Network Working Group
Request for Comments: 4253
Category: Standards Track

T. Ylonen
SSH Communications Security Corp
C. Lonvick, Ed.
Cisco Systems, Inc.
January 2006

The Secure Shell (SSH) Transport Layer Protocol

Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2006).

Abstract

The Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network.

This document describes the SSH transport layer protocol, which typically runs on top of TCP/IP. The protocol can be used as a basis for a number of secure network services. It provides strong encryption, server authentication, and integrity protection. It may also provide compression.

Key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated.

This document also describes the Diffie-Hellman key exchange method and the minimal set of algorithms that are needed to implement the SSH transport layer protocol.

Table of Contents

1. Introduction 3
2. Contributors 3
3. Conventions Used in This Document 3
4. Connection Setup 4
   4.1. Use over TCP/IP 4
   4.2. Protocol Version Exchange 4
5. Compatibility With Old SSH Versions 5
   5.1. Old Client, New Server 6
   5.2. New Client, Old Server 6
   5.3. Packet Size and Overhead 6
6. Binary Packet Protocol 7
   6.1. Maximum Packet Length 8
   6.2. Compression 8
   6.3. Encryption 9
   6.4. Data Integrity 12
   6.5. Key Exchange Methods 13
   6.6. Public Key Algorithms 13
7. Key Exchange 15
   7.1. Algorithm Negotiation 17
   7.2. Output from Key Exchange 20
   7.3. Taking Keys Into Use 21
8. Diffie-Hellman Key Exchange 21
   8.1. diffie-hellman-group1-sha1 23
   8.2. diffie-hellman-group14-sha1 23
9. Key Re-Exchange 23
10. Service Request 24
11. Additional Messages 25
   11.1. Disconnection Message 25
   11.2. Ignored Data Message 26
   11.3. Debug Message 26
   11.4. Reserved Messages 27
12. Summary of Message Numbers 27
13. IANA Considerations 27
14. Security Considerations 28
15. References 29
   15.1. Normative References 29
   15.2. Informative References 30
Authors' Addresses 31
Trademark Notice 31

1. Introduction

The SSH transport layer is a secure, low level transport protocol. It provides strong encryption, cryptographic host authentication, and integrity protection.

Authentication in this protocol level is host-based; this protocol does not perform user authentication. A higher level protocol for user authentication can be designed on top of this protocol.

The protocol has been designed to be simple and flexible to allow parameter negotiation, and to minimize the number of round-trips. The key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated. It is expected that in most environments, only 2 round-trips will be needed for full key exchange, server authentication, service request, and acceptance notification of service request. The worst case is 3 round-trips.

2. Contributors

The major original contributors of this set of documents have been: Tatu Ylonen, Tero Kivinen, Timo J. Rinne, Sami Lehtinen (all of SSH Communications Security Corp), and Markku-Juhani O. Saarinen (University of Jyvaskyla). Darren Moffat was the original editor of this set of documents and also made very substantial contributions.

Many people contributed to the development of this document over the years. People who should be acknowledged include Mats Andersson, Ben Harris, Bill Sommerfeld, Brent McClure, Niels Moller, Damien Miller, Derek Fawcus, Frank Cusack, Heikki Nousiainen, Jakob Schlyter, Jeff Van Dyke, Jeffrey Altman, Jeffrey Hutzelman, Jon Bright, Joseph Galbraith, Ken Hornstein, Markus Friedl, Martin Forssen, Nicolas Williams, Niels Provos, Perry Metzger, Peter Gutmann, Simon Josefsson, Simon Tatham, Wei Dai, Denis Bider, der Mouse, and Tadayoshi Kohno. Listing their names here does not mean that they endorse this document, but that they have contributed to it.

3. Conventions Used in This Document

All documents related to the SSH protocols shall use the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" to describe requirements. These keywords are to be interpreted as described in [RFC2119].

The keywords "PRIVATE USE", "HIERARCHICAL ALLOCATION", "FIRST COME FIRST SERVED", "EXPERT REVIEW", "SPECIFICATION REQUIRED", "IESG APPROVAL", "IETF CONSENSUS", and "STANDARDS ACTION" that appear in this document when used to describe namespace allocation are to be interpreted as described in [RFC2434].

Protocol fields and possible values to fill them are defined in this set of documents. Protocol fields will be defined in the message definitions. As an example, SSH_MSG_CHANNEL_DATA is defined as follows.

```
byte      SSH_MSG_CHANNEL_DATA
uint32    recipient channel
string    data
```

Throughout these documents, when the fields are referenced, they will appear within single quotes. When values to fill those fields are referenced, they will appear within double quotes. Using the above example, possible values for 'data' are "foo" and "bar".

4. Connection Setup

SSH works over any 8-bit clean, binary-transparent transport. The underlying transport SHOULD protect against transmission errors, as such errors cause the SSH connection to terminate.

The client initiates the connection.

4.1. Use over TCP/IP

When used over TCP/IP, the server normally listens for connections on port 22. This port number has been registered with the IANA, and has been officially assigned for SSH.

