;login:
THE MAGAZINE OF USENIX & SAGE
August 2002 volume 27 • number 4

inside:

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

securing FTP

by Gary Cohen

Gary is the CEO and co-founder of Glub Tech, Inc. and is concurrently employed by Adobe Systems. In the past he has worked for the San Diego Supercomputer Center and IBM.

gary@glub.com

In the summer of 1999, fellow classmate Brian Knight and I teamed up to take part in a senior project at the University of California, San Diego (UCSD). Both Brian and I worked as interns in the Computer Security department at the San Diego Supercomputer Center (SDSC) and were quite aware of some of the inherent security problems that lie within the FTP protocol. Under the leadership of Sid Karin and Tom Perrine, we worked with SDSC to build a safer FTP client. Just prior to starting this project, SDSC had made a move to disable all services that transmitted a user's password in the clear and to allow a user to transfer files only via scp (a file-transfer wrapper provided with SSH), FTP via an SSH tunnel, anonymous FTP, or Kerberos-authenticated FTP. Brian and I felt that these options each had their shortcomings.

For example, scp was widely available only on UNIX, and to create a user interface for it on other platforms made little sense since Secure Shell v2.0 was in beta and bundled an FTP emulator, sftp, with it. We looked at writing an FTP client that supported sftp, but the daemon was not very stable, which provided some hurdles, and we needed to finish the project within the 10-week quarter.

The second file-transfer method sanctioned by SDSC was FTP via an SSH tunnel. Unfortunately, this method requires an end-user to perform a fairly complex setup process. Even though SDSC provided documentation on how to configure a tunnel and connect to an FTP server securely, most end-users had difficulty completing all the steps or felt it was too inconvenient to do so.

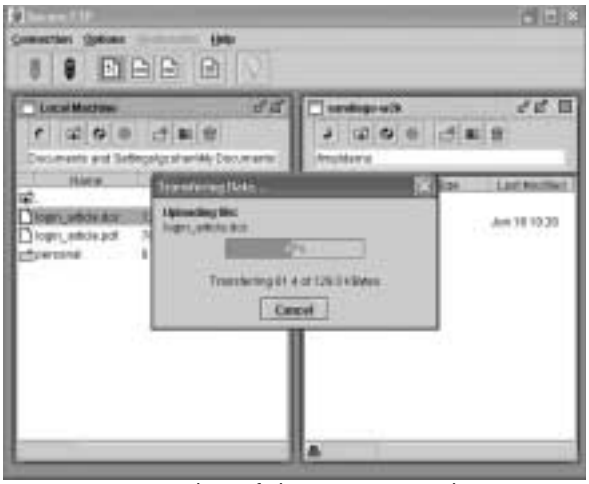
The third method, anonymous FTP, was probably the most popular way to transfer files between systems at SDSC, but the workflow was disjointed and could sometimes cause problems. By using anonymous FTP, a user would upload a file into a write-only directory (or dropbox). Once the file was uploaded, the user would have to log in to a shell, take ownership of the file, and move the file where it needed to go. In order to download the same file, the file would have to be moved to a separate, read-only directory before it could be retrieved by FTP. To say the least, the process was not very efficient.

The fourth method was the only cross-platform option: use FTP with Kerberos authentication. (SDSC also supplied the means to FTP via S/Key, but the number of users for this service was even smaller than the number of those who used FTP with Kerberos.) However, most of the users at SDSC did not use Kerberos, and the ones who did were mostly UNIX folks who preferred the secure file transfer capabilities of scp. Additionally, there were only a few clients that supported FTP via Kerberos.

Weighing our options, Brian and I felt implementing a secure client that did true FTP seemed like a smart decision. FTP had been proven to work since its inception, and people were already familiar with the client interface. But we were debating on how to secure the control channel. Prior to this project I had used an SSL wrapper to secure POP and IMAP for SDSC's mail system. It was fairly easy to set up on the server side (although we encountered a few issues using certificates generated by the Netscape Certificate Manager), and it seemed to work well with the available SSL-enabled mail clients. Keying off that success, we decided SSL looked like a viable security solution

for FTP as well. Our goal was to write an SSL-enabled FTP client that could communicate with an ordinary FTP server through an SSL wrapper.

Brian and I started planning the project (only as well as a couple of college seniors could). After reviewing the design with our advisors, we began writing our Secure FTP client. To ensure our program had the greatest reach across many platforms, we wrote it in Java as both applet and application. I took the role of designing and building the user interface while Brian implemented the networking. Most of the project went smoothly until we reached the phase of adding the SSL layer. Rather than reinvent the wheel, we looked for a Java library that would provide SSL functionality. There were a few to choose from but most were expensive (or at least it seemed that way to us poor college students). Fortunately, a couple of weeks before the project was due, Sun Microsystems released a reference implementation of SSL called the Java Secure Socket Extension (JSSE). The documentation was limited, and the only form of a demo was via Sun's HotJava Web browser. But we were able to get the library to behave as expected.



A screenshot of the Secure FTP client

We had planned to release the source code for the client once we were finished, but with this being our first large Java program, and with a deadline of 10 weeks, the finished code was anything but pretty. Both of us quickly agreed that nobody should ever see this code. It was embarrassing, but it worked as designed. We had written a client that could connect to an SSL wrapper that acted as a proxy to an existing FTP server.

Using our client with an SSL-wrapped FTP server provided a means to encrypt the control channel (which protects the username and password, the main motivation behind the project), but it didn't allow for encryption of the data channel. Although there were SSL wrappers available on UNIX, there wasn't a wrapper for Windows. Shortly after releasing the first revision of the client (aptly named Secure FTP v1.0), we started looking at writing a complementary wrapper in Java. At first this wrapper would only address the same issues as the UNIX-based ones (encryption of the control channel) but with multiplatform support. Furthermore, we wanted to simplify the wrapper configuration and the generation of certificates. We would address data encryption later. Secure FTP Wrapper v1.0 was released to the public in February 2001.

