CHAPTER 18

Enabling Remote Access

In the "old days," people used telnet to access their systems remotely. Nowadays they can't do this: telnet sends its passwords in plain text over the network, and because too often these packets are transmitted across an insecure network, this really is not an option. It is simple for someone with a packet analyzer such as Ethereal to grab packets and read your username and password from the network. Therefore, new methods of remotely accessing a server have been created. I'll discuss two of these techniques in this chapter. First you will learn how to use Secure Shell (SSH) to set up a secure (read: encrypted) connection with a server. Next, you will learn how you can use VNC to get access to the graphical display of your server remotely. VNC isn't secure by itself, but in this chapter you'll learn how to combine it with SSH to make it secure.

In this chapter, I'll cover the following subjects:

Understanding how SSH works: I'll explain how SSH uses encryption keys to establish secure remote sessions. Also, you will learn how to use SSH.

Configuring SSH: I'll explain how you can use the `sshd_config` and `ssh_config` files to tune how SSH works.

Configuring SSH key-based authentication: You will learn how to secure SSH even more by using public/private key technology for authentication.

Tunneling traffic with SSH: I'll explain how to establish a simple VPN connection between hosts using SSH.

Using VNC: You'll learn how to use VNC to get remote access to the server's graphical display.

Understanding How Secure Shell Works

The essence of SSH is its security. Public and private keys play an important role in this security. On first contact, the client and the server exchange public keys, the so-called *host key*. This host key proves the identity of the server to which a client is connecting. When connecting, the server sends its public key to the client. If this is the first time the client is connecting to this host, it replies with the message shown in Listing 18-1.

Listing 18-1. *Establishing an SSH Session with an Unknown Host*

```
The authenticity of host 'localhost'  (127.0.0.1)'  can't be established.
RSA key fingerprint is 79:20:76:ed:93:7e:aa:d7:01:25:e5:d7:de:0b:76:87.
Are you sure you want to continue connecting (yes/no)? yes
```

Only if the client trusts that this is really the intended host should the client answer yes to this request. As a result, the host is added to the file `.ssh/known_hosts` in the home directory of the user who initiated the SSH session. The next time the client connects to the same host, the client checks

this `known_hosts` file to see whether the host is already known. This check is based on the public key fingerprint of the host, which is a unique number that is related to the public key of the host. Only if this number matches the name and public key of the server that the client is connecting to is the connection established. If both pieces of data don't match, it is likely that the host the client is connecting to is not the intended host; therefore, the connection will be refused.

Once you have established the identity of the server you want to connect to, you establish a secured channel between the client and server. To establish this secured channel, you use a session key. This is an encryption key that is the same on both the server and the client; it encrypts all the data sent between the two machines. The session key is negotiated between the client and the server based on their public keys. This negotiation, amongst others, determines the protocol that should be used. Session keys can use 3DES, Blowfish, or IDEA, for example.

After establishing this secured channel, the user on the client is asked for its credentials. If nothing is configured, this will be a prompt where the user is asked to enter a username and password. This, however, is not the only way it can be done, as you'll see in the "Using Key-Based Authentication" section. Alternatively, the user can authenticate with a public/private key pair, thus proving that the user really is the user who he says he is.

All this may sound pretty complicated. The nice part is that the user won't notice anything. The user just has to enter a username and password—that's all. If you want to go beyond simple password-based authentication, however, it is useful to understand what is happening.

Working with Public/Private Key Pairs

The essence of SSH is the public/private key pair. By default, the client tries to authenticate using RSA/DSA key pairs. To make this work, first the client gets the public key of the server to establish a secure session; this happens automatically. Next, the server must get the public key of the client, which is something that has to be configured by hand. (Later in the "Using Key-Based Authentication" section you'll find more information about this procedure.) When the client has a public/private key pair, it will generate an encrypted string with its private key. If the server is able to decrypt this string using the public key of the client, the identity of the client is proved.

When using public/private key pairs, you can configure different aspects of the encryption. First, the user needs to determine what cryptographic algorithm he wants to use. For this purpose, he can choose between RSA and DSA; the latter is considered stronger. Next, the user has to determine whether he wants to protect his private key with a passphrase. This is because the private key really is used as the identity of the user. Should anyone steal this private key, it would be possible to forge the identity of the owner; therefore, it is a good idea to secure private keys with a passphrase.