4.2. Protocol Version Exchange

When the connection has been established, both sides MUST send an identification string. This identification string MUST be

```
SSH-protoversion-softwareversion SP comments CR LF
```

Since the protocol being defined in this set of documents is version 2.0, the 'protoversion' MUST be "2.0". The 'comments' string is OPTIONAL. If the 'comments' string is included, a 'space' character (denoted above as SP, ASCII 32) MUST separate the 'softwareversion' and 'comments' strings. The identification MUST be terminated by a single Carriage Return (CR) and a single Line Feed (LF) character (ASCII 13 and 10, respectively). Implementers who wish to maintain

compatibility with older, undocumented versions of this protocol may want to process the identification string without expecting the presence of the carriage return character for reasons described in Section 5 of this document. The null character MUST NOT be sent. The maximum length of the string is 255 characters, including the Carriage Return and Line Feed.

The part of the identification string preceding the Carriage Return and Line Feed is used in the Diffie-Hellman key exchange (see Section 8).

The server MAY send other lines of data before sending the version string. Each line SHOULD be terminated by a Carriage Return and Line Feed. Such lines MUST NOT begin with "SSH-", and SHOULD be encoded in ISO-10646 UTF-8 [RFC3629] (language is not specified). Clients MUST be able to process such lines. Such lines MAY be silently ignored, or MAY be displayed to the client user. If they are displayed, control character filtering, as discussed in [SSH-ARCH], SHOULD be used. The primary use of this feature is to allow TCP-wrappers to display an error message before disconnecting.

Both the 'protoversion' and 'softwareversion' strings MUST consist of printable US-ASCII characters, with the exception of whitespace characters and the minus sign (-). The 'softwareversion' string is primarily used to trigger compatibility extensions and to indicate the capabilities of an implementation. The 'comments' string SHOULD contain additional information that might be useful in solving user problems. As such, an example of a valid identification string is

```
SSH-2.0-billsSSH_3.6.3q3<CR><LF>
```

This identification string does not contain the optional 'comments' string and is thus terminated by a CR and LF immediately after the 'softwareversion' string.

Key exchange will begin immediately after sending this identifier. All packets following the identification string SHALL use the binary packet protocol, which is described in Section 6.

5. Compatibility With Old SSH Versions

As stated earlier, the 'protoversion' specified for this protocol is "2.0". Earlier versions of this protocol have not been formally documented, but it is widely known that they use 'protoversion' of "1.x" (e.g., "1.5" or "1.3"). At the time of this writing, many implementations of SSH are utilizing protocol version 2.0, but it is known that there are still devices using the previous versions. During the transition period, it is important to be able to work in a

way that is compatible with the installed SSH clients and servers that use the older version of the protocol. Information in this section is only relevant for implementations supporting compatibility with SSH versions 1.x. For those interested, the only known documentation of the 1.x protocol is contained in README files that are shipped along with the source code [ssh-1.2.30].

5.1. Old Client, New Server

Server implementations MAY support a configurable compatibility flag that enables compatibility with old versions. When this flag is on, the server SHOULD identify its 'protoversion' as "1.99". Clients using protocol 2.0 MUST be able to identify this as identical to "2.0". In this mode, the server SHOULD NOT send the Carriage Return character (ASCII 13) after the identification string.

In the compatibility mode, the server SHOULD NOT send any further data after sending its identification string until it has received an identification string from the client. The server can then determine whether the client is using an old protocol, and can revert to the old protocol if required. In the compatibility mode, the server MUST NOT send additional data before the identification string.

When compatibility with old clients is not needed, the server MAY send its initial key exchange data immediately after the identification string.

5.2. New Client, Old Server

Since the new client MAY immediately send additional data after its identification string (before receiving the server's identification string), the old protocol may already be corrupt when the client learns that the server is old. When this happens, the client SHOULD close the connection to the server, and reconnect using the old protocol.

5.3. Packet Size and Overhead

Some readers will worry about the increase in packet size due to new headers, padding, and the Message Authentication Code (MAC). The minimum packet size is in the order of 28 bytes (depending on negotiated algorithms). The increase is negligible for large packets, but very significant for one-byte packets (telnet-type sessions). There are, however, several factors that make this a non-issue in almost all cases:

- The minimum size of a TCP/IP header is 32 bytes. Thus, the increase is actually from 33 to 51 bytes (roughly).

- o The minimum size of the data field of an Ethernet packet is 46 bytes [RFC0894]. Thus, the increase is no more than 5 bytes. When Ethernet headers are considered, the increase is less than 10 percent.

- o The total fraction of telnet-type data in the Internet is negligible, even with increased packet sizes.

The only environment where the packet size increase is likely to have a significant effect is PPP [RFC1661] over slow modem lines (PPP compresses the TCP/IP headers, emphasizing the increase in packet size). However, with modern modems, the time needed to transfer is in the order of 2 milliseconds, which is a lot faster than people can type.

There are also issues related to the maximum packet size. To minimize delays in screen updates, one does not want excessively large packets for interactive sessions. The maximum packet size is negotiated separately for each channel.

