CAT Working Group
<draft-ietf-cat-ftpkeasj-01.txt>
Updates: RFC 959
Internet-Draft
Expire in six months

Russell Housley (SPYRUS)
William A. Nace (NSA)
Peter Yee (SPYRUS)

January 1999

Encryption using KEA and SKIPJACK

Status of this Memo

Abstract

This document defines a method to encrypt a file transfer using the FTP specification RFC 959, 'FILE TRANSFER PROTOCOL (FTP)' (October 1985) and the work in progress document 'FTP Security Extensions' <draft-ietf-cat-ftpsec-09.txt>[1]. This method will use the Key Exchange Algorithm (KEA) to give mutual authentication and establish the data encryption keys. SKIPJACK is used to encrypt file data and the FTP command channel.

1.0 Introduction

The File Transfer Protocol (FTP) provides no protocol security except for a user authentication password which is transmitted in the clear. In addition, the protocol does not protect the file transfer session beyond the original authentication phase.

The Internet Engineering Task Force (IETF) Common Authentication

Technology (CAT) Working Group has proposed security extensions to FTP. These extensions allow the protocol to use more flexible security schemes, and in particular allows for various levels of protection for the FTP command and data connections. This document describes a profile for the FTP Security Extensions by which these mechanisms may be provisioned using the Key Exchange Algorithm (KEA) in conjunction with the SKIPJACK symmetric encryption algorithm.

The FTP Security Extensions are likely to become a standards track RFC in 1997. It provides:

* user authentication -- augmenting the normal password mechanism;

* server authentication -- normally done in conjunction with user authentication;

* session parameter negotiation -- in particular, encryption keys and attributes;

* command connection protection -- integrity, confidentiality, or both;

* data transfer protection -- same as for command connection protection.

In order to support the above security services, the two FTP entities negotiate a mechanism. This process is open-ended and completes when both entities agree on an acceptable mechanism or when the initiating party (always the client) is unable to suggest an agreeable mechanism. Once the entities agree upon a mechanism, they may commence authentication and/or parameter negotiation.

Authentication and parameter negotiation occur within an unbounded series of exchanges. At the completion of the exchanges, the entities will either be authenticated (unilateral or mutually), and may, additionally, be ready to protect FTP commands and data.

Following the exchanges, the entities negotiate the size of the buffers they will use in protecting the commands and data that follow. This process is accomplished in two steps: the client offers a suggested buffer size and the server may either refuse it, counter it, or accept it.

At this point, the entities may issue protected commands within the bounds of the parameters negotiated through the security exchanges. Protected commands are issued by applying the protection services required to the normal commands and Base64 encoding the results. The encoded results are sent as the data field within a ENC (integrity

and confidentiality) command. Base64 is an encoding for mapping binary data onto a textual character set that is able to pass through most 7-bit systems without loss. The server sends back responses in new result codes which allow the identical protections and Base64 encoding to be applied to the results. Protection of the data transfers can be specified via the PROT command which supports the same protections as those afforded the other FTP commands. PROT commands may be sent on a transfer-by-transfer basis, however, the session parameters may not be changed within a session.

2.0 Key Exchange Algorithm (KEA) Profile

This paper profiles KEA with SKIPJACK to achieve certain security services when used in conjunction with the FTP Security Extensions framework. FTP entities may use KEA to give mutual authentication and establish data encryption keys. We specify a simple token format and set of exchanges to deliver these services. Functions that may be performed by the Fortezza Crypto Card.

The reader should be familiar with the extensions in order to understand the protocol steps that follow. In the context of the FTP Security Extensions, we suggest the usage of KEA with SKIPJACK for authentication, integrity, and confidentiality.

A client may mutually authenticate with a server. What follows are the protocol steps necessary to perform KEA authentication under the FTP Security Extensions framework. Where failure modes are encountered, the return codes follow those specified in the Extensions. They are not enumerated in this document as they are invariant among the mechanisms used. The certificates are ASN.1 encoded.

The exchanges detailed below presume a working knowledge of the FTP Security Extensions. The notation for concatenation is " || ". Decryption of encrypted data and certification path validation is implicitly assumed, but is not shown.

```
------------------------------------------------------------------
  Client                              Server

  AUTH KEA-SKIPJACK              -->
                                      <-- 334 ADAT=Base64( Certb || Rb )
  ADAT Base64( Certa || Ra ||
    WMEK || IV || Encrypt(
    Label-Type || Label-Length ||
    Label-List || pad || ICV ) ) -->
                                      <-- 235 ADAT=Base64( IV )
------------------------------------------------------------------
```

Figure 1

The server and client certificates contain KEA public keys. The client and server use KEA to generate a shared SKIPJACK symmetric key, called the TEK. The client uses the random number generator to create a second SKIPJACK key, called the MEK. The MEK is wrapped in the TEK for transfer to the server. An initialization vector (IV) associated with the MEK is generated by the client and transferred to the server. A list of security labels that the client wants to use for this FTP session may be transferred to the server encrypted in the MEK. As shown in Figure 2, the security label data is formatted as a one octet type value, a four octet label length, the security label list, padding, followed by an eight octet integrity check value (ICV). Figure 3 lists the label types. If the label type is absent (value of zero length), then the label size must be zero.