Working with Secure Shell

Basically, SSH is a suite of tools that consists of three main programs and a daemon. The name of the daemon is sshd, and it runs by default on your SUSE server. The commands are `ssh`, `scp`, and `sftp`. The first, `ssh`, establishes a secured remote session. Let's say that it is like telnet but then secured with cryptography. The second, `scp`, is a useful command you can use to copy files to and from another server. The third, `sftp`, is an FTP client interface. By using it, you can establish a secured FTP session to a server that is running the sshd. One of the best features of these tools is that you can use them without any preparation or setup, and you can set them up to work entirely according to your needs. They are easy to use and are specialized tools at the same time.

Using the ssh Command

The simplest way to work with SSH is by just entering the command `ssh`, followed by the name of the host to which you want to connect. For example, to connect to the host AMS.sandervanvugt.com, you would use the following:

```
ssh AMS.sandervanvugt.com
```

Depending on whether you have connected to that host before, SSH can ask you to check the credentials of the host or just ask for your password. The `ssh` command doesn't ask for a username, because it assumes you want to connect to the other host with the same username you are logged in with locally. You have two ways to indicate that you would rather log in with another user account. First, you can specify the username followed by the @ sign and the name of the host to which you want to connect. Alternatively, you can also use the `-l` option followed by the name of the user account you want to use to connect to the other host. So basically, `ssh linda@AMS.sandervanvugt.com` and `ssh -l linda AMS.sandervanvugt.com` are the same. Ready to do your work on the remote host? Enter the `exit` command (or press Ctrl+D) to close the session and return to your own machine.

Now, it seems like it's a lot of trouble to log in completely on a remote host if you need to enter just one or two commands. If this is a situation you face often, it is good to know you can just specify the name of the command at the end of the `ssh` command. So, `ssh -l linda@AMS.sandervanvugt.com halt` would shut down the server (if user `linda` is allowed to do that). Using commands as an option to SSH is especially useful in shell scripts. If you are using SSH in a shell script, it would help if the user could log in without entering a password. Later in this chapter, in the "Configuring Key-Based Authentication" section, you'll learn more about that.

Using scp to Copy Files Securely

Another part of the SSH suite you will definitely like is the `scp` command. You can use it to copy files securely. If you know how the `cp` command works, you'll also know how to handle `scp`. It is just the same, with the only exception that it works with a complete reference including the host name and username of the file you want to copy. Consider the following example:

```
scp /some/file linda@AMS.sandervanvugt.com:/some/file
```

This easy-to-understand command would copy `/some/file` to AMS.sandervanvugt.com and would place it in the directory `/some/file` on that host. Of course, it is possible to do the inverse: `scp root@SFO.sandervanvugt.com:/some/file /some/file` would copy `/some/file` from a remote host to the local host.

Using sftp for Secured FTP Sessions

As an alternative to copying files with `scp`, you can use the `sftp` command to connect to servers running the sshd program and establish a secured FTP session with such a server. From the `sftp` command, you have an interface that really looks a lot like the normal FTP client interface. All the commands you are used to working with in a classic FTP interface work here as well, with the only difference that in this case it is secured. For example, you can use the `ls` and `cd` commands to browse to a directory and see what files are available. From there, you can use the `get` command to copy a file to the local current directory. Figure 18-1 shows an example of this.