With the initial releases of the client and wrapper under our belts, we decided to look at data encryption, starting with the wrapper first. Since FTP uses two channels, one for the commands and one for the data, using a wrapper to encrypt both is nontrivial. Essentially, we would need to write a wrapper that understood the FTP protocol to handle data encryption. So that's what we did – we wrote a smart wrapper.

Version 1.0 of our wrapper acted as a transparent proxy, or "SSL translator," between an SSL-savvy client and a non-SSL FTP server. We got a secure connection from the client and forwarded the plaintext "conversation" to the server. When the user issued a command that required a data transfer, we just passed on the information without intervening. For the most part this worked fine. However, this caused problems for some servers that saw this behavior as a possible port theft. Port theft occurs when an FTP client connects to the server's control channel from a certain IP address but initiates a data transfer from a different one. The problem with our setup was that the FTP server saw the wrapper as the client when handling commands, but when a data transfer was started, the data was being sent to the true client (instead of the wrapper). This

There is no good reason to transmit a password in the clear.

conflict led the server to think the data was being stolen and, in some cases, the FTP server denied the request.

When we started working on the next version of the wrapper, the added support for data encryption had the positive side effect of removing any possible port theft. Instead of the wrapper just passing commands onto the server, it now acted as both a mini-client and mini-server.

Because of the nature of file transfers in FTP, an FTP client can act as both client and server. If the client sends the command PASV, it is requesting the server to open a listening socket to deal with the data transfer; this is known as a passive transfer. On the other hand, if the client sends the command PORT, it is requesting the server to connect to a socket that is listening on the client machine. Because of this dual nature, our wrapper had to deal with both possibilities.

To do this, instead of merely passing on commands to the server from the client, we now checked them against a list of known FTP commands. If we saw a PASV command issued by the client, our wrapper would still pass on that request to the server. But instead of sending the server's response back to the client as is, we create a new server socket, "duct" our new socket with the server's old socket, and rewrite the response to reference the new socket. Depending on the type of data connection, this new socket may be SSL enabled. Handling a PORT connection is similar to the PASV workflow, except the roles are reversed; we create a new server socket, have the server connect to our new socket, create another socket which we use to connect to the client, and "duct" our two sockets together. It sounds complicated (which it is), but this is why we decided to tackle data encryption at a later date.

Version 2.0 of our wrapper worked, but after we had originally started on our Secure FTP crusade, the official spec for FTP security extensions changed. Originally there were two supported mechanisms to handle FTP over SSL – explicit and implicit – but version 8 of the draft inexplicably dropped support for the implicit options. An explicit connection occurs when an FTP client connects to the standard FTP port (port 21) and, to enable security, it issues the command AUTH SSL. If the server supports this command, it would convert the control channel from an insecure to an SSL-enabled, secure channel. The other option, implicit, occurs when the client connects to the IANA-specified port for ftps (port 990). This port already is SSL enabled much the way https is enabled on port 443; when connecting to a secure Web server, an AUTH command is not required since the connection is secure to begin with.

It remains a mystery to us what problem the standards makers were trying to solve by dropping the implicit option. We tried to find a case where the control channel should allow user information to be transmitted in the clear. We can understand why one would not want to force encryption on the data channel due to speed issues, but there is no good reason to transmit a password in the clear.

While we do not agree with this change, we decided to handle it nonetheless by adding support for an explicit connection (Secure FTP Wrapper v2.5). However (due to our stubbornness), we have kept the implicit connection on by default. Additionally, we added a flag to our configuration that makes a secure control (and data) connection a requirement. When an explicit connection is made, and the requirement of a secure control connection is set, we will not allow the user to send a password without first sending the AUTH command.

Unfortunately, adding support for explicit SSL complicates the wrapper concept. Having a wrapper listen on the standard FTP control port (port 21) can cause configuration issues since the existing FTP server may already be listening on that port. Due to this obstacle, we do not enable the explicit connection if the wrapper IP address and port are the same as those of the destination server. There are a couple of solutions to make an explicit connection possible. One option would be for the server to listen on a different port with the same IP address as the wrapper. Another option would be to bind the server to a different address (such as localhost).

Separating the SSL support from the server might add complexity to a system administrator's job, but we think the advantages outweigh the disadvantages. The major advantage is you can continue to use your legacy FTP server. The last thing a busy administrator wants to do is upgrade an FTP server and make sure the users can still get their work done. In addition, our wrapper can be configured to listen on a border router (straddling the DMZ) and allow a secure connection from the Internet into a server that exists in your intranet. This kind of flexibility cannot be done if the encryption is taking place in the FTP daemon.

Now that there was a wrapper that supported encryption on both channels, we added the much-requested support for data encryption in the client (Secure FTP v1.6).

What's next? We are planning on externalizing our FTP via SSL implementation into a Java bean so others can incorporate it into their own Java programs. We also have some ideas to make our wrapper even smarter, but for now we'll keep that secret.

For more information on Secure FTP, or to download the client and wrapper, we invite you to take a look at the Secure FTP Web site (*http://secureftp.glub.com*) and the Secure FTP Wrapper Web site (*http://wrapper.glub.com*).

This paper was written with input from Brian Knight, CTO of Glub Tech, Inc.

We are planning on externalizing our FTP via SSL implementation into a Java bean so others can incorporate it into their own Java programs.