6. Binary Packet Protocol

Each packet is in the following format:

```
uint32    packet_length
byte      padding_length
byte[n1]  payload; n1 = packet_length - padding_length - 1
byte[n2]  random padding; n2 = padding_length
byte[m]   mac (Message Authentication Code - MAC); m = mac_length
```

packet_length
: The length of the packet in bytes, not including 'mac' or the 'packet_length' field itself.

padding_length
: Length of 'random padding' (bytes).

payload
: The useful contents of the packet. If compression has been negotiated, this field is compressed. Initially, compression MUST be "none".

random padding
: Arbitrary-length padding, such that the total length of (packet_length || padding_length || payload || random padding) is a multiple of the cipher block size or 8, whichever is

larger. There MUST be at least four bytes of padding. The padding SHOULD consist of random bytes. The maximum amount of padding is 255 bytes.

mac
Message Authentication Code. If message authentication has been negotiated, this field contains the MAC bytes. Initially, the MAC algorithm MUST be "none".

Note that the length of the concatenation of 'packet_length', 'padding_length', 'payload', and 'random padding' MUST be a multiple of the cipher block size or 8, whichever is larger. This constraint MUST be enforced, even when using stream ciphers. Note that the 'packet_length' field is also encrypted, and processing it requires special care when sending or receiving packets. Also note that the insertion of variable amounts of 'random padding' may help thwart traffic analysis.

The minimum size of a packet is 16 (or the cipher block size, whichever is larger) bytes (plus 'mac'). Implementations SHOULD decrypt the length after receiving the first 8 (or cipher block size, whichever is larger) bytes of a packet.

6.1. Maximum Packet Length

All implementations MUST be able to process packets with an uncompressed payload length of 32768 bytes or less and a total packet size of 35000 bytes or less (including 'packet_length', 'padding_length', 'payload', 'random padding', and 'mac'). The maximum of 35000 bytes is an arbitrarily chosen value that is larger than the uncompressed length noted above. Implementations SHOULD support longer packets, where they might be needed. For example, if an implementation wants to send a very large number of certificates, the larger packets MAY be sent if the identification string indicates that the other party is able to process them. However, implementations SHOULD check that the packet length is reasonable in order for the implementation to avoid denial of service and/or buffer overflow attacks.

6.2. Compression

If compression has been negotiated, the 'payload' field (and only it) will be compressed using the negotiated algorithm. The 'packet_length' field and 'mac' will be computed from the compressed payload. Encryption will be done after compression.

Compression MAY be stateful, depending on the method. Compression MUST be independent for each direction, and implementations MUST allow independent choosing of the algorithm for each direction. In practice however, it is RECOMMENDED that the compression method be the same in both directions.

The following compression methods are currently defined:

none	REQUIRED	no compression
zlib	OPTIONAL	ZLIB (LZ77) compression

The "zlib" compression is described in [RFC1950] and in [RFC1951]. The compression context is initialized after each key exchange, and is passed from one packet to the next, with only a partial flush being performed at the end of each packet. A partial flush means that the current compressed block is ended and all data will be output. If the current block is not a stored block, one or more empty blocks are added after the current block to ensure that there are at least 8 bits, counting from the start of the end-of-block code of the current block to the end of the packet payload.

Additional methods may be defined as specified in [SSH-ARCH] and [SSH-NUMBERS].

6.3. Encryption

An encryption algorithm and a key will be negotiated during the key exchange. When encryption is in effect, the packet length, padding length, payload, and padding fields of each packet MUST be encrypted with the given algorithm.

The encrypted data in all packets sent in one direction SHOULD be considered a single data stream. For example, initialization vectors SHOULD be passed from the end of one packet to the beginning of the next packet. All ciphers SHOULD use keys with an effective key length of 128 bits or more.

The ciphers in each direction MUST run independently of each other. Implementations MUST allow the algorithm for each direction to be independently selected, if multiple algorithms are allowed by local policy. In practice however, it is RECOMMENDED that the same algorithm be used in both directions.

The following ciphers are currently defined:

3des-cbc	REQUIRED	three-key 3DES in CBC mode
blowfish-cbc	OPTIONAL	Blowfish in CBC mode
twofish256-cbc	OPTIONAL	Twofish in CBC mode, with a 256-bit key
twofish-cbc	OPTIONAL	alias for "twofish256-cbc" (this is being retained for historical reasons)
twofish192-cbc	OPTIONAL	Twofish with a 192-bit key
twofish128-cbc	OPTIONAL	Twofish with a 128-bit key
aes256-cbc	OPTIONAL	AES in CBC mode, with a 256-bit key
aes192-cbc	OPTIONAL	AES with a 192-bit key
aes128-cbc	RECOMMENDED	AES with a 128-bit key
serpent256-cbc	OPTIONAL	Serpent in CBC mode, with a 256-bit key
serpent192-cbc	OPTIONAL	Serpent with a 192-bit key
serpent128-cbc	OPTIONAL	Serpent with a 128-bit key
arcfour	OPTIONAL	the ARCFOUR stream cipher with a 128-bit key
idea-cbc	OPTIONAL	IDEA in CBC mode
cast128-cbc	OPTIONAL	CAST-128 in CBC mode
none	OPTIONAL	no encryption; NOT RECOMMENDED

