本章内容：

- 该技术的局限
- 基于公钥的配置
- 可信主机访问控制
- 用户 rc 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。

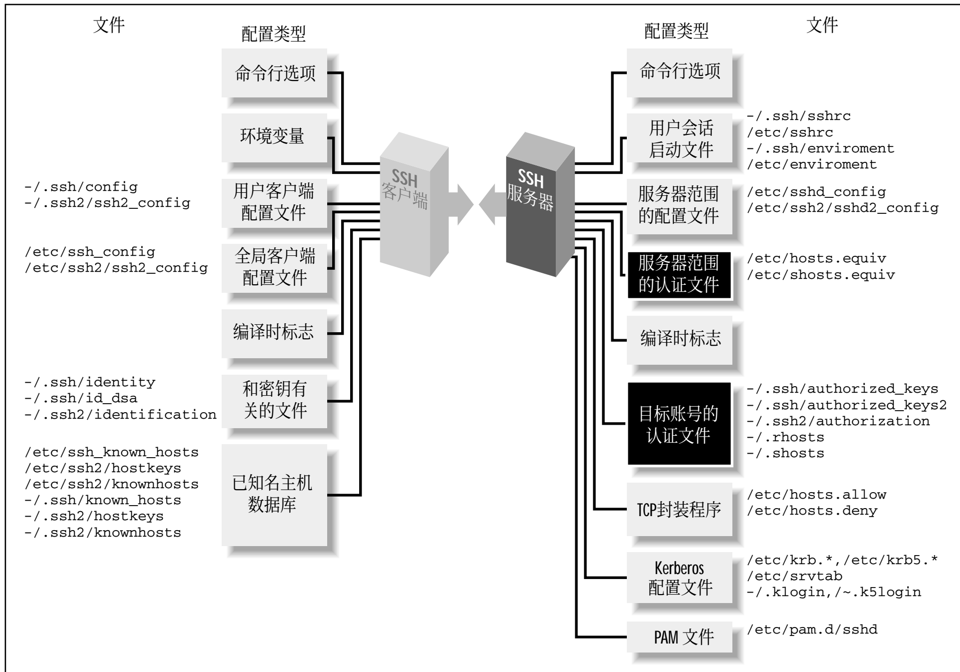

图 8-1：每账号配置（高亮显示部分）

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（*ssh1 —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，它到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 AAAAB3NzaC1kc3MA... My OpenSSH key
```

8.2.4 强制命令

通常，SSH 连接会调用客户端选定的远程命令：

```
# 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*。

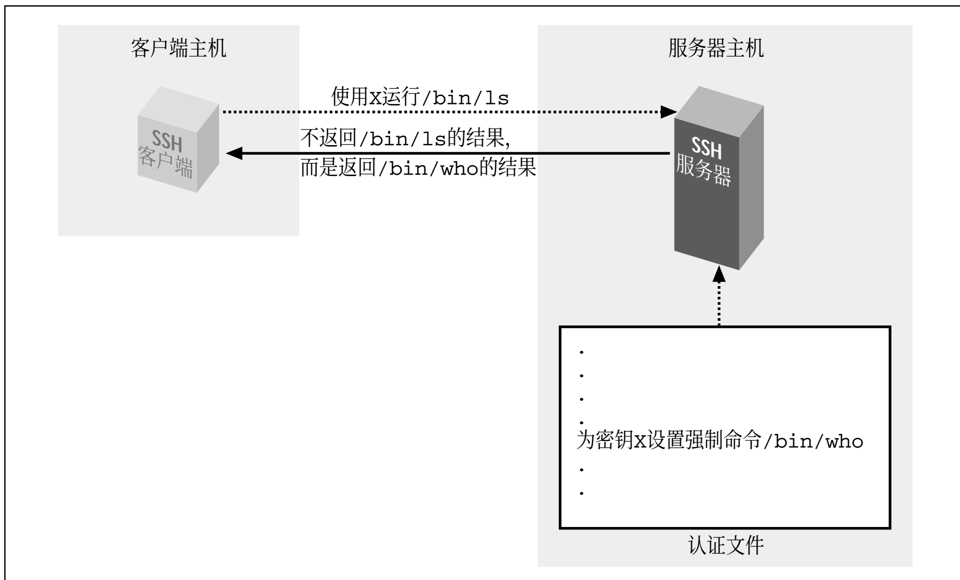

图 8-2：强制命令使用 /bin/who 取代 /bin/ls

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？

在这种情况中，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

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，这些问题是由于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 envrionment 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 惟一确定的。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 IdelTimeout 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 `AllowHosts` 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 RhostsRSAuthentication（SSH1，OpenSSH）或 hostbased（SSH2），它可以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。