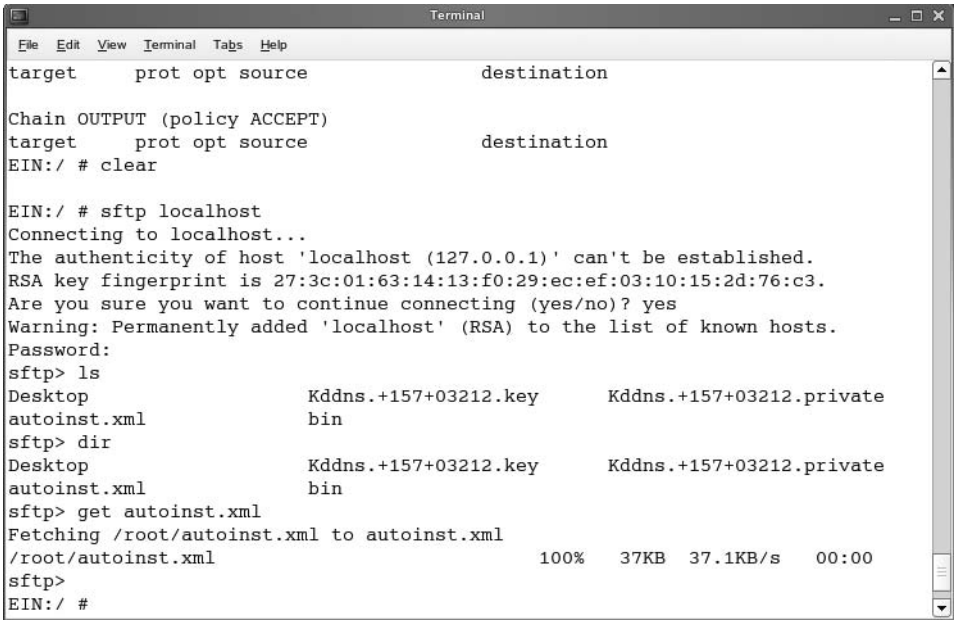

Figure 18-1. *From an* `sftp` *session, you can do all the things you are used to doing from a normal FTP session.*

Configuring SSH

In an SSH environment, a node can be a client and a server at the same time. So as you can imagine, both of these aspects have a configuration file. The client is configured in `/etc/ssh/ssh_config`, and the server has its configuration in `/etc/ssh/sshd_config`. Setting options for the server isn't hard to understand; just set them in `/etc/ssh/sshd_config`. For the client settings, however, the situation is more complicated, because you have several ways of overwriting the default client settings:

- `/etc/ssh/ssh_config` is a generic file that is applied to all users initiating an SSH session. The settings in this file can be overwritten by individual users who create an `.ssh_config` in the directory `.ssh` in their home directory.
- An option in `/etc/ssh/ssh_config` has to be supported by the `sshd_config` file on the server to which you are connecting. For example, if you are allowing password-based authentication from the client side but the server doesn't allow it, it will not work.
- Options in both files can be overwritten by using command-line options.

Table 18-1 gives an overview of the most useful options for `ssh_config`.

Table 18-1. *Most Interesting Options in* `ssh_config`

Option	Description
`Host`	This option applies the following declarations (up to the next `Host` keyword) to a specific host. Therefore, this option is applied on a host to which a user is connecting. The host name is taken as specified on the command line. Use this parameter to add some extra security to specific hosts. It is possible to use wildcards such as * and ? to refer to more than one host name.
`CheckHostIP`	If this option is set to `yes` (which is the default value), SSH will check the host IP address in the `known_hosts` file. Use this as a protection against DNS or IP address spoofing.
`Ciphers`	This option can have multiple values to specify the order in which the different encryption algorithms should be tried in an SSH version 2 session.
`Compression`	This option, which can have the values `yes` or `no`, specifies whether to use compression. The default is `no`.
`ForwardX11`	This useful option specifies whether X11 connections will be forwarded. If set to `yes`, graphical screens from an SSH session can be forwarded over the secure tunnel. The result is that the `DISPLAY` environment variable that determines where to draw graphical screens is set correctly. If you don't want to enable X-forwarding by default, you can use the option `-X` on the command line when establishing an SSH session.
`LocalForward`	Specifies that a TCP/IP port on the local machine is forwarded over SSH to the specified port on a remote machine. See the "Using Generic TCP Port Forwarding" section later in this chapter for more details.
`LogLevel`	Use this option to specify the verbosity level for log messages. The default value is `INFO`. If this doesn't go deep enough, `VERBOSE`, `DEBUG`, `DEBUG1`, `DEBUG2`, and `DEBUG3` will provide more information.
`PasswordAuthentication`	Use this option to specify whether you want to use password authentication. By default, you can use password authentication. In a secure environment where keys are used for authentication, you can safely set this option to `no` to disable password authentication completely.
`Protocol`	This option specifies the protocol version that SSH should use. The default value is set to 2,1; version 2 is used first and if that doesn't work, version 1 is tried. It is a good idea to disable version 1 completely, because it has some known security issues.
`PubkeyAuthentication`	Use this option to specify whether you want to use public key–based authentication. This option should always be set to the default value `yes`, because public key–based authentication is the safest way of authenticating.

The counterpart of `ssh_config` on the client computer is the `/etc/ssh/sshd_config` file on the server. Many options that are used in the `ssh_config` file can be used in the `sshd_config` file as well. Some options, however, are specific for the server side of SSH. Table 18-2 gives an overview of some of these options.