The "3des-cbc" cipher is three-key triple-DES (encrypt-decrypt-encrypt), where the first 8 bytes of the key are used for the first encryption, the next 8 bytes for the decryption, and the following 8 bytes for the final encryption. This requires 24 bytes of key data (of which 168 bits are actually used). To implement CBC mode, outer chaining MUST be used (i.e., there is only one initialization vector). This is a block cipher with 8-byte blocks. This algorithm is defined in [FIPS-46-3]. Note that since this algorithm only has an effective key length of 112 bits ([SCHNEIER]), it does not meet the specifications that SSH encryption algorithms should use keys of 128 bits or more. However, this algorithm is still REQUIRED for historical reasons; essentially, all known implementations at the time of this writing support this algorithm, and it is commonly used because it is the fundamental interoperable algorithm. At some future time, it is expected that another algorithm, one with better strength, will become so prevalent and ubiquitous that the use of "3des-cbc" will be deprecated by another STANDARDS ACTION.

The "blowfish-cbc" cipher is Blowfish in CBC mode, with 128-bit keys [SCHNEIER]. This is a block cipher with 8-byte blocks.

The "twofish-cbc" or "twofish256-cbc" cipher is Twofish in CBC mode, with 256-bit keys as described [TWOFISH]. This is a block cipher with 16-byte blocks.

The "twofish192-cbc" cipher is the same as above, but with a 192-bit key.

The "twofish128-cbc" cipher is the same as above, but with a 128-bit key.

The "aes256-cbc" cipher is AES (Advanced Encryption Standard) [FIPS-197], in CBC mode. This version uses a 256-bit key.

The "aes192-cbc" cipher is the same as above, but with a 192-bit key.

The "aes128-cbc" cipher is the same as above, but with a 128-bit key.

The "serpent256-cbc" cipher in CBC mode, with a 256-bit key as described in the Serpent AES submission.

The "serpent192-cbc" cipher is the same as above, but with a 192-bit key.

The "serpent128-cbc" cipher is the same as above, but with a 128-bit key.

The "arcfour" cipher is the Arcfour stream cipher with 128-bit keys. The Arcfour cipher is believed to be compatible with the RC4 cipher [SCHNEIER]. Arcfour (and RC4) has problems with weak keys, and should be used with caution.

The "idea-cbc" cipher is the IDEA cipher in CBC mode [SCHNEIER].

The "cast128-cbc" cipher is the CAST-128 cipher in CBC mode with a 128-bit key [RFC2144].

The "none" algorithm specifies that no encryption is to be done. Note that this method provides no confidentiality protection, and it is NOT RECOMMENDED. Some functionality (e.g., password authentication) may be disabled for security reasons if this cipher is chosen.

Additional methods may be defined as specified in [SSH-ARCH] and in [SSH-NUMBERS].

6.4. Data Integrity

Data integrity is protected by including with each packet a MAC that is computed from a shared secret, packet sequence number, and the contents of the packet.

The message authentication algorithm and key are negotiated during key exchange. Initially, no MAC will be in effect, and its length MUST be zero. After key exchange, the 'mac' for the selected MAC algorithm will be computed before encryption from the concatenation of packet data:

```
mac = MAC(key, sequence_number || unencrypted_packet)
```

where unencrypted_packet is the entire packet without 'mac' (the length fields, 'payload' and 'random padding'), and sequence_number is an implicit packet sequence number represented as uint32. The sequence_number is initialized to zero for the first packet, and is incremented after every packet (regardless of whether encryption or MAC is in use). It is never reset, even if keys/algorithms are renegotiated later. It wraps around to zero after every 2^32 packets. The packet sequence_number itself is not included in the packet sent over the wire.

The MAC algorithms for each direction MUST run independently, and implementations MUST allow choosing the algorithm independently for both directions. In practice however, it is RECOMMENDED that the same algorithm be used in both directions.

The value of 'mac' resulting from the MAC algorithm MUST be transmitted without encryption as the last part of the packet. The number of 'mac' bytes depends on the algorithm chosen.

The following MAC algorithms are currently defined:

hmac-sha1	REQUIRED	HMAC-SHA1 (digest length = key length = 20)
hmac-sha1-96	RECOMMENDED	first 96 bits of HMAC-SHA1 (digest length = 12, key length = 20)
hmac-md5	OPTIONAL	HMAC-MD5 (digest length = key length = 16)
hmac-md5-96	OPTIONAL	first 96 bits of HMAC-MD5 (digest length = 12, key length = 16)
none	OPTIONAL	no MAC; NOT RECOMMENDED

The "hmac-*" algorithms are described in [RFC2104]. The "*-n" MACs use only the first n bits of the resulting value.

SHA-1 is described in [FIPS-180-2] and MD5 is described in [RFC1321].

Additional methods may be defined, as specified in [SSH-ARCH] and in [SSH-NUMBERS].

6.5. Key Exchange Methods

The key exchange method specifies how one-time session keys are generated for encryption and for authentication, and how the server authentication is done.

Two REQUIRED key exchange methods have been defined:

```
diffie-hellman-group1-sha1 REQUIRED
diffie-hellman-group14-sha1 REQUIRED
```

These methods are described in Section 8.

