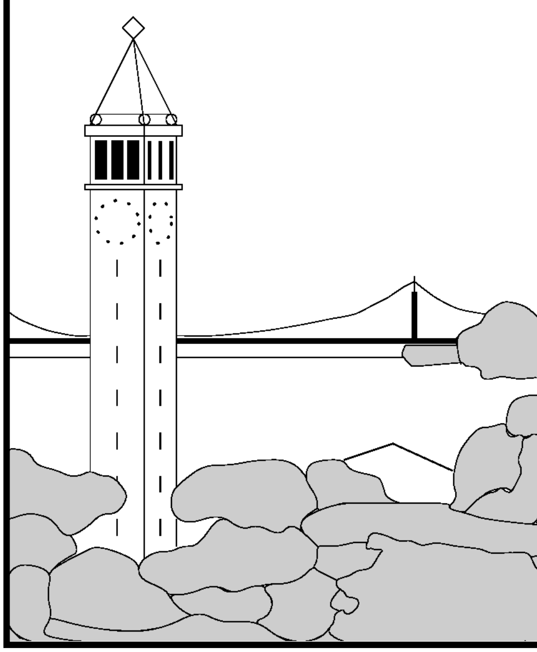
SafeTP: Transparently Securing FTP Network Services

Dan Bonachea and Scott McPeak
CS Division, EECS Department
University of California, Berkeley
{bonachea,smcpeak}@cs.berkeley.edu



Report No. UCB/CSD-01-1152

February 2001

Computer Science Division (EECS)
University of California
Berkeley, California 94720

SafeTP: Transparently Securing FTP Network Services

Dan Bonachea and Scott McPeak

Computer Science Department
UC Berkeley
Berkeley, CA 94720

U.C. Berkeley Tech Report ID: UCB/CSD-01-1152

Abstract

One of the most challenging practical aspects of providing end-to-end network security for legacy client-server protocols such as non-anonymous FTP (File Transfer Protocol) is convincing end users to actually use the secure alternatives, rather than abandoning them in favor of simpler, more familiar, or more fully featured insecure clients. A number of secure alternatives to the FTP protocol have been developed, but thus far have met with only limited success – we feel this is primarily due to the fact that these solutions almost universally require the end user to learn a new, unfamiliar client interface or tweak complicated settings in order to make the security work. The average end user is interested in maintaining the security of their account, but is unwilling to invest a significant effort to setup a complicated system or the time to learn a whole new interface.

SafeTP is a unique new FTP security system that strikes at the heart of this problem by providing completely transparent FTP security for users of Microsoft Windows[1]. SafeTP operates by installing a transparent proxy in the Windows networking stack which detects outgoing FTP connections from any legacy (insecure) Windows FTP client, and silently secures them using modern cryptographic techniques (the server must also support SafeTP in order for a secure connection to be successfully established). SafeTP is 100% compatible with existing (insecure) FTP servers, and will operate in an insecure mode if the server does not yet support the SafeTP protocol. One key feature of the SafeTP client proxy is that it was designed to be completely transparent to the client FTP application. This way, users can reap the benefits of FTP security, while continuing to use their existing FTP software.

Since its recent release on the internet, SafeTP has become extremely popular and is rapidly gaining acceptance in a diverse user community that includes numerous corporations, educational institutions and private users. In this paper, we describe the design of SafeTP and our experiences in implementing and maintaining this successful system. We discuss various challenges encountered in designing a fully transparent and interoperable security layer, and the solutions we implemented. We also describe various aspects of the hybrid public-key and shared-key cryptosystem used to provide confidentiality, integrity, and authenticity for FTP sessions.

1. Introduction: FTP is Insecure

This section presents some problems with FTP as it exists today, as motivation for the solution.

1.1 FTP Compromises Passwords

File Transfer Protocol (FTP) [RFC 959], is commonly used to transfer files from one system to another. It is convenient in a wide range of circumstances because it does not require any initial mutual trust relationship, as opposed to, say, NFS [RFC 1094].

However, this flexibility comes at a price: the user must enter a username and password at the start of every session. What's more, this information is sent over the network in the clear.

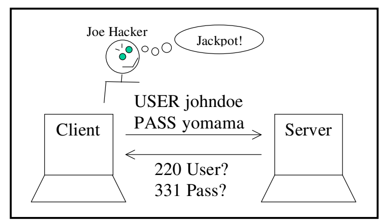


Figure 1 : Joe Hacker

These characteristics make FTP particularly vulnerable to an eavesdropping attack, such as packet sniffing. Anybody on the network between the user and the server, with sufficient access to put a network card in promiscuous mode, can easily get your password. As a result, insecure FTP has been recognized as one of the largest remaining security holes in many server systems.

1.2 Data is Sent in the Clear

A problem of secondary importance to the compromise of one's password, is the privacy, integrity, and authenticity of the FTP data itself.

With unprotected FTP, the data is sent in the clear, just like the password. Additionally, unprotected FTP is particularly vulnerable to unintended corruption of data, because it signals end-of-file by closing the TCP connection.[2] Such a closure need not be the result of a hack; any network outage that causes connection closure (e.g. a modem that hangs up) in the middle of a download will have the same effect.[3]

[1] The transparent client runs on MS Windows 9x/ME/NT/2000. A non-transparent, command-line SafeTP client is also available for UNIX. The SafeTP server proxy is available on UNIX and Windows NT/2000. All are freely available for download on the web (see [BM98] for details).

[2] FTP uses one connection for all control information during the session. However, it uses a separate connection for each data transfer, including directory listings.

[3] Most FTP clients protect against losing part of a file by waiting for the 226 (file transfer successful) reply from the

Finally, after the user issues a PORT[4] command, anyone can potentially connect to that port and send data, which the FTP client will happily write to disk. This could be used to, for example, substitute a Trojan horse for a downloaded program.