In order to ensure that the length of the plain text is a multiple of the cryptographic block size, padding shall be performed as follows. The input to the SKIPJACK CBC encryption process shall be padded to a multiple of 8 octets. Let n be the length in octets of the input. Pad the input by appending 8 - (n mod 8) octets to the end of the message, each having the value 8 - (n mod 8), the number of octets being added. In hexadecimal, he possible pad strings are: 01, 0202, 030303, 04040404, 0505050505, 060606060606, 07070707070707, and 0808080808080808. All input is padded with 1 to 8 octets to produce a multiple of 8 octets in length. This pad technique is used whenever SKIPJACK CBC encryption is performed.

An ICV is calculated over the plaintext security label and padding. The ICV algorithm used is the 32-bit one's complement addition of each 32-bit block followed by 32 zero bits. This ICV technique is used in conjunction with SKIPJACK CBC encryption to provide data integrity.

Label Type	1 Octet
Label Length	4 octets
Label List	variable length
Pad	1 to 8 octets
ICV	8 octets

Figure 2

Label Type	Label Syntax	Reference
0	Absent	Not applicable
1	MSP	SDN.701[1]
2-255	Reserved for Future Use	To Be Determined

Figure 3

FTP command channel operations are now confidentiality protected. To provide integrity, the command sequence number, padding, and ICV are appended to each command prior to encryption.

Sequence integrity is provided by using a 16-bit sequence number which is incremented for each command. The sequence number is initialized with the least significant 16-bits of Ra. The server response will include the same sequence number as the client command.

An ICV is calculated over the individual commands (including the carriage return and line feed required to terminate commands), the sequence number, and pad.

```
  Client                              Server

  ENC Base64(Encrypt("PBSZ 65535"
      || SEQ || pad || ICV ))     -->
                                      <-- 632  Base64(Encrypt("200" ||
                                                 SEQ || pad || ICV))
  ENC Base64(Encrypt("USER yee"
        || SEQ || pad || ICV))    -->
                                      <-- 632  Base64(Encrypt("331" ||
                                                 SEQ || pad || ICV))
  ENC Base64(Encrypt("PASS
        fortezza" || SEQ ||
        pad || ICV))              -->
                                      <-- 631  Base64(Sign("230"))
```

Figure 4

After decryption, both parties verifying the integrity of the PBSZ commands by checking for the expected sequence number and correct ICV. The correct SKIPJACK key calculation, ICV checking, and the validation of the certificates containing the KEA public keys provides mutual identification and authentication.

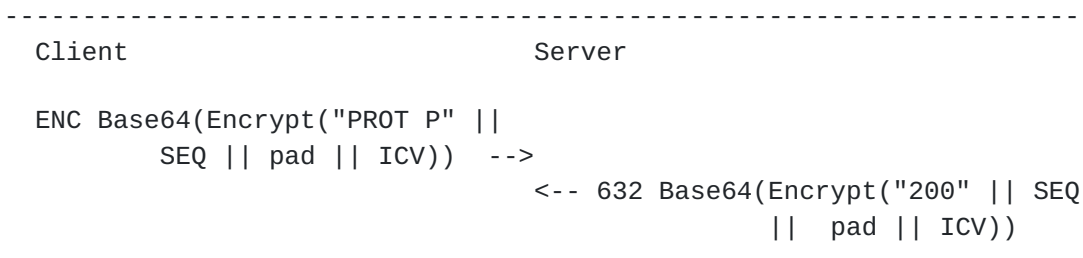

Figure 5

At this point, files may be sent or received with encryption and integrity services in use. If encryption is used, then the first buffer will contain the token followed by enough encrypted file octets to completely fill the buffer (unless the file is too short to fill the buffer). Subsequent buffers contain only encrypted file octets. All buffers are completely full except the final buffer.

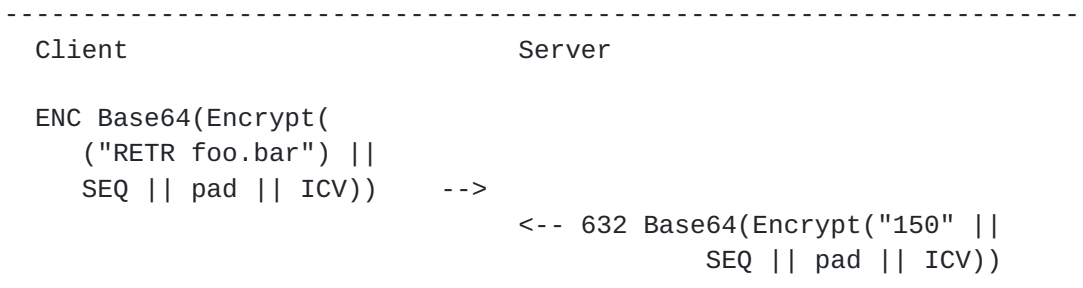

Figure 6

The next figure shows the header information and the file data.