Additional methods may be defined as specified in [SSH-NUMBERS]. The name "diffie-hellman-group1-sha1" is used for a key exchange method using an Oakley group, as defined in [RFC2409]. SSH maintains its own group identifier space that is logically distinct from Oakley [RFC2412] and IKE; however, for one additional group, the Working Group adopted the number assigned by [RFC3526], using diffie-hellman-group14-sha1 for the name of the second defined group. Implementations should treat these names as opaque identifiers and should not assume any relationship between the groups used by SSH and the groups defined for IKE.

6.6. Public Key Algorithms

This protocol has been designed to operate with almost any public key format, encoding, and algorithm (signature and/or encryption).

There are several aspects that define a public key type:

- Key format: how is the key encoded and how are certificates represented. The key blobs in this protocol MAY contain certificates in addition to keys.

- Signature and/or encryption algorithms. Some key types may not support both signing and encryption. Key usage may also be restricted by policy statements (e.g., in certificates). In this case, different key types SHOULD be defined for the different policy alternatives.

- Encoding of signatures and/or encrypted data. This includes but is not limited to padding, byte order, and data formats.

The following public key and/or certificate formats are currently defined:

```
ssh-dss           REQUIRED     sign   Raw DSS Key
ssh-rsa           RECOMMENDED  sign   Raw RSA Key
pgp-sign-rsa      OPTIONAL     sign   OpenPGP certificates (RSA key)
pgp-sign-dss      OPTIONAL     sign   OpenPGP certificates (DSS key)
```

Additional key types may be defined, as specified in [SSH-ARCH] and in [SSH-NUMBERS].

The key type MUST always be explicitly known (from algorithm negotiation or some other source). It is not normally included in the key blob.

Certificates and public keys are encoded as follows:

```
string    certificate or public key format identifier
byte[n]   key/certificate data
```

The certificate part may be a zero length string, but a public key is required. This is the public key that will be used for authentication. The certificate sequence contained in the certificate blob can be used to provide authorization.

Public key/certificate formats that do not explicitly specify a signature format identifier MUST use the public key/certificate format identifier as the signature identifier.

Signatures are encoded as follows:

```
string    signature format identifier (as specified by the
          public key/certificate format)
byte[n]   signature blob in format specific encoding.
```

The "ssh-dss" key format has the following specific encoding:

```
string    "ssh-dss"
mpint     p
mpint     q
mpint     g
mpint     y
```

Here, the 'p', 'q', 'g', and 'y' parameters form the signature key blob.

Signing and verifying using this key format is done according to the Digital Signature Standard [FIPS-186-2] using the SHA-1 hash [FIPS-180-2].

The resulting signature is encoded as follows:

```
string    "ssh-dss"
string    dss_signature_blob
```

The value for 'dss_signature_blob' is encoded as a string containing r, followed by s (which are 160-bit integers, without lengths or padding, unsigned, and in network byte order).

The "ssh-rsa" key format has the following specific encoding:

```
string    "ssh-rsa"
mpint     e
mpint     n
```

Here the 'e' and 'n' parameters form the signature key blob.

Signing and verifying using this key format is performed according to the RSASSA-PKCS1-v1_5 scheme in [RFC3447] using the SHA-1 hash.

The resulting signature is encoded as follows:

```
string    "ssh-rsa"
string    rsa_signature_blob
```

The value for 'rsa_signature_blob' is encoded as a string containing s (which is an integer, without lengths or padding, unsigned, and in network byte order).

The "pgp-sign-rsa" method indicates the certificates, the public key, and the signature are in OpenPGP compatible binary format ([RFC2440]). This method indicates that the key is an RSA-key.

The "pgp-sign-dss" is as above, but indicates that the key is a DSS-key.

7. Key Exchange

Key exchange (kex) begins by each side sending name-lists of supported algorithms. Each side has a preferred algorithm in each category, and it is assumed that most implementations, at any given time, will use the same preferred algorithm. Each side MAY guess

which algorithm the other side is using, and MAY send an initial key exchange packet according to the algorithm, if appropriate for the preferred method.

The guess is considered wrong if:

- o the kex algorithm and/or the host key algorithm is guessed wrong (server and client have different preferred algorithm), or

- o if any of the other algorithms cannot be agreed upon (the procedure is defined below in Section 7.1).

Otherwise, the guess is considered to be right, and the optimistically sent packet MUST be handled as the first key exchange packet.

However, if the guess was wrong, and a packet was optimistically sent by one or both parties, such packets MUST be ignored (even if the error in the guess would not affect the contents of the initial packet(s)), and the appropriate side MUST send the correct initial packet.

A key exchange method uses explicit server authentication if the key exchange messages include a signature or other proof of the server's authenticity. A key exchange method uses implicit server authentication if, in order to prove its authenticity, the server also has to prove that it knows the shared secret, K, by sending a message and a corresponding MAC that the client can verify.

The key exchange method defined by this document uses explicit server authentication. However, key exchange methods with implicit server authentication MAY be used with this protocol. After a key exchange with implicit server authentication, the client MUST wait for a response to its service request message before sending any further data.

