

SSL Tunnels

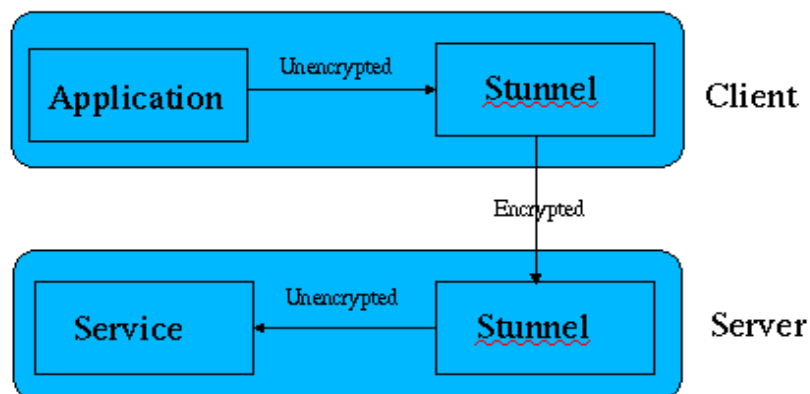
Introduction

As you probably know, SSL protects data communications by encrypting all data exchanged between a client and a server using cryptographic algorithms. This makes it very difficult, if not impossible, for hackers to extract information from network traffic, even if they manage to capture data packets using capturing tools like tcpdump or ethereal (recently renamed WireShark).

While many network services, such as web servers, support using SSL directly, other services do not. The stunnel application is designed to allow administrators to protect those services that do not natively support SSL encryption, without needing to modify the original service in any way.

If the client application supports SSL, then no changes are needed at that end. However, you can also pass data through an SSL tunnel even if the client side application does not support SSL. To make this work, you must run the stunnel program both on the local client and also on the remote server.

The client side stunnel will be configured to accept an incoming (unencrypted) data on a specific port. Whenever it receives this data, it will encrypt the data and forward it to the stunnel program running on the server. The server's stunnel program will then decode the data back into its original format and then forwards the decrypted data to the actual service's port. Of course it also captures any data returned by the service, encrypts the responses and send the returned data back to the client's stunnel.



The advantage to this is that neither the application, nor the service, need to be modified in any way, yet your data is protected. You will need to know the port number the application uses to communicate with the service of course. There is one other advantage to using an SSL tunnel in this manner. If desired, you can configure the server to only accept incoming connections from clients that have a known SSL certificate. To do that you must generate new certificates for the client and copy the resulting file to both the server and the client in some secure fashion, such as via an SSH connection, an SSL web page, or perhaps via hand delivered floppy disk.

Installing STunnel

The stunnel program is available for both UNIX and Windows based systems. Visit <http://www.stunnel.org> to download either the source code, or a pre-built binary version for your computers. You will also need a version of OpenSSL, which is a library that performs the actual encryption/decryption of the data. Most Linux distributions include a copy of OpenSSL. If you do not have it or you can download the OpenSSL from <http://www.openssl.org>. The Windows version of the stunnel program already includes the required OpenSSL library.

If there is a binary package available for your system (RPM, DEB, etc...), use your Linux package management tools to install. For other Linux systems, you must build and install the stunnel program following the standard 5 step process:

```
$ tar xzvf <path>/stunnel-4.04.tar.gz
$ cd stunnel-4.04
$ ./configure
$ make
$ make install
```

Now that stunnel is installed, it is time to configure it. You should have two computers available at this point, one to act as the client and another as the server.

Configuring STunnel

You can run stunnel two different ways. You can setup your network super-server, inetd (or xinetd) to run the stunnel program only when a new incoming connection is received, or you can run stunnel in daemon mode which means it runs continuously. The server based mode is faster and a bit easier, so let's setup an SSL tunnel to protect the IMAP service.

First we need to setup a configuration file that controls the options used by stunnel. Create the file **/usr/local/etc/stunnel/stunnel-imap.conf** and add the following lines:

```
# IMAP over SSL configuration file

cert = /usr/local/etc/stunnel/stunnel.pem
setuid = nobody
setgid = nobody
chroot = /usr/local/etc/stunnel
pid = /stunnel-imap.pid

# Optional entries
output = /var/log/stunnel-imap.log
debug = 7

[imaps]
accept = 993
connect = 143
```

Explanation:

The file **/usr/local/etc/stunnel/stunnel.pem** will be used as the certificate for encryption and/or authentication. This certificate was generated when you built stunnel, but you can create additional certificates if needed.