Table 18-2. *Most Important Options in* `sshd_config`

Option	Description
`AllowTcpForwarding`	Use this option to specify whether you want to allow clients to do TCP port forwarding. Since this is a useful feature, you probably want to leave it to its default value, `yes`.
`Port`	Specifies the port on which the server is listening. By default, `sshd` is listening on port 22.
`PermitRootLogin`	Use this option to specify whether you want to allow root logins. To add additional security to your server, consider setting this option to `no`. If set to `no`, the `root` user has to establish a connection as a regular user and from there use `su` to become `root` or use `sudo` to perform certain tasks with root permissions.
`PermitEmptyPasswords`	This option specifies whether you want to accept users coming in with an empty password. From a security perspective, this might not be a good idea; therefore, the default value `no` suits in most cases. If you want to run SSH from a script and establish a connection without entering a password, however, it can be useful to change the value of this parameter to `yes`.
`X11Forwarding`	Use this option to specify whether you want to allow clients to use X11-forwarding. On SUSE, the default value for this parameter is `yes`.

Using Key-Based Authentication

Now that you know all about the basic use of SSH, it's time to look at some of the more advanced options. One of the most important of these options is key-based authentication. To use this kind of authentication, SSH uses public/private key–based authentication. Before diving into the configuration of key-based authentication, you'll learn how you can use these keys.

Introducing Cryptography

In general, you can use two methods for encryption: symmetric and asymmetric encryption. Symmetric encryption is fast, but not so secure. Asymmetric encryption is slower but more secure. In a symmetric key environment, both parties use the same key to encrypt and decrypt messages. In an asymmetric key environment, a public/private key pair is used. The latter is the important technique that is used for SSH.

If asymmetric keys are used, every user needs his own public/private key pair, and every server needs a pair of them as well. Of these keys, the private key must be protected by all means. If the private key gets compromised, the identity of the owner of the private key gets compromised as well. Therefore, a private key ordinarily is stored in a secure place where no one can access it besides the owner of the key. The public key on the contrary is available to everyone.

You can use public/private keys, generally speaking, for two purposes. The first of them is to send encrypted messages. In this scenario, the sender of the message encrypts the message with the public key of the receiver of the message, and the receiver of the message is the only one who can decrypt the message with the matching private key. This scenario requires of course that before sending an encrypted message, you need to have the public key of the person to whom you want to send the message.

The other option is to use public/private keys for authentication or to prove that a message has not changed since it was created. The latter is also known as *nonrepudiation*. In the example of authentication, the private key generates an encrypted token, the *salt*. If this salt can be decrypted

with the public key of the person who wants to authenticate, then there is enough proof that a server is really dealing with the right person; therefore, access can be granted. This technique requires the public key to be copied to the server before any authentication can happen, however.

Using Public/Private Key–Based Authentication in an SSH Environment

When you use SSH key-based authentication, you have to make sure that, for all users who need to use this technology, the public key is available on the servers where they want to log in. When logging in, the user creates an authentication request that is signed with his private key. This authentication request is matched to the public key of the same user on the server where that user wants to authenticate. If it matches, the user is allowed to come in; if it doesn't, the user is denied access.

Public/private key–based authentication is enabled by default on SUSE Linux Enterprise Server; therefore, only when no keys are present will the server prompt the user for a password. The following summarizes what happens when a user tries to establish an SSH session with a server:

1. If public key authentication is enabled, which by default is the case, SSH checks the `.ssh` directory in the user's home directory to see whether a private key is present.
2. If a private key is found, SSH creates a packet with some data in it (the salt), encrypts that packet with the private key, and next sends it to the server. With this packet, the public key is sent as well.
3. The server now checks whether a file with the name `authorized_keys` exists in the home directory of the user. If it doesn't, the user cannot authenticate with his keys. If this file does exist and the public key is an allowed key and also is identical to the key that was previously stored on the server, the server uses this key to check the signature.
4. If the signature could be verified, the user is granted access. If it didn't work out, the server will prompt the user who tries to connect for his password.

All this sounds pretty complicated, but it isn't. Everything is happening transparently, if everything has been set up correctly. Also, you won't even notice a delay. All this ordinarily happens in less than a second.

Setting Up SSH for Key-Based Authentication

The best way to explain how to set up SSH for key-based authentication is by showing an example. In the following procedure, key-based authentication is enabled for the user `root`:

1. On the desktop where `root` is working from, use the command `ssh-keygen -t dsa -b 1024`. This generates a 1,024-bit public/private key pair. Listing 18-2 shows what happens.

