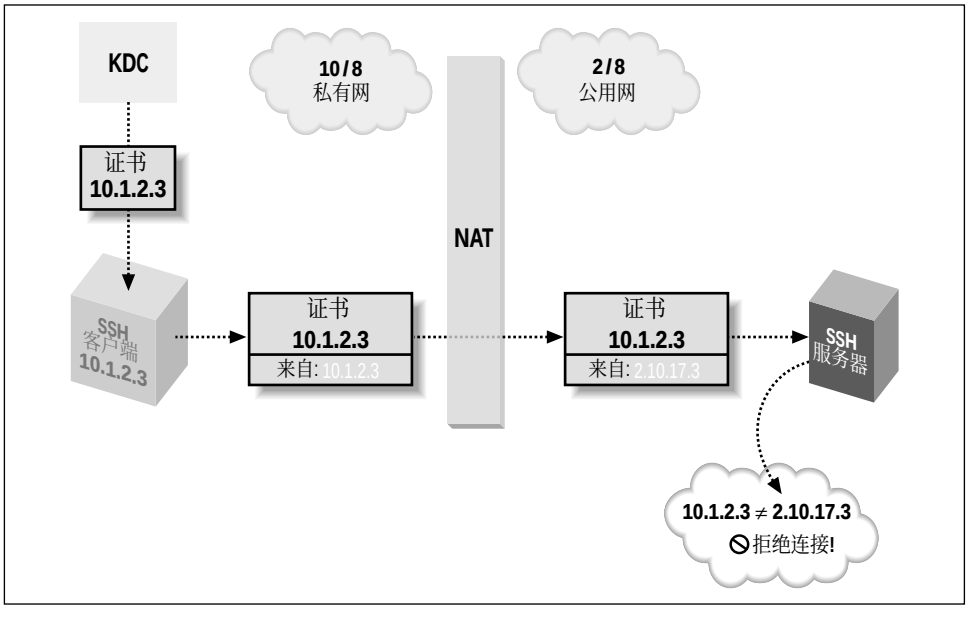


图 11-13：Kerberos 与 NAT

### 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闲逛。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。

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:
  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 ENGINERING 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。

ENGINERING 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_tgt_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 `--ocalstatedir=/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
    kadmind_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 | *rcmd.hostname@REALM* | *host/hostname@REALM* |
| 配置文件 | */etc/krb.conf* | */etc/krb5.conf* |

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 | */etc/srvtab* | */etc/krb5.keytab* |
| 认证文件 | *~/.klogin* | *~/.k5login* |

#### 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（crendentials 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。

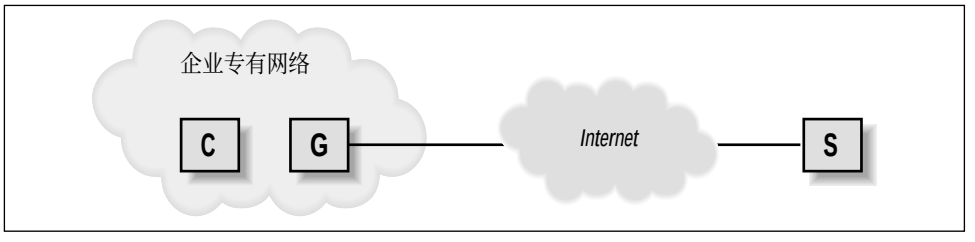


图11-14：代理网关

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）。

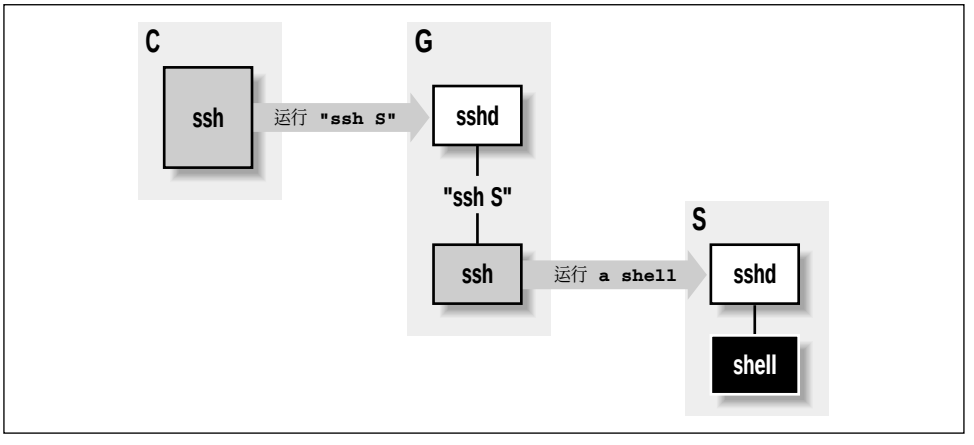


图11-15：通过代理网关的SSH连接链

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
```

然后在网关G上使用你的密钥启动到*S*的SSH连接，并关联一个强制命令：[8.2.4]

```
 # 网关G上的~/.ssh/authorized_keys
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。不这样做的原因通常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 ... S: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）。

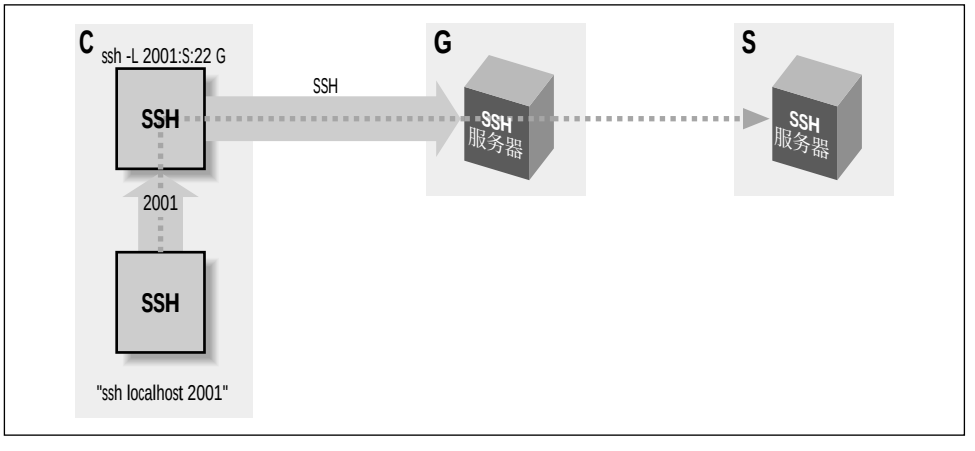


图 11-16：通过代理网关转发 SSH 连接

像这样：

```
# 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 -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 *—x* 和 *—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。