The *setuid* and *setgid* lines force stunnel to assume the identity of the named user and group, which also adds a measure of extra security should a hacker gain control over the stunnel program. In this case, the hacker would only gain the privileges of the user named "nobody".

The *chroot* line makes stunnel treat the directory **/usr/local/etc/stunnel** directory as its root directory. This adds a bit of protection should a hacker manage to gain control over the stunnel program itself. Even if this should happen, the hacker could only gain access to files and directories under the **/usr/local/etc/stunnel** directory, which should limit the amount of possible damage they can perform. Remember that the user listed under the *setuid* entry must

be able to read and create files in this directory, which means you probably should change the ownership of the directories and files to match the *setuid* user account.

The *pid* line tells stunnel to store its process id in the named file. That can be used by scripts to kill and restart the stunnel program. Keep in mind that this file will be created using the chroot entry. In our example, this creates the file ***/usr/local/etc/stunnel/stunnel-
imap.pid***.

The *output* and *debug* entries force stunnel to use debug level 7 (lots of messages) and saves its messages to ***/var/log/stunnel-
imap.log***. If you do not have these entries, then stunnel will normally log its messages to the ***/var/log/messages*** file using your standard syslog daemon.

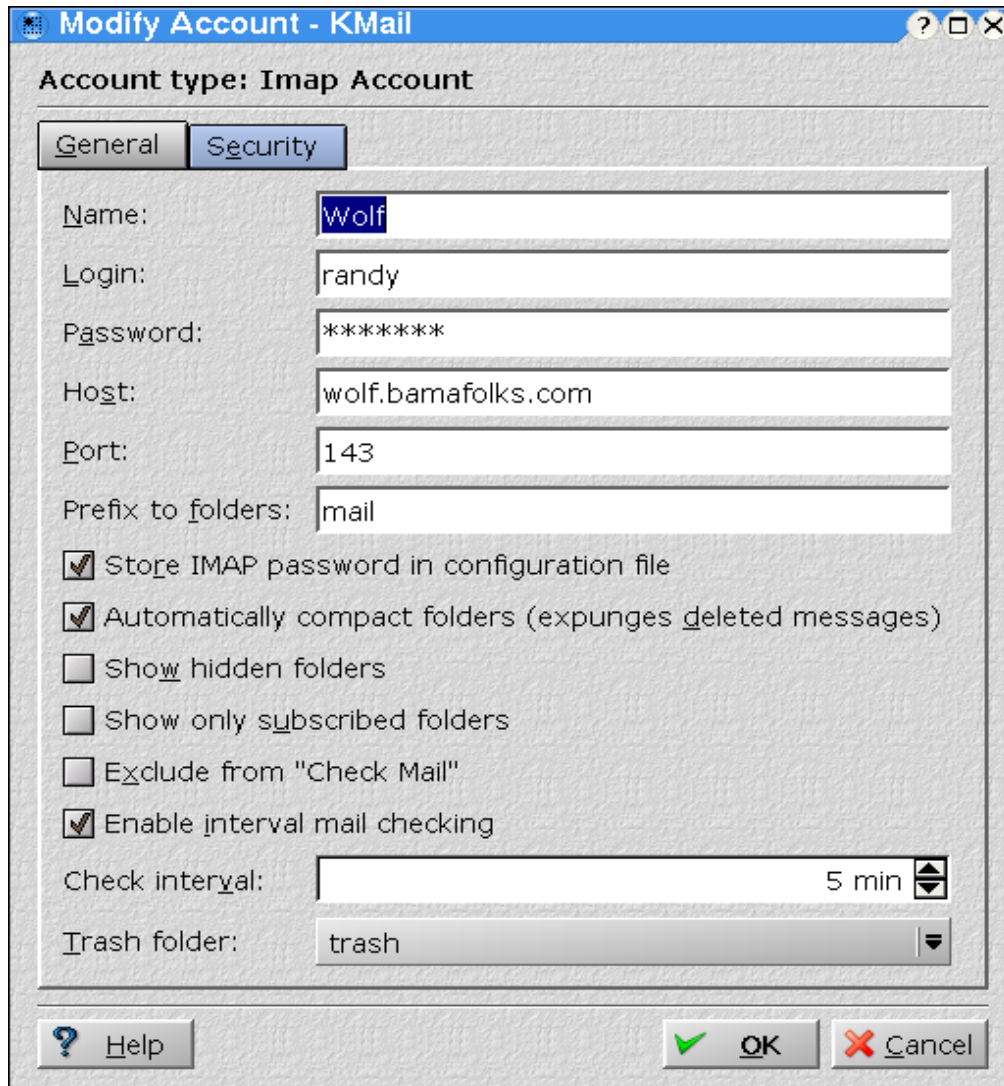
Finally, the *[imaps]* section tells stunnel to accept incoming SSL encrypted data on port 993 and to send the data to port 143 (standard IMAP) after it is decrypted. The name of the service *imaps* must match the name of the port from your ***/etc/services*** file for this to work correctly.