7.1. Algorithm Negotiation

Key exchange begins by each side sending the following packet:

```
byte         SSH_MSG_KEXINIT
byte[16]     cookie (random bytes)
name-list    kex_algorithms
name-list    server_host_key_algorithms
name-list    encryption_algorithms_client_to_server
name-list    encryption_algorithms_server_to_client
name-list    mac_algorithms_client_to_server
name-list    mac_algorithms_server_to_client
name-list    compression_algorithms_client_to_server
name-list    compression_algorithms_server_to_client
name-list    languages_client_to_server
name-list    languages_server_to_client
boolean      first_kex_packet_follows
uint32       0 (reserved for future extension)
```

Each of the algorithm name-lists MUST be a comma-separated list of algorithm names (see Algorithm Naming in [SSH-ARCH] and additional information in [SSH-NUMBERS]). Each supported (allowed) algorithm MUST be listed in order of preference, from most to least.

The first algorithm in each name-list MUST be the preferred (guessed) algorithm. Each name-list MUST contain at least one algorithm name.

cookie
: The 'cookie' MUST be a random value generated by the sender. Its purpose is to make it impossible for either side to fully determine the keys and the session identifier.

kex_algorithms
: Key exchange algorithms were defined above. The first algorithm MUST be the preferred (and guessed) algorithm. If both sides make the same guess, that algorithm MUST be used. Otherwise, the following algorithm MUST be used to choose a key exchange method: Iterate over client's kex algorithms, one at a time. Choose the first algorithm that satisfies the following conditions:

  + the server also supports the algorithm,

  + if the algorithm requires an encryption-capable host key, there is an encryption-capable algorithm on the server's server_host_key_algorithms that is also supported by the client, and

+ if the algorithm requires a signature-capable host key, there is a signature-capable algorithm on the server's server_host_key_algorithms that is also supported by the client.

If no algorithm satisfying all these conditions can be found, the connection fails, and both sides MUST disconnect.

server_host_key_algorithms
A name-list of the algorithms supported for the server host key. The server lists the algorithms for which it has host keys; the client lists the algorithms that it is willing to accept. There MAY be multiple host keys for a host, possibly with different algorithms.

Some host keys may not support both signatures and encryption (this can be determined from the algorithm), and thus not all host keys are valid for all key exchange methods.

Algorithm selection depends on whether the chosen key exchange algorithm requires a signature or an encryption-capable host key. It MUST be possible to determine this from the public key algorithm name. The first algorithm on the client's name-list that satisfies the requirements and is also supported by the server MUST be chosen. If there is no such algorithm, both sides MUST disconnect.

encryption_algorithms
A name-list of acceptable symmetric encryption algorithms (also known as ciphers) in order of preference. The chosen encryption algorithm to each direction MUST be the first algorithm on the client's name-list that is also on the server's name-list. If there is no such algorithm, both sides MUST disconnect.

Note that "none" must be explicitly listed if it is to be acceptable. The defined algorithm names are listed in Section 6.3.

mac_algorithms
A name-list of acceptable MAC algorithms in order of preference. The chosen MAC algorithm MUST be the first algorithm on the client's name-list that is also on the server's name-list. If there is no such algorithm, both sides MUST disconnect.

Note that "none" must be explicitly listed if it is to be acceptable. The MAC algorithm names are listed in Section 6.4.

compression_algorithms
A name-list of acceptable compression algorithms in order of preference. The chosen compression algorithm MUST be the first algorithm on the client's name-list that is also on the server's name-list. If there is no such algorithm, both sides MUST disconnect.

Note that "none" must be explicitly listed if it is to be acceptable. The compression algorithm names are listed in Section 6.2.

languages
This is a name-list of language tags in order of preference [RFC3066]. Both parties MAY ignore this name-list. If there are no language preferences, this name-list SHOULD be empty as defined in Section 5 of [SSH-ARCH]. Language tags SHOULD NOT be present unless they are known to be needed by the sending party.

first_kex_packet_follows
Indicates whether a guessed key exchange packet follows. If a guessed packet will be sent, this MUST be TRUE. If no guessed packet will be sent, this MUST be FALSE.

After receiving the SSH_MSG_KEXINIT packet from the other side, each party will know whether their guess was right. If the other party's guess was wrong, and this field was TRUE, the next packet MUST be silently ignored, and both sides MUST then act as determined by the negotiated key exchange method. If the guess was right, key exchange MUST continue using the guessed packet.

After the SSH_MSG_KEXINIT message exchange, the key exchange algorithm is run. It may involve several packet exchanges, as specified by the key exchange method.

Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message (Section 7.3), it MUST NOT send any messages other than:

- Transport layer generic messages (1 to 19) (but SSH_MSG_SERVICE_REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);

- Algorithm negotiation messages (20 to 29) (but further SSH_MSG_KEXINIT messages MUST NOT be sent);

- Specific key exchange method messages (30 to 49).

The provisions of Section 11 apply to unrecognized messages.

Note, however, that during a key re-exchange, after sending a SSH_MSG_KEXINIT message, each party MUST be prepared to process an arbitrary number of messages that may be in-flight before receiving a SSH_MSG_KEXINIT message from the other party.