Listing 18-2. *Generating a Public/Private Key Pair with* `ssh-keygen`

```
workstation # ssh-keygen -t dsa -b 1024
Generating public/private dsa key pair.
Enter file in which to save the key (/root/.ssh/id_dsa) :
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_dsa.
Your public key has been saved in /root/.ssh/id_dsa.pub.
The key fingerprint is:
59:63:b5:a0:c5:2c:b5:b8:2f:99:80:5b:43:77:3c:dd root@workstation
```

I'll explain what happens now. The user in this example uses the `ssh-keygen` command to generate a public key and a private key. The type encryption algorithm used to generate this key is DSA, which is considered more secure than its alternative RSA. The option `-b 1024` specifies that 1,024-bit encryption should be used for this key. The longer this number, the more secure it will be. Notice, however, that a many-bits encryption algorithm will also require more system resources to use it. After generating the keys, the command asks you where to save it. By default, it will create a directory with the name `.ssh` in your home directory, and in this directory it creates the file `id_dsa`. This file contains the private key.

Next, you are prompted to enter a passphrase. This passphrase is an extra layer of protection that you can add to the key. Since anyone who has access to your private key (which isn't that easy to do) can forge your identity, your private key should always be passphrase protected. After entering the same passphrase twice, the private key is saved, and the related public key is generated and saved in the file `/root/.ssh/id_dsa.pub`. Also, a key fingerprint is generated. This fingerprint is a summary of your key, a checksum that is calculated on the key to see whether anything has happened with the key.

2. After creating the public/private key pair, you must transfer the public key to the server. The ultimate goal is to get the contents of the `id_dsa.pub` file in the file `/root/.shh/authorized_keys`. You can, however, not simply copy the file to the destination file `authorized_keys`; this is because other keys may already be stored in that file. Therefore, first use `scp` to copy the file to a temporary location. The command `scp /root/.ssh/id_dsa.pub root@server:/root/from_workstation_key.pub` will do the job.
3. Now that the public key is on the server, you have to put it in the `authorized_keys` file. Before doing this, make sure the directory `.ssh` exists on the server in the home directory of the user `root`, that is, has user and group `root` as its owner and the permission mode 700. Then, on the server with the directory `/root` as your current directory, use `cat from_workstation_key.pub >> .ssh/authorized_keys`. This appends the content of the public key file to the `authorized_keys` file, thus not overwriting any file that may have been there already.
4. If no errors occurred, you were successful! Return to your workstation, and start an SSH session to the server where you have just copied your public key to the `authorized_keys` file. You will notice that you aren't prompted for a password anymore; you are prompted for a passphrase instead. This proves you were successful. Notice, however, that you need to repeat this procedure for every server you want to be able to establish a session with that is secured with keys.

Working with keys as described is an excellent way to make SSH authentication more secure. It has a drawback, though: if from a shell script or cron job you need to establish an SSH session automatically, it is not practical to be prompted for a key first. Therefore, you need some method to execute such jobs automatically. One solution is to create a special user account with limited permissions. If you have such an account, it doesn't hurt if that user account is using a public/private key pair without a passphrase assigned to the private key. Another solution is to run `ssh-agent`, which caches the keys before they are used. In the next section, you will learn how that works.

Caching Keys with ssh-agent

To prevent yourself from entering private keys all the time, you can use ssh-agent. This useful program caches keys for a given shell environment. After starting ssh-agent for a given shell, you need to add the passphrase for the private key you want to use. This is something you will do for a specific shell, so after you close that specific shell or load another shell, you need to add the passphrase to that shell again.

After adding a passphrase to ssh-agent, the passphrase is stored in RAM. It is stored in a way that it cannot be accessed; only the user who added the key to RAM is able to read it from there. Also, ssh-agent listens only to the ssh and scp processes that were started locally, so you have no way to access a key that is kept by ssh-agent over the network. So, you can be sure that using ssh-agent is pretty secure. Apart from being secure, it is pretty easy to do as well. Enabling ssh-agent and adding a passphrase to it is just a simple two-step procedure:

1. From the shell prompt, use ssh-agent, followed by the name of the shell you want to use it for. For example, use `ssh-agent /bin/bash` to activate ssh-agent for the bash shell.
2. Now type `ssh-add`. This will prompt you for the passphrase of your current private key. As the result of this action, you'll see the message identity added, followed by the private key of which the passphrase is added to ssh-agent.

■ **Tip** SSH is a great method to get access to other hosts. But did you know you can also use it to mount a file system on a remote system? All modern versions of SSH support this feature: just use sshfs, which gives access to all files and directories on the remote server that as a normal user on that server you can access. If you know how to mount a directory with `mount`, working with sshfs is easy; for example, the command `sshfs linda@AMS:/data /mnt/AMS` would give access to the `/data` directory on the remote server and connect that directory to `/mnt/AMS` on the local server.