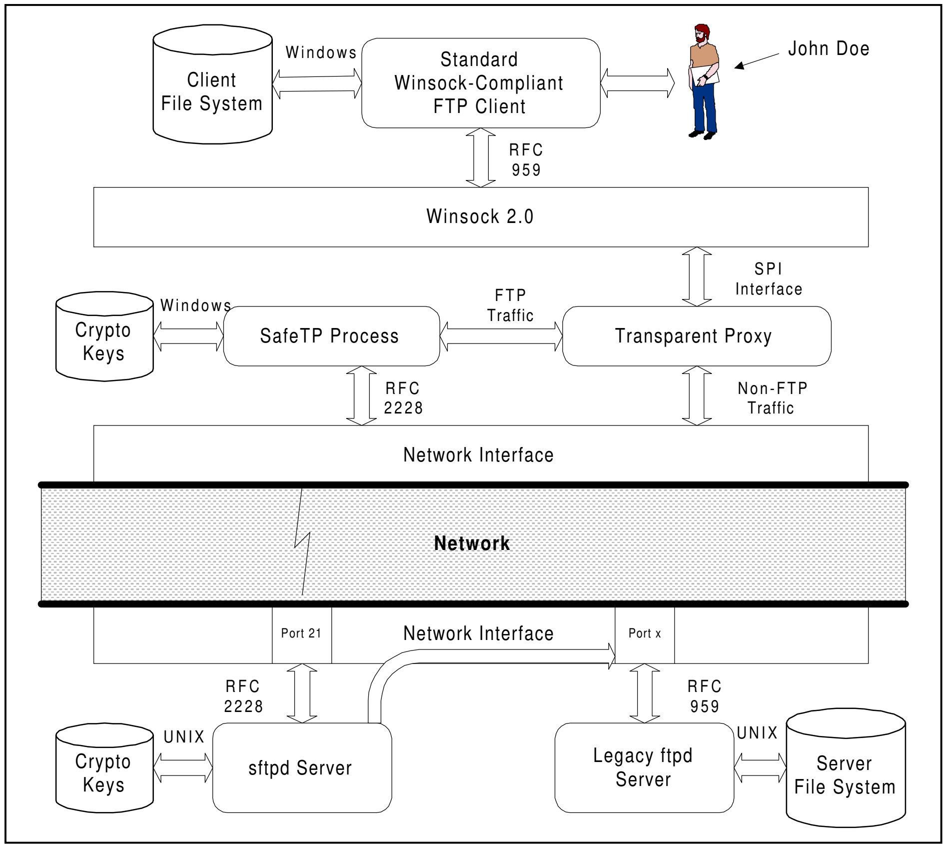


Figure 2: High Level System Architecture of SafeTP

1.3 Existing Solutions are Non-ideal

Of course, there currently exist encrypted data transfer agents. Some, such as HTTPS do not use FTP at all. Others, such as FileDrive [Diff98], carry on a normal FTP session via a secure sockets implementation such as SSL [Nets98]. Still others, such as Kerberized FTP [SNS88], require a special client application which can handle Kerberos security tickets and negotiate secure sessions. The basic problem with these alternatives is the lack of support for interoperability with existing FTP clients and servers, and the lack of transparency for users.

On the client side, alternative solutions require learning and using new client software. Inexperienced users are inconvenienced by having to learn a different interface, and experienced users are often frustrated at the lack of features that they so valued in their favorite client.

On the server side, installing a new daemon is yet another potential avenue for whole-system compromise. New implementations of server daemons must typically run as root for part or all of their running time, which is by itself a security risk. Further, the correct installation and support of a new system is a drain on limited resources.

2. The Solution: SafeTP

In this section we outline the high-level design of the SafeTP system, which had three primary goals: security, transparency and interoperability with existing FTP services.

Security

The new system *must* protect users' passwords.

We also wanted to protect users' data, though this should be optional if the performance impact is significant.

Transparency

The new system should appear to end users just like the old. Users should not have to switch to new client software, or connect to a different server (or different port on the same server), etc. Ideally, end users need not even be aware that their connection is being secured. Finally, system administrators should be able to continue using their current version of ftpd,[5] which is presumably tried and true.

RFC 959 Interoperability

SafeTP users should be able to connect to existing (insecure) FTP servers without telling the software in advance. They should be able to seamlessly and simultaneously connect to both secure and insecure servers.

Similarly, SafeTP servers must be able to support insecure client software. It is clear that any attempt to secure a widely-used service must offer a smooth transition path, and support for existing client software is absolutely essential for this. However, the server should also allow administrators to enforce connection security requirements as they see fit.

Our solution is to use transparent proxies on both the client and the server (see Figure 2). That is, we interpose an additional layer between the legacy software and the network, but hide the presence of that layer from both.

---

server, before they tell the user the transfer succeeded. But since the 226 usually precedes the actual end of file by a few kilobytes, the ends of files are still vulnerable (i.e. the control and data channels are not synchronized).

[4] FTP file transfers are preceded by a PORT command, which tells the server to which port on the client machine it should connect. The client then sends, e.g., RETR (retrieve file) or LIST (directory listing), which causes the server to actually begin the transfer.

[5] ftpd is the name of the normal UNIX FTP daemon.

The proxies communicate via a secure protocol. The framework for the protocol is specified by RFC 2228 [RFC 2228], and the details are specified by the SafeTP protocol [MB98]. The protocol's full name, as used during the negotiation, is "X-SafeTP1."[6] For brevity, we refer to it as X-SafeTP1 from here forward.

Each proxy allows the legacy software do its job normally. The server (typically) does not *care*, and the client can not *know*, that it is talking to a proxy. The SafeTP client proxy implementation is very careful to hide itself from the client, because it must interoperate with a wide variety of client software. The SafeTP server proxy can afford to be less transparent, because the variety of server software is less diverse and the administrator configuring it is assumed to be somewhat knowledgeable. The proxies encapsulate the communications in the secure protocol.

The proxies also accept the responsibility of distinguishing secure RFC 2228 peers from insecure RFC 959 peers. This determination is made before the legacy software gets involved, so the proxy always know how to maintain the illusion. When operating in RFC 959 compatibility mode, both proxies simply forward control traffic from one peer to the other, with minimal further examination (just enough to maintain the illusion).

The next three sections examine the goals of transparency, interoperability, and security in more detail. We then summarize some of the policy decisions in SafeTP. Finally, we present a performance evaluation and our conclusions.

3. Transparency

3.1 Completely Transparent Client Proxy

One key feature of the SafeTP client proxy is that it was designed to be completely transparent to the client FTP application. The purpose was to allow users to reap the benefits of FTP security, while continuing to use their existing FTP software. This transparency was accomplished using the Winsock 2.0 SPI interface [Micr97], which allows software "service providers" (such as our client proxy) to register with the OS to intercept socket calls (e.g. socket(), connect(), recv(), send(), etc.) and replace or augment the default behavior with additional processing. Winsock 2.0 comes with Windows 98/2000 and NT 4.0, and can be quickly and easily installed on Windows 95.