7.2. Output from Key Exchange

The key exchange produces two values: a shared secret K, and an exchange hash H. Encryption and authentication keys are derived from these. The exchange hash H from the first key exchange is additionally used as the session identifier, which is a unique identifier for this connection. It is used by authentication methods as a part of the data that is signed as a proof of possession of a private key. Once computed, the session identifier is not changed, even if keys are later re-exchanged.

Each key exchange method specifies a hash function that is used in the key exchange. The same hash algorithm MUST be used in key derivation. Here, we'll call it HASH.

Encryption keys MUST be computed as HASH, of a known value and K, as follows:

- o Initial IV client to server: HASH(K || H || "A" || session_id) (Here K is encoded as mpint and "A" as byte and session_id as raw data. "A" means the single character A, ASCII 65).

- o Initial IV server to client: HASH(K || H || "B" || session_id)

- o Encryption key client to server: HASH(K || H || "C" || session_id)

- o Encryption key server to client: HASH(K || H || "D" || session_id)

- o Integrity key client to server: HASH(K || H || "E" || session_id)

- o Integrity key server to client: HASH(K || H || "F" || session_id)

Key data MUST be taken from the beginning of the hash output. As many bytes as needed are taken from the beginning of the hash value. If the key length needed is longer than the output of the HASH, the key is extended by computing HASH of the concatenation of K and H and the entire key so far, and appending the resulting bytes (as many as HASH generates) to the key. This process is repeated until enough key material is available; the key is taken from the beginning of this value. In other words:

```
K1 = HASH(K || H || X || session_id)   (X is e.g., "A")
K2 = HASH(K || H || K1)
K3 = HASH(K || H || K1 || K2)
...
key = K1 || K2 || K3 || ...
```

This process will lose entropy if the amount of entropy in K is larger than the internal state size of HASH.

7.3. Taking Keys Into Use

Key exchange ends by each side sending an SSH_MSG_NEWKEYS message. This message is sent with the old keys and algorithms. All messages sent after this message MUST use the new keys and algorithms.

When this message is received, the new keys and algorithms MUST be used for receiving.

The purpose of this message is to ensure that a party is able to respond with an SSH_MSG_DISCONNECT message that the other party can understand if something goes wrong with the key exchange.

```
byte      SSH_MSG_NEWKEYS
```

8. Diffie-Hellman Key Exchange

The Diffie-Hellman (DH) key exchange provides a shared secret that cannot be determined by either party alone. The key exchange is combined with a signature with the host key to provide host authentication. This key exchange method provides explicit server authentication as defined in Section 7.

The following steps are used to exchange a key. In this, C is the client; S is the server; p is a large safe prime; g is a generator for a subgroup of GF(p); q is the order of the subgroup; V_S is S's identification string; V_C is C's identification string; K_S is S's public host key; I_C is C's SSH_MSG_KEXINIT message and I_S is S's SSH_MSG_KEXINIT message that have been exchanged before this part begins.

1. C generates a random number x (1 < x < q) and computes e = g^x mod p. C sends e to S.

2. S generates a random number y (0 < y < q) and computes f = g^y mod p. S receives e. It computes K = e^y mod p, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K) (these elements are encoded according to their types; see below), and signature s on H with its private host key. S sends (K_S || f || s) to C. The signing operation may involve a second hashing operation.

3. C verifies that K_S really is the host key for S (e.g., using certificates or a local database). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes K = f^x mod p, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K), and verifies the signature s on H.

Values of 'e' or 'f' that are not in the range [1, p-1] MUST NOT be sent or accepted by either side. If this condition is violated, the key exchange fails.

This is implemented with the following messages. The hash algorithm for computing the exchange hash is defined by the method name, and is called HASH. The public key algorithm for signing is negotiated with the SSH_MSG_KEXINIT messages.

First, the client sends the following:

```
byte      SSH_MSG_KEXDH_INIT
mpint     e
```

The server then responds with the following:

```
byte      SSH_MSG_KEXDH_REPLY
string    server public host key and certificates (K_S)
mpint     f
string    signature of H
```

The hash H is computed as the HASH hash of the concatenation of the following:

```
string    V_C, the client's identification string (CR and LF
          excluded)
string    V_S, the server's identification string (CR and LF
          excluded)
string    I_C, the payload of the client's SSH_MSG_KEXINIT
string    I_S, the payload of the server's SSH_MSG_KEXINIT
string    K_S, the host key
mpint     e, exchange value sent by the client
mpint     f, exchange value sent by the server
mpint     K, the shared secret
```

This value is called the exchange hash, and it is used to authenticate the key exchange. The exchange hash SHOULD be kept secret.

The signature algorithm MUST be applied over H, not the original data. Most signature algorithms include hashing and additional padding (e.g., "ssh-dss" specifies SHA-1 hashing). In that case, the data is first hashed with HASH to compute H, and H is then hashed with SHA-1 as part of the signing operation.