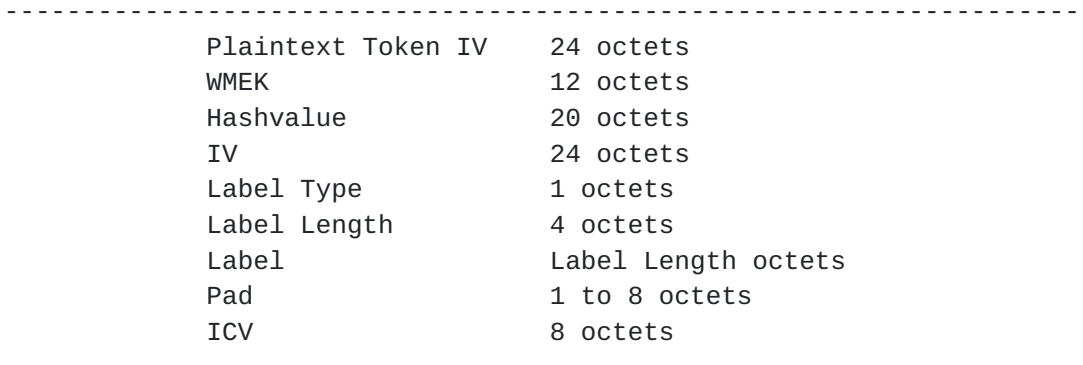

Figure 7

2.1 Pre-encrypted File Support

In order to support both on-the-fly encryption and pre-encrypted files, a token is defined for carrying a file encryption key (FEK). To prevent truncation and ensure file integrity, the token also

includes a hash computed on the complete file. The token also contains the security label associate with the file. This FEK is wrapped in the session TEK. The token is encrypted in the session TEK using SKIPJACK CBC mode. The token contains a 12 octet wrapped FEK, a 20 octet file hash, a 24 octet file IV, a 1 octet label type, a 4 octet label length, a variable length label value, a one to 8 octet pad, and an 8 octet ICV. The first 24 octets of the token are the plaintext IV used to encrypt the remainder of the token. The token requires its own encryption IV because it is transmitted across the data channel, not the command channel, and ordering between the channels cannot be guaranteed. Storage of precomputed keys and hashes for files in the file system is a local implementation matter; however, it is suggested that if a file is pre-encrypted, then the FEK be wrapped in a local storage key. When the file is needed, the FEK is unwrapped using the local storage key, and then rewrapped in the session TEK. Figure 8 shows the assembled token.

Plaintext Token IV	24 octets
Wrapped FEK	12 octets
Hashvalue	20 octets
IV	24 octets
Label Type	1 octet
Label Length	4 octets
Label	Label Length octets
Pad	1 to 8 octets
ICV	8 octets

Figure 8

3.0 Table of Key Terminology

In order to clarify the usage of various keys in this protocol, Figure 9 summarizes key types and their usage:

Key Type	Usage
TEK	Encryption of token at the beginning of each file, also wraps the MEK
MEK	Encryption of command channel
FEK	Encryption of the file itself (may be done out of scope of FTP)

Figure 9

4.0 Security Considerations

This entire memo is about security mechanisms. For KEA to provide the authentication and key management discussed, the implementation must protect the private key from disclosure. For SKIPJACK to provide the confidentiality discussed, the implementation must protect the shared symmetric keys from disclosure.

5.0 Acknowledgements

I would like to thank Todd Horting for insights gained during implementation of this specification.

6.0 References

[1] - M. Horowitz and S. J. Lunt. FTP Security Extensions. RFC 2228. October 1997.

[2] - Message Security Protocol 4.0 (MSP), Revision A. Secure Data Network System (SDNS) Specification, SDN.701, February 6, 1997.

7.0 Author's Address

Russell Housley
SPYRUS
381 Elden Street
Suite 1120
Herndon, VA 20170
USA

Phone: +1 703 707-0696
Email: housley@spyrus.com

DIRNSA
Attn: X22 (W. Nace)
9800 Savage Road
Fort Meade, MD 20755-6000
USA

Phone: +1 410 859-4464
Email: WANace@missi.ncsc.mil

Peter Yee
SPYRUS
5303 Betsy Ross Drive
Santa Clara, CA 95054
USA

Phone: +1 408 327-1901
Email: yee@spyrus.com