Tunneling Traffic with SSH

Apart from establishing remote login sessions, copying files, and executing commands on remote hosts, it is possible to use SSH for TCP port forwarding. This way, SSH is used as a simple VPN solution, with the capability of tunneling almost any nonsecured protocol over a secured connection. In the following sections, I'll first talk about X-forwarding and then you can read how to forward almost any protocol using SSH.

Using X-Forwarding

Wouldn't it be useful if you could start an application on a server, where all the workload is performed by the server while you can do the work itself from your client? You can with SSH X-forwarding. When using X-forwarding, you first establish an SSH session to the server to which you want to connect. Next, from this SSH session, you'll start the graphical application. This application will draw its screen on your workstation while doing all the work on the server.

Sound good? Establishing such an environment has only two requirements:

- Make sure the option `X11Forwarding` is set to `yes` in `/etc/ssh/sshd_config` on the server.
- Connect to the server with the `ssh -X` command from your client. Alternatively, you can set the option `X11Forwarding` in the client configuration file `/etc/ssh/ssh_config`, which allows you to forward graphical sessions by default. Since, however, this poses a minor security threat, this setting is not enabled by default on SUSE Linux Enterprise Server.

Now that you have established the SSH session with your server, start any command you want to use. This even allows you to run YaST from a Debian workstation!

■ **Note** Forwarding X sessions with SSH is really cool, but it has a limitation. You need an X-server on the client from which you are establishing the SSH session. On Linux, Unix, or the Mac, this is not a problem since an X-server is available for each of these operating systems. On Windows, however, this is a problem. The most-used SSH client for Windows is putty, which a useful client, but it doesn't contain an X-server. If you want to use an X-server that runs on Windows, use Cygwin/X. You can find this free X-server for Windows at `http://x.cygwin.com`.

Using Generic TCP Port Forwarding

X is the only service for which port forwarding is hard-coded in the SSH software. For everything else, you need to do it by hand, using the `-L` or the `-R` option. Refer to the example in Figure 18-2.

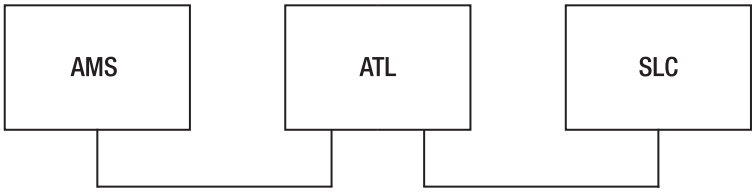

Figure 18-2. *Example network*

The example network shown in Figure 18-2 has three nodes. Node AMS is the node where the administrator is working. ATL is the node in the middle. AMS has a direct connection to ATL but not to SLC, which is behind a firewall. ATL, however, does have a direct connection, not hindered by any firewall, to SLC.

An easy example of port forwarding is the command:

```
linda@AMS:~> ssh -L 4444:ATL:110 linda@ATL
```

In this example, user `linda` forwards connections to port 4444 on her local host to port 110 on the host ATL as user `linda` on that host. This is what you would use, for example, to establish a secure session to the insecure POP service on that host. The local host first establishes a connection to the SSH server running on ATL. This SSH server connects to port 110 at ATL, whereas SSH binds to port 4444 on the local host. Now an encrypted session is established between local port 4444 and server port 110; everything sent to port 4444 on the local host would really go to port 110 at the server. For example, if you would configure your POP mail program to get its mail from local port 4444, it would really get it from port 110 at ATL. Notice this example uses a nonprivileged port. Only user `root` can connect to a privileged port with a port number less than 1024. No matter what port you are connecting to, you should always check in the configuration file `/etc/services`, where port numbers are matched to names of services if the port number is already in use by some other process, and use `netstat -patune | grep <your-intended-port>` to make sure the port is not already in use.

A little variation on the local port forwarding shown earlier is remote port forwarding. If you wanted to do that, you would forward all the connections to a given port on the remote port to a local port on your machine. For example, use the `-R` option as in the following example:

```
linda@AMS:~> ssh -R 4444:AMS:110 linda@ATL
```

In this example, user `linda` connects to host ATL (see the last part of the command). On this host, port 4444 is addressed by using the construction `-R 4444`. This remote port is redirected to port 110 on the local host. As a result, anything going to port 4444 on ATL is redirected to port 110 on AMS. This example would be useful if ATL were the client and AMS were the server running a POP mail server to which `linda` wants to connect.