The client proxy implementation was divided into two modules: a call-interception DLL layer which implements the SPI interface and a proxy application that implements all the RFC 2228 and X-SafeTP1 semantics. This separation was made primarily for transparency reasons - the SPI interface implies the service provider shares an address space, TCP stack and other per-process resources with the Winsock client application (i.e. the FTP client software). This makes it very difficult for the proxy code to invisibly carry on its network activities without affecting the Winsock-TCP state of the FTP client. By isolating the proxy code in a separate process, we've made it virtually impossible for Winsock clients to detect the presence of the call-interception layer. Furthermore, the separation optimizes network startup time - the layer code is very lightweight, but the proxy code takes about 2 seconds to load and startup. By placing the proxy code in a separate, multithreaded process, a single instance can persist across multiple FTP sessions, handling all the FTP traffic for the system while amortizing a single startup cost.

Because the layer has no way to determine *a priori* whether a particular network application will open an FTP connection, it must run beneath all network applications and passively monitor their connection behavior, intervening at the correct moment if an FTP connection is initiated and redirecting this connection to the proxy process for handling. Once redirected, the layer takes steps to maintain the illusion that the Winsock application is connected directly to the remote server, such as intercepting the getpeername() function and substituting the server's address for the proxy's.

Table 1 lists the socket calls that are monitored by the layer, and gives the overhead it imposes on FTP (with data channel protection) and non-FTP sockets for those calls. With the exception of the first socket() call made by an application (which causes the layer to be loaded by the OS) the overhead imposed on non-FTP network applications is too small to be measured using the millisecond-granularity Windows clock. When data channel protection is enabled, the accept() function has a 2 ms overhead because the incoming data connection is being routed through the client proxy for decryption. The getsockname() overhead is an artifact of the separation between layer and proxy, but is called relatively infrequently by FTP clients so it makes little difference.

3.2 Mostly Transparent Server Proxy

The SafeTP server proxy (sftpd) sits between the insecure ftpd and the client proxy. The client proxy connects to sftpd at the standard, well-known FTP port (21). sftpd then connects to ftpd, which has been configured to listen on a different port.

During the initial negotiation, sftpd exchanges encryption information with the client proxy. Once negotiation is complete, sftpd proceeds to forward traffic back and forth, encrypting and decrypting as necessary.

While the connection between sftpd and ftpd uses the insecure RFC 959 protocol, this communication is assumed secure because it never goes over the network; in this case, network sockets essentially behave as an expensive form of interprocess communication.

Installation of sftpd on a Unix machine is straightforward and fully automated with installation scripts. First, /etc/services is modified to direct ftpd to listen on a new TCP port other than

[6] RFC 2228 specifies that names be registered with the IANA (http://www.iana.org), and that protocols not registered must begin with "X". IANA registration of X-SafeTP is still pending.

21. An entry for sftpd is added to /etc/services, at port 21. Finally, an sftpd entry is added to /etc/inetd.conf, which will cause inetd to spawn sftpd when an incoming connection is detected on port 21.

Since inetd, and not sftpd, listens to port 21, sftpd **need not run as root**. Further, since it forwards FTP commands to ftpd, which performs the bulk of the "real" file work on user's files, sftpd doesn't even need to assume the user's identity. The only file access sftpd needs is to its keys. Therefore, the ideal configuration creates a new user (called, e.g., safetp), who may access only his home directory, where the keys are stored, readable only by the safetp user. We see this as an important feature because it limits the potential for damage in the event the sftpd process is somehow compromised.

3.3 Transparent Proxies in General

We feel transparent proxies are a general technique that can be used to add security to existing insecure protocols and software. It applies on the client side whenever the proxy can intercept transport layer calls, and on the server side whenever the proxy can intercept the inbound connections.

Under the above conditions, even a fairly primitive set of proxies can secure a single TCP connection. However, multi-connection protocols, such as FTP, that imply a trust relationship across related parallel connections must be interpreted at both ends by both proxies, to detect imminent use of another connection. This requires the proxies to have some knowledge of the semantics of the insecure protocol. In SafeTP, for example, we interpret PORT and RETR (among others) to secure each data connection. The asynchronous nature of the parallel connections also implies an additional level of engineering complexity required to provide a robust forwarding layer that correctly handles all possible timings.

Interoperability also requires protocol knowledge. SafeTP takes advantage of FTP's request / reply model to insert an extra step into the early stages of negotiation. A similar approach for a protocol such as telnet could insert this request into the terminal negotiation sequence.

4. Interoperability

This section examines the issues associated with providing interoperability with the RFC 959 insecure FTP standard.

4.1 RFC 2228

RFC 2228 is an Internet standards track protocol, written by M. Horowitz and S. Lunt, and published as a Request For Comments (RFC) in October 1997. The standard proposes extensions to RFC 959, which defines FTP as it is widely deployed today.

RFC 2228 specifies a framework, not a complete protocol. It defines the syntax for some new FTP commands and describe what they generally mean in terms of security, but not how that security is achieved. It is left up to the implementer to define what RFC 2228 calls a "security mechanism," which must specify both the key exchange protocol (if there is one) and the particular algorithms to use.

RFC 2228 defines eight new commands, which we summarize here.

AUTH
The AUTH command requests that the server use a particular named security mechanism. The server must agree to a mechanism proposed by the client, or reject them all.

ADAT
The ADAT command is one half of the key-exchange protocol framework. A negotiation sequence consists of one or more AUTH's and zero or more ADAT's.

PROT
The PROT command enables data channel protection, and specifies the level of protection desired.

PBSZ
The PBSZ command defines the maximum encryption block size for data channel protection.

CCC
CCC is Clear Command Channel. It is used to disable control channel protection, and is not recommended.

MIC, CONF, ENC
MIC, CONF, and ENC are three variants of a general encoded request. They specify different levels of security for control channel requests and replies. ENC is the most secure, providing both integrity and confidentiality. RFC 959 FTP commands are encapsulated inside MIC, CONF, and ENC commands.

RFC 2228 also defines several new reply codes. The most important are:

534: Security mechanism unknown
If the client tries to AUTH a mechanism that the server does not know, it responds with a 534 reply. The client is then free to try another mechanism, or give up.

234, 235, 334, 335: ADAT replies
These replies form the second half of the key-exchange protocol. The different codes indicate whether the negotiation is finished, among other things.

631, 632, 633: Protected replies
These are the counterpart replies to MIC, ENC, and CONF. Protected requests provoke protected replies, to maintain the security of both. RFC 959 FTP replies are encapsulated inside 631, 632, and 633.

4.2 Compatibility with RFC 959 Server

When the SafeTP client proxy connects to an FTP server, it issues an AUTH command. If the server responds with a 500 (unknown) or 502 (unimplemented) error code, the client deduces the server does not support SafeTP-secured connections, and reverts to (insecure) RFC 959 compatibility mode. The user can opt to receive a warning when this happens, to prevent inadvertently transmitting their login/password over an insecure session.