Now that we have a valid configuration file, we can start the stunnel program using a command like this:

```
/usr/local/sbin/stunnel /usr/local/etc/stunnel/stunnel-  
imap.conf
```

We are now ready to test the tunnel. If your e-mail program support the SSL protocol, you can begin using port 993 to connect to the server instead of port 143. KMail has SSL support built-in, so it can begin using the new SSL tunnel directly.

Original Configuration (IMAP without SSL):



Modify Account - KMail ? □ ×

Account type: **Imap Account**

General | **Security**

Name:

Login:

Password:

Host:

Port:

Prefix to folders:

Store IMAP password in configuration file

Automatically compact folders (expunges deleted messages)

Show hidden folders

Show only subscribed folders

Exclude from "Check Mail"

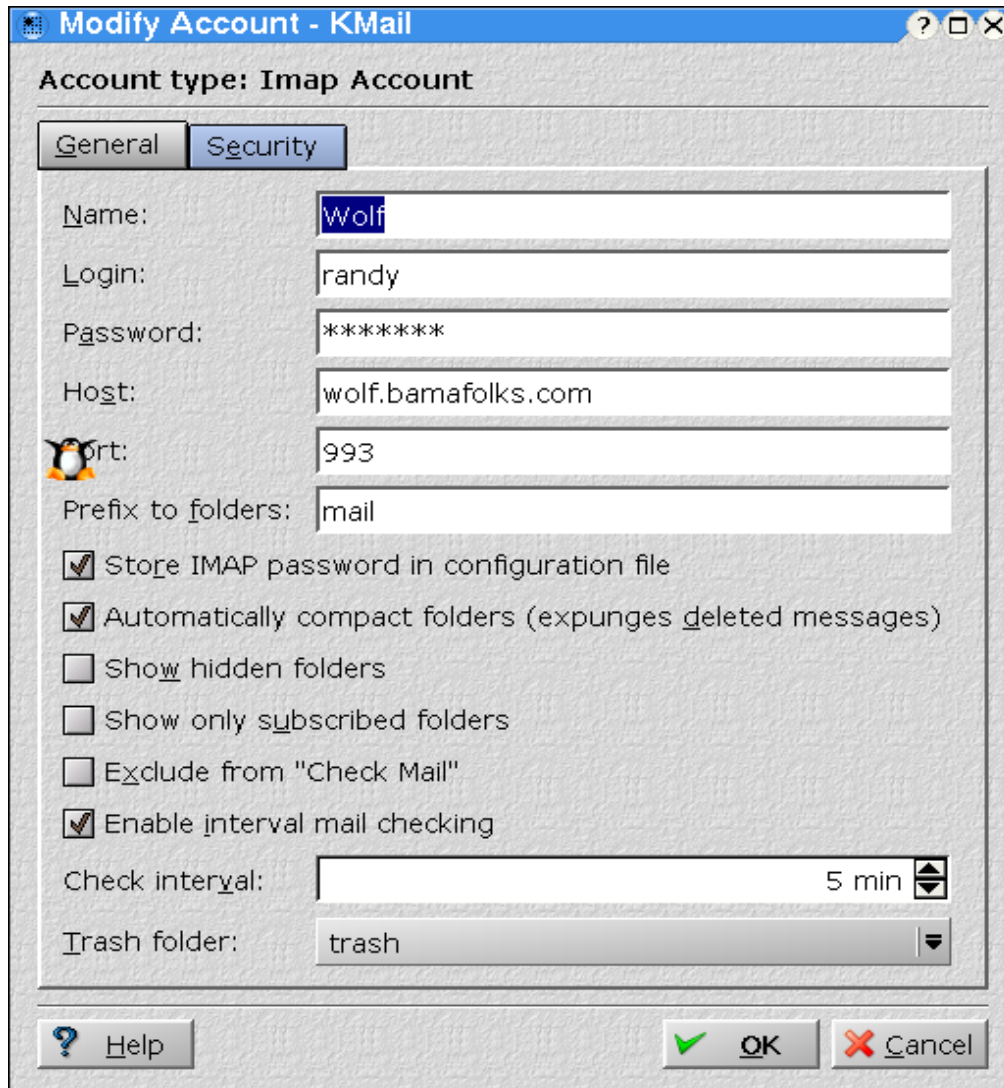
Enable interval mail checking

Check interval: 5 min

Trash folder:

? Help OK Cancel

New Configuration (IMAP with SSL):



The screenshot shows the 'Modify Account - KMail' dialog box with the 'Security' tab selected. The account type is 'Imap Account'. The fields are filled with the following information:

Name:	Wolf
Login:	randy
Password:	*****
Host:	wolf.bamafolks.com
Port:	993
Prefix to folders:	mail

Below the fields are several checkboxes:

- Store IMAP password in configuration file
- Automatically compact folders (expunges deleted messages)
- Show hidden folders
- Show only subscribed folders
- Exclude from "Check Mail"
- Enable interval mail checking

The 'Check interval' is set to 5 min, and the 'Trash folder' is set to trash. At the bottom, there are buttons for Help, OK, and Cancel.

However, if your e-mail program does not understand SSL natively, then this will not work correctly. In that case, you also need to run a client side stunnel. Let's do that next.

Configuring a Client-Side Tunnel

This is very similar to setting up the server side. First you must download and install the stunnel program of course. The only real difference in the configuration is the direction of the port numbers. Create the file **/usr/local/etc/stunnel/stunnel-imap.conf** on the client. Add the following lines:

```
# Client IMAP over SSL configuration file

cert = /usr/local/etc/stunnel/stunnel.pem
client = yes
chroot = /usr/local/etc/stunnel
pid = /stunnel-imap.pid
setuid = nobody
setgid = nobody

# Optional entries
output = /var/log/stunnel-imap.log
debug = 7

[imap2]
accept = 143
connect = wolf.bamafolks.com:993
```

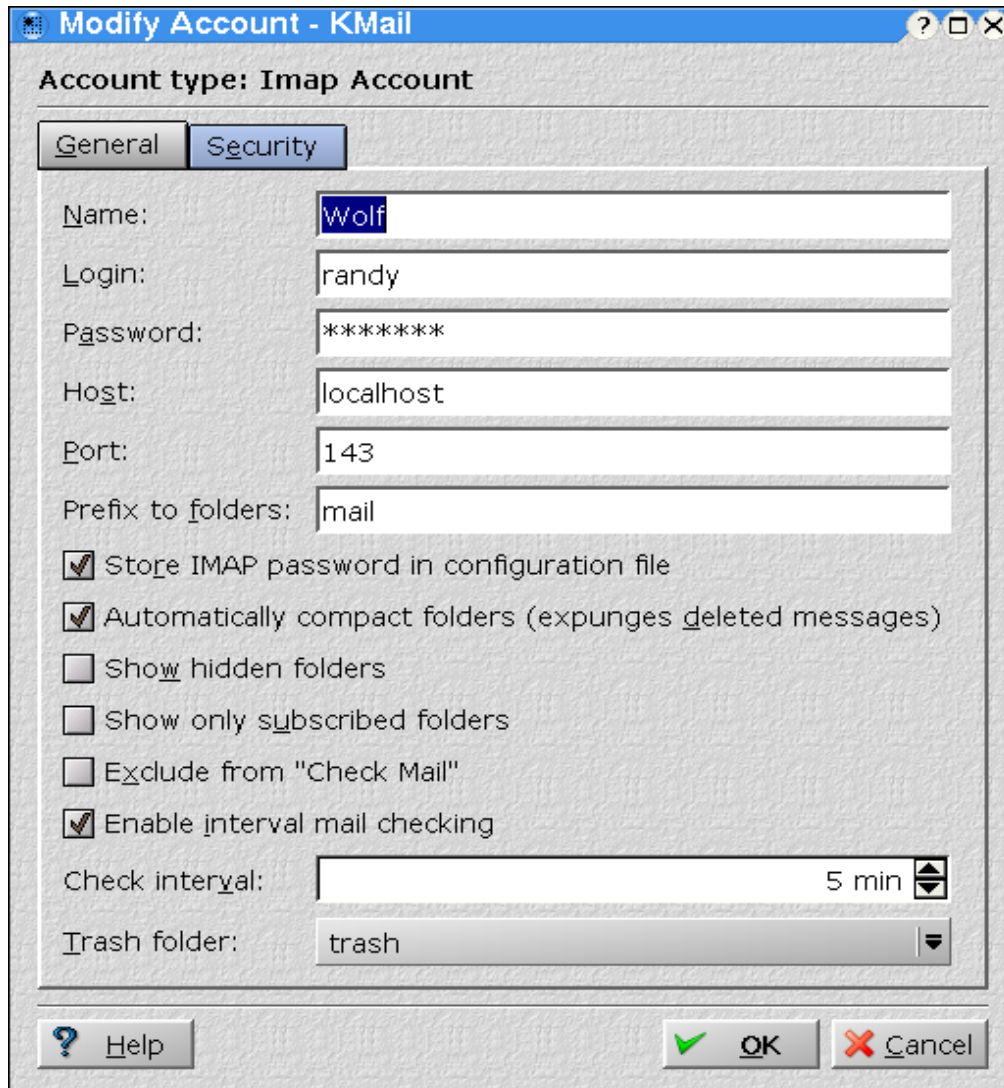
We have added the new *client* line and of course we have a section called *[imap2]*. It uses reversed port numbers for the *accept* and *connect* lines. You should also notice how the connect line specifies a remote computer name and port number.