Another useful example is when the host you want to forward to cannot be reached directly, for example because it is behind a firewall. In that case, you can establish a tunnel to another host that is reachable with SSH. Imagine that in the example in Figure 18-2, the host SLC is running a POP mail server that user `linda` wants to connect to; this user would use the following command:

```
linda@AMS:~> ssh -L 4444:SLC:110 linda@ATL
```

In this example, `linda` forwards connections to port 4444 on her local host to server ATL that is running SSH. This server would forward the connection to port 110 on server SLC. Note that in this scenario, the only requirement is that ATL has the SSH service activated; no sshd is needed on SLC for this to work. Also note that there is no need for host AMS to get in direct contact with SLC, because this would happen from host ATL.

In the previous examples, you learned how to use the SSH command to do port forwarding. This isn't your only way of doing it. If you need to establish a port-forwarding connection all the time, you can put it in the SSH configuration file on the client computer. Put it in `.ssh/config` in your home directory if you want it to work for your user account only or in `/etc/ssh/ssh_config` if you want it to apply for all users on your machine. The parameter you should use as an alternative to `ssh -L 4444:ATL:110` is as follows:

```
LocalForward 4444 ATL:110
```

Using Other Methods for Remote Access

Although certainly it's the most secure and reliable method for remote access, SSH isn't the only way you can manage your server remotely. Another method that is rather popular is VNC, which allows you to take over a complete desktop remotely. In the following sections, you'll learn how to configure your server for VNC, how to use VNC remotely, and how to use `screen` to establish a remote session in which screens are synchronized, which is an ideal solution for helping people remotely.

Using VNC for Remote Access to Graphical Screens

Enabling VNC is easy: the YaST Remote Administration option in the Network Services section allows you to set up VNC access quickly. As shown in Figure 18-3, this module gives access to two choices: Allow Remote Administration and Do Not Allow Remote Administration. Want to enable remote administration for your server? Just click Allow Remote Administration, and you are almost there. If you have an active firewall protecting your server, don't forget to select Open Port in Firewall. This option is available only if the firewall on your server is really enabled; if it isn't, the option is disabled. Then click Finish to complete the setup for remote administration.