4.3 Compatibility with RFC 959 Client

When the SafeTP server proxy receives a connection, it relays the legacy server's initial 220 (hello) response, and waits for the first request. If the first request is an AUTH, it proceeds with RFC 2228 negotiation. If the first request is USER (username) or ACCT (account identifier), the server reverts to RFC 959 compatibility mode. The server proxy can optionally be configured to refuse insecure FTP connections to enforce the use of secure client connections.

5. Security

5.1 Cryptographic Algorithms

In this section we briefly describe the encryption algorithms used in the X-SafeTP1 security mechanism. It is intended for a reader with a basic, but not necessarily thorough, understanding of modern cryptography (i.e. that the reader knows the difference between public-key and shared-key encryption). For each algorithm, we describe who created it, what it does, and how we use it.

DSA: Authentication

Digital Signature Algorithm (DSA) is part of the Digital Signature Standard (DSS), proposed by the National Institute of Standards and Technology (NIST) [NIST94]. NIST has made the algorithm publicly available, royalty-free. It is a public-key signature and verification algorithm based on discrete logarithms. An important feature of this algorithm is that each signature uses a random number, k. If an attacker ever recovers k, or ever sees two messages signed with the same k, the private DSA key is compromised.

We use DSA for server authentication.

ElGamal: Public Key Encryption

ElGamal, invented by T. ElGamal, is a public-key encryption algorithm that like DSA is based on discrete logarithms [ElGa85]. It is publicly available, and not covered by any patents. Like DSA, it uses a random number, k, for each encryption. If k becomes known, the private key is compromised.

We use ElGamal to encrypt the master session key.

Triple-DES: Shared Key Encryption

Data Encryption Standard (DES) is a shared-key, block cipher, using 56-bit keys, standardized by the National Bureau of Standards (now the NIST) in 1976 [BGK76]. Triple-DES, a stronger version of DES, uses three 56-bit keys. A particular optimization makes a brute-force attack on Triple-DES comparable to a brute-force attack on a shared-key algorithm with 112 bits.

We use Triple-DES to encrypt the ftp commands and data.

SHA1: One-Way Hash Function

Secure Hash Algorithm (SHA) is part of the Secure Hash Standard (SHS), proposed by the National Institute of Standards and Technology (NIST) [NIST94]. It is a one-way hash function, similar to, but believed more secure than, MD5. SHA1 is a slight variation on the original SHA.

We use SHA1 to construct the challenges, the master session key, and the session keys themselves.

SHA1HMAC: Message Authentication

A Hash Message Authentication Code (HMAC) is a one-way hash function, with a key that is required to compute and to verify the digest. SHA1HMAC uses the SHA1 hash function, and does something slightly more complicated than hashing the key as well as the message, to produce the digest.

We use SHA1HMAC to protect the integrity of the ftp commands and data.

Base64: ASCII Encoding of Binary Data

Base64 is a simple encoding scheme defined by RFC 2228.[7] It encodes three bytes of 8-bit data as four bytes of 6-bit data. The intent is this 6-bit data will then survive the various mutations applied by proxies, firewalls, and the like.

We use Base64 in the way prescribed by RFC 2228. Specifically, we encode all encrypted requests and replies with Base64.

5.2 Key Management

Keys are Not Encrypted on Disk

Keys are stored on disk (or, for the client, in the Windows Registry) in unencrypted form. The rationale for this is different for client and server.

On the server, it would be very inconvenient if the system operator were required to type a password at the console to decrypt the keys. It would require manual intervention on every reboot. This is prohibitive.

On the client, it is conceivable that the user would type a password to decrypt keys. However, the client's key is simply an ElGamal encryption key (as opposed to the server's more-important DSA authentication key); its compromise affects just that user, rather than an entire user community. Furthermore, regeneration of a key is a cost paid by just that user. Therefore, we chose to rely on the physical security of the user's machine.

Branded keys

In our system, DSA public keys are accompanied by the server's name, signed with the corresponding DSA private key. For this purpose, the server name may be simply its host name, or an organization name (e.g., "UCB CS Dept."), or some combination. The point is the key is permanently associated with its creator's human-readable name. We call this association "branding."

[7] There are several other encodings in existence that also use this name.

5.3 Key Exchange Protocol

In this section we explain the X-SafeTP1 key exchange protocol. The rationale behind the design should become clear in section 5.6 where we demonstrate how it defends against various forms of attack.

Client: Connect

First, the client connects a socket to sftpd, at port 21.

Server: 220

inetd then accepts the connection, and spawns sftpd to handle it. sftpd connects to ftpd's port (as listed in /etc/services) and inetd then spawns ftpd to handle this connection. ftpd sends an initial 220 (hello) response, which sftpd forwards to the client.

Client: Mechanism Proposal

Upon receipt of the 220, the client sends "AUTH X-SafeTP1".

Server: DSA Public Key

The server, seeing the AUTH command, recognizes that the client understands the RFC 2228 protocol. Recognizing the named security mechanism as well, it begins by sending, in a 334 (first ADAT) reply, its DSA public key. As mentioned above, this public key is branded with the server's name.

Client: ElGamal Public Key

The client tries to match the server's DSA public key with keys it has already seen. This is a crucial moment for the client in terms of policy; see Section 6 (Policy) for more detail. For now, we will assume that the client is happy with the server's DSA key.

The client sends, in an ADAT request, its ElGamal public key, and a randomly chosen challenge string.

Server: Master Key

The server replies, in a 335 (middle ADAT), with its own challenge string, the client's challenge string, its IP address, and a randomly chosen master key. The master key is encrypted with the client's ElGamal public key. The whole message is signed with the server's DSA private key.

At this point, the server computes the session keys from the master key.

Client: Verify Challenge

The client compares its original challenge to what the server sent. If they match, the server is authenticated. The client also compares the server's IP address to what was contained in the server's reply; these must also match. Finally, it decrypts the master key using its ElGamal private key.

At this point, the client computes the session keys from the master key.

The client then sends, as an ADAT request, the server's challenge string, encrypted with the client's Triple-DES key.

Server: Verify Challenge

Finally, the server decrypts and compares what the client sent against its original challenge string. If they match, the client is authenticated (in the sense that the client cannot be a hacker using a replay attack). The server completes the initial negotiation phase by sending a 235 (negotiation complete) reply.

Protect Data Channel (optional)

At this point, the client and server have a secure communications link. However, the client may want to protect the data channel as well. This requires two more RFC 2228 commands: PBSZ (protection buffer size) and PROT (data protection level). Both are straightforward.