Next, run the stunnel program on the client using a command like this:

```
/usr/local/sbin/stunnel /usr/local/etc/stunnel/stunnel-imap.conf
```

Now you can use the SSL tunnel to communicate with the IMAP server even if the e-mail program does not understand SSL. In this case, point your e-mail program to the local computer's port 143. The locally running stunnel will then make a connection to the remote computer's port 993, which is also running stunnel. The remote stunnel will in turn connect to its local port 143 and even though you have not asked your e-mail program to use SSL, all network packets are encrypted and the communications between the e-mail client and e-mail server are secure.

KMail configuration (Using local stunnel)



You can use this kind of setup to encrypt almost any protocol, not just IMAP. You just need to determine the port number the service uses and run a tunnel at each end.

A few protocols are quite complex and cannot be run through an SSL tunnel in this manner, notably FTP and telnet. The SSH protocol includes replacement programs for both telnet and ftp, called ssh and sftp, so you should use those programs instead if you want security. You should also take a look at the scp utility, which can also transfer files securely using a syntax similar to the standard cp command.

Authentication Using STunnel

Now that we can establish SSL tunnels, we can look at restricting the users that can use the tunnels. This feature adds an extra level of security, since not only will the SSL certificate be used to encrypt the data, but the server will refuse to open a connection unless it recognizes the certificate the client is using.

Server Side Configuration

Let's modify our previous IMAP setup to require certificate verification. First, modify the server's stunnel configuration file to read as follows:

```
# IMAP over SSL configuration file

cert = /usr/local/etc/stunnel/server.pem
setuid = nobody
setgid = nobody
chroot = /usr/local/etc/stunnel
pid = /stunnel-imap.pid
verify = 3
CAfile = /etc/ssl/misc/demoCA/cacert.pem
CApath = /certs

# Optional entries
output = /var/log/stunnel-imap.log
debug = 7

[imaps]
accept = 993
connect = 143
```

If you examine this file closely, you will see only I made three new entries named *verify*, *CAfile*, and *CApath*.

The *verify* option controls the level of peer certificate authentication to perform when clients connect to the server. Possible values are:

- 1 - verify certificate if present
- 2 - verify certificate always
- 3 - verify certificate against locally installed versions
- default - no verification needed

Option 1 will test the client's certificate and reject the connection if invalid, but does not require the client to use one at all. In other words, you could comment out the *cert* line on the the client side at it will still work. Self-signed certificates are not considered valid however, so only certificates used by clients that are signed by a valid

CA will be allowed.

If you use option 2, then the client's certificate must both be available and valid to allow a connection.

Finally, option 3 restricts connections to clients that are using a certificate and that certificate has also been copied to the server. This is a bit trickier to setup, but is not too difficult.

The *CAfile* entry is required if you are creating self-signed certificates. Since I used the **/etc/ssl/misc/CA.pl** script to create both a new server and client certificates, I entered the full path to the demoCA's certificate file.

Finally, we need a place to store client certificates. I choose to create the subdirectory named **/usr/local/etc/stunnel/certs** for this purpose. If you use the *chroot* option, remember this directory must be found under the *chroot* directory. If you do not use the *chroot* option, then you should enter the full path to the directory where you store the client certificates.