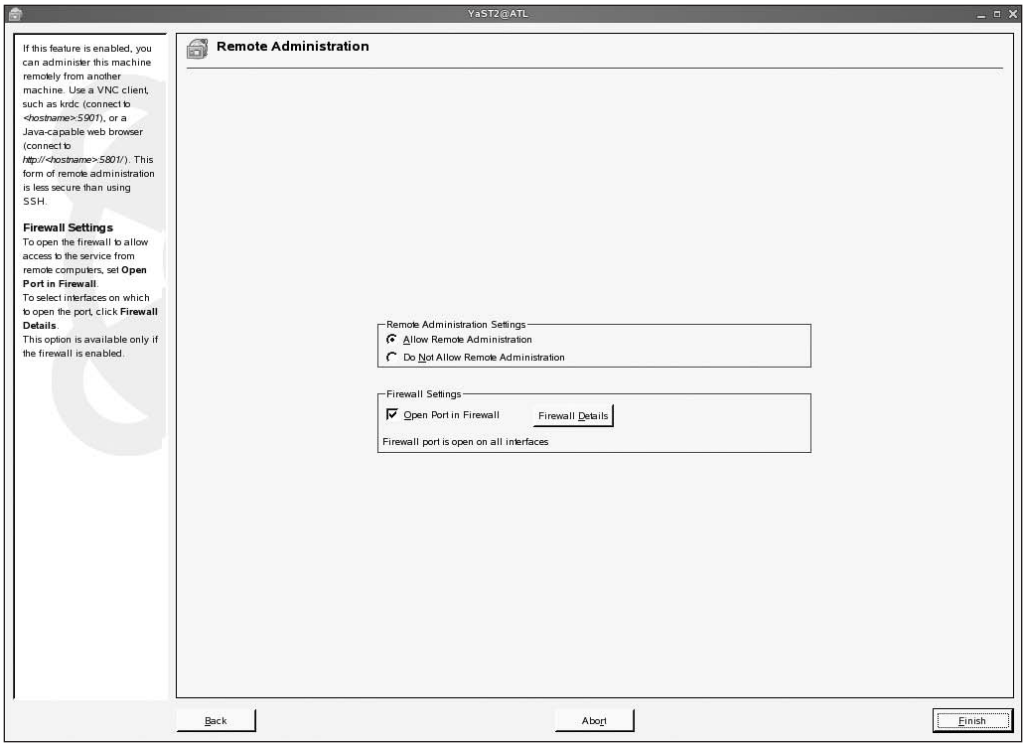

Figure 18-3. *You can easily set up VNC remote administration using YaST.*

Clicking Finish isn't the final step to make remote administration possible for your server. This is because the component that is used to log in to your server has to be enabled for remote administration as well. Depending on the graphical desktop environment you are using, this component is xdm (generic for X), kdm (for KDE), or gdm (for the GNOME desktop environment). Remote administration of your server is possible only after this component is restarted. On SUSE Linux Enterprise Server, you can accomplish this by using the command `rcxdm restart` from a shell command line. Next, your server is enabled for VNC remote access.

You can now connect with a Java-enabled web browser to VNC port 5801 on your server by going to `http://yourserver:5801`. As an alternative, you can use a dedicated VNC client from either Linux or Windows to connect to port 5901 at the remote server. As a result, you will see a login prompt that you can connect to, and that will give you access to the remote server (see Figure 18-4).

You should note that there are some small differences between the normal login interface that XDM is offering you and the interface offered when accessing a server via VNC. The latter option offers an Administration button. If you click this button, you are asked for the password of user `root`. After entering it, you are redirected to YaST, which is available as your only option to administer the remote system.

Figure 18-4. *VNC gives access to the graphical login of a server from a browser or a dedicated VNC client.*

Enabling VNC via xinetd

Using the remote administration option from YaST is one way to enable VNC. There is also another way that offers some more advantages, and that is to enable VNC via `xinetd`. Since xinetd is the subject of the next chapter; I will not cover the details of this configuration here. You should, however, know that there are some advantages when using xinetd to configure access to VNC. The most important of these is that some more access control is possible. When combining xinetd with TCP Wrapper, you can specify exactly what hosts you do want to give access to and what hosts you don't. Check Chapter 19 for more details on how this works.

Securing VNC Remote Access with SSH

Setting up your server for remote administration is one thing; making sure this remote administration happens in a secure way is another thing. By default, VNC traffic is sent over the network unencrypted. Tunneling VNC over SSH is an easy solution for that problem:

1. On your workstation, use the command `ssh -X root@yourserver -L 5901:yourserver:5901`. This makes sure that all traffic addressed to local port 5901 is forwarded to port 5901 on your server.
2. Now from your workstation, use a tool like vncviewer to connect to the local VNC port. Note that you shouldn't connect to port 5901 but to port 1, which is an internal VNC port.

You can now access VNC, just like you did when connecting to it without encryption. The only difference is that when the tool asks for passwords, they aren't sent in plain text over the network anymore.

Using screen to Synchronize Remote Sessions

Ever tried to imagine what someone is seeing while working on a remote system? Don't imagine! With `screen`, you can see just what happens. The idea is simple: the user on the remote server uses the `screen` command. Next, you use `screen -x` from an SSH session to attach to that screen. The next step is that everything the user in question types into his console is displayed in your SSH session as well, and everything you are typing shows up on his screen. You are in fact sharing the same screen in this scenario. The next procedure shows how to set this up. Note that this is just one of the many uses of `screen`. Check its `man` page for more information about this versatile utility:

1. On the server, just type `screen` in a terminal window.
2. From a client, establish an SSH session to the server. Any plain SSH session will do fine.
3. Now from the client, type `screen -x`. This gives a list of screens to which you can connect (see Listing 18-3).

Listing 18-3. *Example of Screen Usage*

```
AMS:~ # screen -x
There are several suitable screens on:
          7068.pts-1.AMS  (Attached)
          7188.pts-6.AMS  (Attached)
Type "screen [-d] -r [pid.]tty.host" to resume on of them.
```

4. Read the message that `screen -x` is giving you, and next connect to one of the screen sessions that are mentioned. In this example, you can do that by using the command `screen -x -r 7068.pts-1.AMS`. This command will connect you to the console window where `screen` is running on the server. Enjoy!

Summary

If people are talking about remote administration or remote access, they are probably talking about SSH, which is the real standard for remote administration on Linux. In this chapter, you learned everything you should know to manage your server remotely with SSH. However, SSH isn't the only option you can use to manage your server remotely. Another popular solution is VNC, which you can enable from YaST easily. You not only learned how to do this but also how to use VNC securely by tunneling it with SSH. Finally, you read about another useful tool, the `screen` command. In the next chapter, you will read how to enable lots of network services by using the Xinetd "superdaemon."