USER and PASS are Secure

The client's commands, especially USER (username) and PASS (password), will now be encrypted, as will the server's replies.

5.4 Protocol Block Packaging

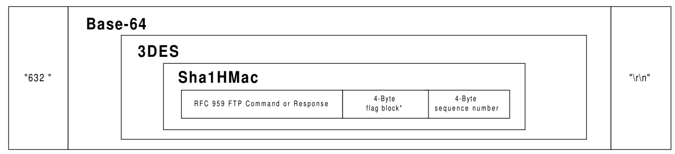


Figure 3: Control Block Packaging

Control Block

Every RFC 959 request and reply is tagged with a sequence number and a flags block, signed with SHA1HMAC, encrypted with Triple-DES, and encoded with Base64. Requests are preceded by "ENC ", a new (for RFC 2228) FTP command that means the request is encrypted. Replies are preceded by "632 ", a new (for RFC 2228) reply code that means the reply is encrypted.

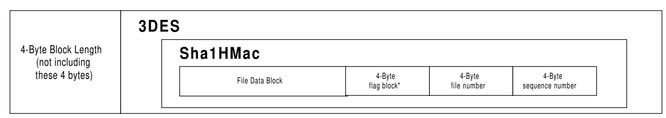


Figure 4: Data Block Packaging

Data Block

When data channel protection is on, the data is divided into blocks. The size of the block is, in practical terms, between 4 kb and 128 kb. Each data block is tagged with a file and block sequence number and a flags block, signed with SHA1HMAC, and encrypted with Triple-DES. Each encrypted block is then preceded by a 32-bit block length.

The end-of-file is indicated by encoding a block of zero length. Both the server and client proxies stall the 226 (file transfer succeeded) reply until they have received the EOF block. This provision is not really security-related, but fixes the irritating problem of distinguishing a prematurely closed connection from the real end-of-file.

5.5 Random Number Generation

Strong (i.e. not predictable) random number generation is central to our security system. DSA, ElGamal, both challenges, and the key generation itself all require a source of cryptographically strong random numbers.

During installation on both the client and server, the user is asked to type some sentences. Entropy is extracted from the variations in the timing of the user's keystrokes and serves to bootstrap the key generation process.

However, as the system is used over time, entropy is gradually lost as the random number generator is used. So, our system continues to gather entropy whenever the opportunity presents itself. We sample a variety of sources, including mouse position, window contents, start and connect times, network latency, and disk access latency, among others. The entropy from these sources is accumulated in a "pool" of randomness, which is itself stirred by a one-way hash function.

5.6 Some Particular Attacks and Defenses

This section discusses some potential attacks and describes how SafeTP defends against them. It is not meant as an exhaustive list, simply an exposition of various aspects of the protocol.

Eavesdropping
A simple eavesdropping attack is defeated by the encryption of data. First, the master session key is encrypted with ElGamal, and after that everything is encrypted with a Triple-DES session key. Only by attacking the algorithms themselves, or the random generator (in the case of ElGamal), can an eavesdropper learn more than the approximate data sizes and conversation duration.

Modification
Attacks involving selective deletion, modification, or retransmission of bytes in transit are more complicated. We consider such an attack at the protocol negotiation level and at the data block level.

During protocol negotiation, from the client's point of view, the danger is submitting their username and password to a hacker. But this can only happen if the server can decrypt the username and password, which requires the master key. If the hacker supplied the master key, he would be unable to sign the server's 2nd ADAT, because it requires the server's DSA private key. If he doesn't, he cannot decrypt the master key, because it requires the client's ElGamal private key.

An alternative attack is to substitute data after the negotiation is complete. However, doing so requires a SHA1HMAC key *and* a Triple-DES key. Both require access to the master key, which can only be set by the holder of the DSA private key, and can only be read by the holder of the ElGamal private key.

Packet Replay
A possible replay attack is to repeat the last control request or data block. This is defeated by including sequence numbers in each request, reply, and data block.

File Replay
To prevent the attacker from replaying an entire file transfer (block sequence numbers reset to zero each time), each file transfer block includes a file sequence number.

Echo Replay
Another possible replay attack is to send the data just sent by, say, the client, right back *to* the client. This is prevented by Triple-DES and SHA1HMAC, since the client and server use different keys for each. As an additional layer of security, each request, reply, and data block contains a flags block, which specifies (1) whether the sender is the client or the server, and (2) whether this is a control or data block.

Session Replay
The final replay attack is a whole-session attack. If the hacker records an entire session, he can then replay it for either the client or the server. Both the client and the server protect themselves against this attack by generating random challenge strings. This forces the server to sign new data, and the client to encrypt new data, for every session.

Coerced RFC 959 Fallback
A possible attack is to swallow all of the AUTH's, forcing the client and server to drop down into RFC 959 compatibility mode. The first defense is for the client to refuse to drop down if it has a DSA public key for that host. The second (optional) defense is to disallow RFC 959 FTP connections altogether.

Denial of Service
Like all internet-based services, SafeTP is subject to denial of service attacks. SafeTP is no more or less vulnerable to this.

5.7 The Weakness: Trust First Time

The main weakness of our protocol, present also in SSH [SSH98], is that we trust the server we reach is actually the server we intended to reach the first time we connect. Assuming the first connection is not tampered with, the client has a branded DSA public key which is saved in a key database and used to validate future connections to that same IP address.

Because trust first time may be a questionable policy in security-critical settings, we provide a interface for managing the key database – users may add server DSA keys acquired through other secure means, thus preventing the vulnerability on the first connection to those servers. SafeTP can be configured to automatically install a default key database at client installation time.

A Domain Name Service (DNS) attack (where the client is redirected to a new IP address using a DNS spoof) can still succeed because the client proxy always operates at the (numeric) IP address level. However, the user will receive a trust-first-time message in this situation to indicate the attack. A possible defense is to infer domain names by some means.

Our first, and in some situations only, line of defense is the user. The client proxy does not add new keys without user approval, and can be configured to not replicate (see Section 6, Policy) keys without user approval. The brands that accompany the DSA public keys can substantially improve the quality of the information available to the user at the time a decision must be made.