Creating Certificates

At this point we need to create a new certificate for the client. If your Linux distribution includes tools to create and manage SSL certificates (for example, Yast under SUSE), then you should use that tool. If you do not have a SSL management tool, you should be able to adapt the following steps to your distribution:

I used the same scripts under the **/etc/ssl/misc** directory as we used to generate a certificate for our Apache server. I also generated a new server certificate using those same scripts.

Here are the commands need to generate the certificates in the correct format for use with stunnel:

Step 1: Make the `/etc/ssl/misc` directory the current working directory

```
$ cd /etc/ssl/misc
```

Step 2: Create a new Certificate Authority (only do this one time!)

```
$ ./CA.pl -newca  
{Enter values as needed here}
```

Step 3: Create a new request for a certificate (server)

```
$ ./CA.pl -newreq  
{Enter values as needed here}
```

Step 4: Sign the request and create the actual certificate file.

```
$ ./CA.pl -sign  
{Enter pass phrase and answer questions here}
```

Step 5: Remove the pass phrase (causes problems with stunnel)

```
$ openssl rsa -in newreq.pem -out newreq_nopasswd.pem
```

Step 6: Combine the request and the certificate into one file

```
$ cat new_reqnopasswd.pem newcert.pem > server.pem
```

Step 7: Copy the new certificate to the correct location

```
$ cp server.pem /usr/local/etc/stunnel
```

Step 8: Create a new request for a certificate (client)

```
$ ./CA.pl -newreq  
{Enter values as needed}
```

Step 9: Sign the request to create the certificates

```
$ ./CA.pl -sign  
{Enter pass phrase and answer questions here}
```

Step 10: Remove the pass phrase (causes problems with stunnel)

```
$ openssl rsa -in newreq.pem -out newreq_nopasswd.pem
```

Step 11: Combine the request and the certificate into one file

```
$ cat new_reqnopasswd.pem newcert.pem > client.pem
```

Step 12: Copy the client's certificate to the correct location

```
$ cp client.pem /usr/local/etc/stunnel/certs
```

NOTE: It would be a good idea to rename the file to something other than client.pem. For example, *hostname-client.pem* would be a good choice. That would allow you to keep track of the client certificates much easier.

Step 13: Make the certificate directory the current working directory

```
$ cd /usr/local/etc/stunnel/certs
```

Step 14: Determine the hashed filename to use for the certificate

```
$ /etc/ssl/misc/c_hash client.pem
```

This command will print a line similar to this:

```
9248922f.0 => client.pem
```

NOTE: This is needed because of the way SSL works. When the server receives the client's certificate, it does not know exactly which file might hold the local copy of the certificate. To avoid having to open and test every certificate in the directory, the SSL routines perform a hashing algorithm that converts the certificate information into a apparently garbled file name. That garbled name is used to open and verify the certificate from the client.

Step 15: Create a link to the certificate using the hashed number printed.

```
$ ln -s client.pem 9248922f.0
```

NOTE: We could have also just renamed the certificate file, but after doing this many times it would be difficult to remember which hashed file name belongs to which client. By creating a link instead, you can find the correct file to delete very easily if you want to remove a client's privileges.

That should do it for the server side. Restart the stunnel program and let's move on to configuring the client, which is much simpler.

Client Side Configuration

Before attempting to setup the client side, you must copy the client's certificate from the server to the client computer. You can do this many ways, including using scp, FTP, e-mail, or even a floppy disk. Keep in mind that however you do this, your security is only guaranteed if nobody can copy or steal the client's certificate. Personally I like to use the scp command to transfer the certificate file. A floppy disk would also work, provided you reformat it after the transfer is complete.

On the client side, we must edit the **/usr/local/etc/stunnel/stunnel-*imap.conf*** file. The only line that should be changed is the *cert* entry. Make sure it points to the certificate file you just copied to the client.

```
# Client IMAP over SSL configuration file
```

```
cert = /usr/local/etc/stunnel/client.pem  
client = yes  
chroot = /usr/local/etc/stunnel  
pid = /stunnel-imap.pid  
setuid = nobody  
setgid = nobody
```

```
# Optional entries  
output = /var/log/stunnel-imap.log  
debug = 7
```

```
[imap2]  
accept = 143  
connect = wolf.bamafolks.com:993
```

After you restart the client's stunnel program, everything should be working. However unlike the prior example, not just everybody can use the tunnel. Attempts to connect to the tunnel without first copying the client's certificate to the server will be rejected. Security is pretty tight as long as your server does not get compromised.