In the final analysis, the X-SafeTP1 protocol as it stands is vulnerable because it tries to avoid the inconvenience of using a Certification Authority. As SafeTP is now reaching wider deployment, it may soon make sense to work towards a certification infrastructure capable of closing the trust-first-time vulnerability.

```
class Transform {
  virtual int maximumEncodedSize(int decodedSize) const=0;
  virtual int maximumDecodedSize(int encodedSize) const=0;
  virtual void encode(DataBlock &data)=0;
  virtual void decode(DataBlock &data)=0;
  };

class ControlSecurity : public Transform {
  virtual bool hasOutgoingAdat() const;
  virtual void getNextOutgoingAdat(DataBlock &block);
  virtual bool expectingIncomingAdat() const;
  virtual void incomingAdat(DataBlock &block);
  };
```

Figure 5: Security Interfaces

5.8 Drop-in Security Mechanisms

Figure 5 shows the two most important interfaces within our RFC 2228 framework. Any particular security mechanism, such as X-SafeTP1, will create an object that is a subclass of ControlSecurity.

Transform is a general data transformation object, with enough functionality to allocate all required memory before the first transformation begins (this reduced the number of data copies). ControlSecurity has methods to support key-exchange protocols in a protocol-independent way.

With the X-SafeTP1 object, the proxy client and server can engage in a well-formed ADAT negotiation. They do not need to know anything about DE3S, only that it implements ControlSecurity.

Similarly, if a new security mechanism is created, it can ignore RFC 2228 protocol syntax, and instead just implement ControlSecurity's methods.

5.9 Why Not SSL?

Given the existence of Netscape's Secure Sockets Layer (SSL) [Nets98], the question is obvious: Why not just use SSL? The benefits are clear: SSL is a defined standard, it is generally regarded as secure, and a free implementation, SSLeay [HY98], is available.

Even so, we did not use SSL. While there is not a simple reason why, we present some of the rationale. It is not our contention that SSL could not have been used, but rather that, for this project with these design goals, it was not the best solution.

RFC 959 Interoperability

One of the key design goals is to allow SafeTP clients to work with RFC 959 servers, and SafeTP servers to work with RFC 959 clients. Using RFC 2228, this is very easy, because we simply shift to using a different command set once both parties are seen to support it. With SSL, we would need to define an FTP command to signal such support, and upon affirmative reply, upgrade both sides of the connection to SSL. Upgrading such a connection after it is already established may be difficult or impossible with some implementations.

Socket Startup Cost

FTP uses a new data connection for every file *and* directory listing. SSL is not optimized for fast socket setup and teardown; it was designed to be used for HTTP, which uses connections very differently. SSL contains provisions for session reuse, which may satisfy this issue, but the matter remains unclear.

Extensibility

We wanted a system that has a clear path for extending its security provisions with a minimum of effort. Implementing X-SafeTP1 within the RFC 2228 framework required 3 person-days and 800 lines of C++.[8] SSL's extension mechanisms are more complicated and seem to be able to leverage less of the infrastructure.

Blocking Semantics

Because SSL operates at the transport layer instead of the application layer, the semantics of individual operations (such as blocking on a recv[9] call), are affected. Especially in sftpd, which runs under UNIX, we were wary of a system that might require non-portable solutions to implement the semantics we need. With RFC 2228, there is no such concern.

RFC 2228 Compatibility

RFC 2228 defines a draft standard for secure extensions to FTP. There may be other implementations of RFC 2228, either now or in the future. We would like to be interoperable with as many other secure FTP implementations as possible, and adhering to an existing RFC is a good way to promote that objective.

5.10 DIGT command

Because the security mechanism negotiation happens in the clear, a possible attack is to force the client and server to fall back on a weak mechanism, if more than one exists. To prevent this, we propose an extension to RFC 2228[10]: the DIGT request. This request causes the server to send a digest of all the requests and replies up to, but not including the DIGT request itself. The client then only proceeds if the server's digest matches his own.

If several security mechanisms are possible, for example an export[11] version and a non-export version, it is possible for an attacker to swallow all of the AUTH's, until the client tries the

[8] We used crypto++, a C++ cryptographic algorithm library [Dai98], and GNU's GMP multi-precision library [GMP00]

[9] recv is the Berkeley Sockets call to read from a network socket (connection).

[10] We've proposed and implemented several minor extensions to RFC 2228, which are documented in an internet draft [BM99].

[11] Until recently, the U.S. strictly limited the strength of exported encryption technology. This was, naturally enough, a sore point with the encryption community.

weaker protocol. This lets the attacker choose the protocol he can attack most easily.

The protection against this is the DIGT command. Once security negotiation is complete, the client asks the server for a digest of all the previous messages. If the attacker has modified any of the AUTH's, the digests computed by the server and the client will not match.

This defense only works if the attacker is not fast enough to actually break the mechanism by the time the DIGT is issued. This is not a problem in the current implementation because we do not currently have more than one protocol.

6. Policy summary

6.1 Key length

3DES: (effectively) 112 bits, length fixed by algorithm

DSA: 1024 bits

ElGamal - user-definable: 768, 1024 or 2048 bits. The system administrator may set a minimum ElGamal key length required for incoming FTP sessions.

6.2 Trust First Time

The general security policy for the acquisition of new server keys is trust first time - the first time a user connects to a particular server which supports the X-SafeTP1 protocol, he will see a message box prompting him to accept or refuse the new server DSA key. The key is then stored in that user's section of the Windows registry, where it will be checked on all future connects to that same server IP as part of the server authentication process.

Multihomed Servers

Some servers have multiple IP addresses which all access the same system services. The SafeTP software was designed to handle such servers with minimal user interaction necessary. When connecting to a particular IP address for the first time, the DSA key database is searched using the branded DSA key provided by the server during negotiation, and if a match is found, the client decides this is a multihomed host and silently accepts the new IP address. This is called key replication This does not open any security holes because the DSA key is the unit of trust in our system - if the party we're connected to can prove though challenge/response that they possess the private key for a trusted DSA key, then they pass the authentication test, regardless of the IP address we used to reach them. The IP address is merely a method for detecting key changes on servers to which we've previously connected.

Out-of-band Key Entry

For the ultra-paranoid, SafeTP allows the use of other, out-of-band methods for acquiring new keys. The SafeTP Manager user interface allows users to view their current keys, delete keys and add new keys.

6.3 Unknown FTP Commands

The sftpd server software recognizes all the FTP commands defined by RFC 959, which is the most widely accepted standard for the FTP protocol. In addition, sftpd also recognizes a subset of the "special-purpose" FTP commands defined by other RFC's (for example, the XPWD command). When data channel protection is inactive, sftpd allows all client requests to pass directly to the legacy ftpd server, regardless of whether it recognizes them or not (at which point ftpd will reply, possibly with an error). This is not a problem because once negotiation has completed, it is not possible for the client to issue any command that would cause the control channel to become insecure. However, when sftpd accepts data channel protection, it makes a commitment that all data transfers for the remainder of this FTP session will be secure. In order to ensure this, sftpd must recognize and take control of all data transfers - if it were to allow unrecognized commands to pass to ftpd, then it may be possible for these commands to initiate data transfers outside of sftpd's knowledge. For this reason, unrecognized client requests are denied by sftpd and not forwarded to ftpd when data channel protection is active.

6.4 RFC 959 Compatibility

For system administrators seeking extra security, we provide a setting to enable/disable the RFC 959 drop-down capabilities of sftpd. By configuring ftpd to accept only connections from the local machine, users will be forced to use secure ftp connections. We recommend 959 compatibility mode be enabled as part of a transition period, after which the administrator may switch to requiring all FTP connections to be secure.

6.5 No CCC

sftpd explicitly refuses the CCC command defined by RFC 2228, whose purpose is to remove control channel encryption after the USER/PASS commands have been sent. It was felt there was no reason to warrant implementing this potentially dangerous command, especially because the performance penalty incurred by the control channel encryption after negotiation is minimal (very low bandwidth).

6.6 Disallow 3rd Party Transfers

RFC 2228 explicitly disallows 3rd party (server-to-server) file transfers when data channel encryption is enabled.

6.7 Policy Recommendations

The sftpd user's home directory should be securely replicated to the local disk of every machine running sftpd, to avoid compromising the keys through NFS (may not be necessary when using a secure network file system). If sftpd is running on physically insecure workstations, each workstation should have its own DSA key.

The sftpd DSA keys should never be changed unless absolutely necessary (such as if a compromise is detected), because this will cause a security warning the next time previous users try to log in, and could eventually breed user apathy towards SafeTP security warnings.

The users should be provided with an out-of-band method for obtaining the server DSA public key - for example a read-only file in a public directory, or a page on a secure web site. Users

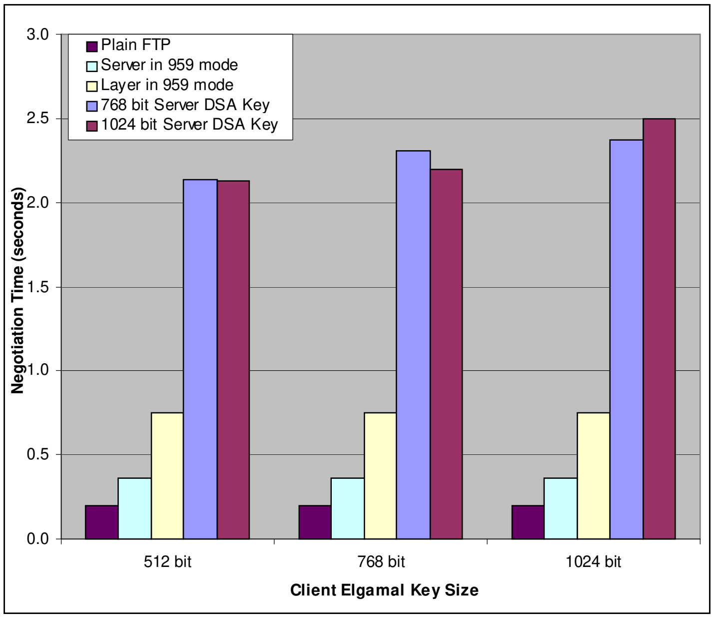


Figure 6: Control Channel Negotiation Time

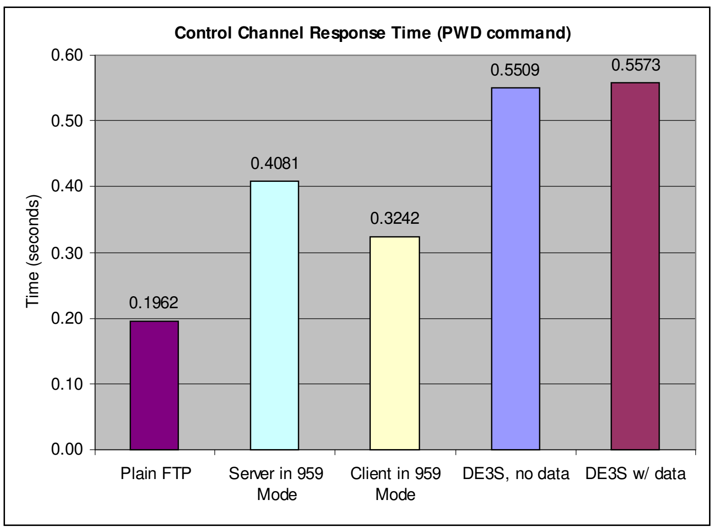


Figure 7: Control Channel Response Time

can cut and paste this key into the SafeTP Manager for future use in authentication.

7. Performance

As with most encryption and integrity systems, the security provided by SafeTP comes at a price of decreased performance. This section is intended to illustrate that the overheads imposed by SafeTP are not unreasonable, with the caveat that the current implementation is largely unoptimized, and the authors expect performance improvements in future revisions. All performance measurements were performed on a dedicated 10-MBit Ethernet LAN connecting two Pentiums. Performance measurements are given for the client and server proxies operating together in X-SafeTP1 mode, both with and without data protection enabled, as well as for the proxies operating independently in 959-compatibility mode, and a baseline measurement for regular FTP with no proxies (Note the figures use "DE3S" to mean X-SafeTP1).

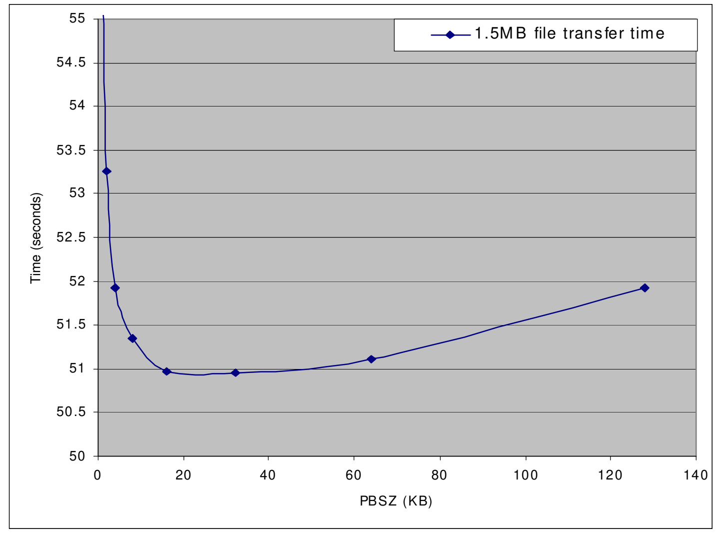


Figure 8: Transfer Time vs. PBSZ

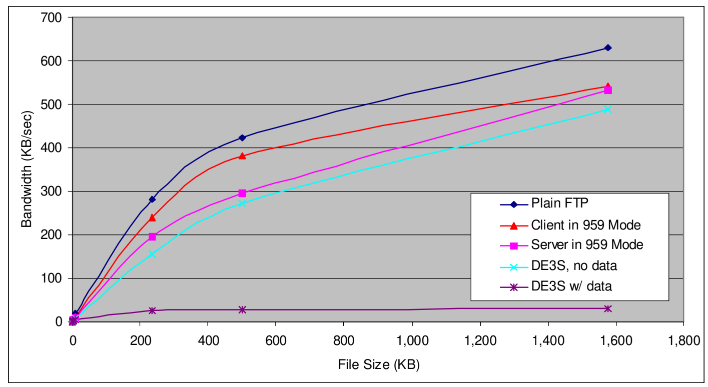


Figure 9: Data Connection Bandwidth

Figure 6 demonstrates the latencies of initial connection negotiation associated with the various FTP options, measured as the time from the FTP client issuing connect() to the time where it receives the server's 220 message. Our network demonstrated about a 2-second cost associated with the X-SafeTP1 negotiation, which appears to be mostly encryption overhead - delays associated with the 2 extra round-trip times will obviously vary with network conditions.

Figure 7 shows the response time latency associated with regular FTP commands (such as PWD) on the control channel. The additional latencies added by the client and server are due to encryption-decryption overhead (in the X-SafeTP1 cases) and extra trips up and down the network stack (address space crossings) in routing the command and reply.

Figure 8 illustrates a brief investigation into the optimal PBSZ (maximum data block size) for large encrypted data transfers. The competing factors causing performance to vary with block size are the increasing fraction of header bytes to data bytes (header overhead) with smaller block sizes, and the increasing fraction of the first and last block size to the entire file size, because the encryption of the first block and the decryption of the last block can't be overlapped with other computation or network communication. The optimal PBSZ for a 1.5MB file appears to be about 32 KB.

Figure 9 demonstrates the data connection bandwidth of the various configurations for different file sizes. These bandwidth measures were derived from time measurements that include the overhead associated with the control channel activity necessary to initiate the file transfer (i.e. PORT, RETR, and the server replies), which is why the proxy configurations without data encryption differ slightly from the ideal case which does not encrypt this control activity (these cases are otherwise identical - recall the data connection does not pass through either proxy when data protection is disabled). Clearly, the relative fraction of this difference diminishes as the control time is amortized over larger file sizes. The true difference comes in the encryption overheads imposed by data channel protection, which we unfortunately found to be rather large. This result argues for implementing different levels of data protection to take advantage of situations where a weaker protection guarantee (such as integrity without secrecy) is sufficient.

8. Conclusion

Transparent, protocol-aware proxies offer a good way to add security to existing protocols and software. They allow the security mechanism to remain largely orthogonal to the protocol being secured, let users continue to use tried and true software and permit an orderly transition period by interoperating with both old and new systems.

SafeTP is an example of such a system. We built a secure cryptosystem, implemented transparent proxy layers at both ends and leveraged RFC 2228 to interoperate with existing RFC 959 systems.

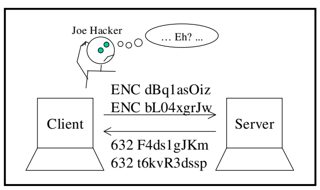


Figure 10 : Joe Hacker

The SafeTP system has been incredibly successful, and has been installed in many systems worldwide. The feedback from our users has been very positive, and seems to indicate this paradigm of transparent security is a valuable one. The SafeTP distribution and full documentation are freely available on the SafeTP web page [BM98].

9. Acknowledgments

We wish to thank David Wagner and Ian Goldberg for sage advice on cryptographic algorithms and protocol evaluation. All errors or omissions in either are, however, ours and not theirs. We also thank the countless users who've provided invaluable feedback concerning their experience with SafeTP.

References

[BGK76] D.K. Branstad, J. Gait, and S. Katzke. "Report on the Workshop on Cryptography in Support of Computer Security." NBSIR 77-1291, National Bureau of Standards, Sep 21-22, 1976, September 1977.

[BM98] Dan Bonachea and Scott McPeak. "SafeTP Webpage" http://safetp.cs.berkeley.edu/

[BM99] Dan Bonachea and Scott McPeak. "Protocol Negotiation Extensions to Secure FTP", Internet Draft, http://safetp.cs.berkeley.edu/draft-bonachea-sftp-00.txt

[ElGa85] T. ElGamal. "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 10-18

[Dai98] Wei Dai. "Crypto++ 2.3" http://www.eskimo.com/~weidai/cryptlib.html

[Diff98] Differential Inc. "FileDrive." http://www.downloader.com/filedrive_client.html

[GMP00] Gnu Multi-precision library http://www.swox.com/gmp/

[HY98] T.J. Hudson and E.A. Young. "SSLeay and SSL apps FAQ." http://www.psy.uq.oz.au/~Eftp/Crypto/

[MB98] Scott McPeak and Dan Bonachea. "X-SafeTP1 Protocol Specification", http://safetp.cs.berkeley.edu/protocol.txt

[Micr97] Microsoft, Inc. "Windows Sockets 2 API." ftp://ftp.microsoft.com/bussys/winsock/winsock2/wsapi22.doc

[Micr97] Microsoft, Inc. "Windows Sockets 2 Service Provider Interface (SPI)." ftp://ftp.microsoft.com/bussys/winsock/winsock2/wsspi22.doc

[Nets98] Netscape Communications Inc. "SSL 3.0 SPECIFICATION." http://home.netscape.com/eng/ssl3/

[NIST94] National Institute of Standards and Technology, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, Feb. 94.

[RFC 959] J. Postel, J. Reynolds. Request for Comments 959, "File Transfer Protocol (FTP)." October 1985.

[RFC 1094] Sun Microsystems, Inc. Request for Comments 1094, "NFS: Network File System Protocol Specification." March 1989.

[RFC 2228] M. Horowitz, S. Lunt. Request for Comments 2228, "FTP Security Extensions." October 1997.

[SSH98] SSH Communications Security, Inc. "SSH Protocols and Secure Shell." http://www.ssh.fi/sshprotocols2/index.html

[SNS88] J. Steiner, C. Neuman and J. Schiller "Kerberos: An Authentication Service for Open Network Systems", January, 1988.