8.1. diffie-hellman-group1-sha1

The "diffie-hellman-group1-sha1" method specifies the Diffie-Hellman key exchange with SHA-1 as HASH, and Oakley Group 2 [RFC2409] (1024-bit MODP Group). This method MUST be supported for interoperability as all of the known implementations currently support it. Note that this method is named using the phrase "group1", even though it specifies the use of Oakley Group 2.

8.2. diffie-hellman-group14-sha1

The "diffie-hellman-group14-sha1" method specifies a Diffie-Hellman key exchange with SHA-1 as HASH and Oakley Group 14 [RFC3526] (2048-bit MODP Group), and it MUST also be supported.

9. Key Re-Exchange

Key re-exchange is started by sending an SSH_MSG_KEXINIT packet when not already doing a key exchange (as described in Section 7.1). When this message is received, a party MUST respond with its own SSH_MSG_KEXINIT message, except when the received SSH_MSG_KEXINIT already was a reply. Either party MAY initiate the re-exchange, but roles MUST NOT be changed (i.e., the server remains the server, and the client remains the client).

Key re-exchange is performed using whatever encryption was in effect when the exchange was started. Encryption, compression, and MAC methods are not changed before a new SSH_MSG_NEWKEYS is sent after the key exchange (as in the initial key exchange). Re-exchange is processed identically to the initial key exchange, except for the session identifier that will remain unchanged. It is permissible to change some or all of the algorithms during the re-exchange. Host keys can also change. All keys and initialization vectors are recomputed after the exchange. Compression and encryption contexts are reset.

It is RECOMMENDED that the keys be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. However, since the re-exchange is a public key operation, it requires a fair amount of processing power and should not be performed too often.

More application data may be sent after the SSH_MSG_NEWKEYS packet has been sent; key exchange does not affect the protocols that lie above the SSH transport layer.

10. Service Request

After the key exchange, the client requests a service. The service is identified by a name. The format of names and procedures for defining new names are defined in [SSH-ARCH] and [SSH-NUMBERS].

Currently, the following names have been reserved:

```
ssh-userauth
ssh-connection
```

Similar local naming policy is applied to the service names, as is applied to the algorithm names. A local service should use the PRIVATE USE syntax of "servicename@domain".

```
byte      SSH_MSG_SERVICE_REQUEST
string    service name
```

If the server rejects the service request, it SHOULD send an appropriate SSH_MSG_DISCONNECT message and MUST disconnect.

When the service starts, it may have access to the session identifier generated during the key exchange.

If the server supports the service (and permits the client to use it), it MUST respond with the following:

```
byte      SSH_MSG_SERVICE_ACCEPT
string    service name
```

Message numbers used by services should be in the area reserved for them (see [SSH-ARCH] and [SSH-NUMBERS]). The transport level will continue to process its own messages.

Note that after a key exchange with implicit server authentication, the client MUST wait for a response to its service request message before sending any further data.

11. Additional Messages

Either party may send any of the following messages at any time.

11.1. Disconnection Message

```
byte      SSH_MSG_DISCONNECT
uint32    reason code
string    description in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

This message causes immediate termination of the connection. All implementations MUST be able to process this message; they SHOULD be able to send this message.

The sender MUST NOT send or receive any data after this message, and the recipient MUST NOT accept any data after receiving this message. The Disconnection Message 'description' string gives a more specific explanation in a human-readable form. The Disconnection Message 'reason code' gives the reason in a more machine-readable format (suitable for localization), and can have the values as displayed in the table below. Note that the decimal representation is displayed in this table for readability, but the values are actually uint32 values.

Symbolic name	reason code
SSH_DISCONNECT_HOST_NOT_ALLOWED_TO_CONNECT	1
SSH_DISCONNECT_PROTOCOL_ERROR	2
SSH_DISCONNECT_KEY_EXCHANGE_FAILED	3
SSH_DISCONNECT_RESERVED	4
SSH_DISCONNECT_MAC_ERROR	5
SSH_DISCONNECT_COMPRESSION_ERROR	6
SSH_DISCONNECT_SERVICE_NOT_AVAILABLE	7
SSH_DISCONNECT_PROTOCOL_VERSION_NOT_SUPPORTED	8
SSH_DISCONNECT_HOST_KEY_NOT_VERIFIABLE	9
SSH_DISCONNECT_CONNECTION_LOST	10
SSH_DISCONNECT_BY_APPLICATION	11
SSH_DISCONNECT_TOO_MANY_CONNECTIONS	12
SSH_DISCONNECT_AUTH_CANCELLED_BY_USER	13
SSH_DISCONNECT_NO_MORE_AUTH_METHODS_AVAILABLE	14
SSH_DISCONNECT_ILLEGAL_USER_NAME	15

If the 'description' string is displayed, the control character filtering discussed in [SSH-ARCH] should be used to avoid attacks by sending terminal control characters.

Requests for assignments of new Disconnection Message 'reason code' values (and associated 'description' text) in the range of 0x00000010 to 0xFDFFFFFF MUST be done through the IETF CONSENSUS method, as described in [RFC2434]. The Disconnection Message 'reason code' values in the range of 0xFE000000 through 0xFFFFFFFF are reserved for PRIVATE USE. As noted, the actual instructions to the IANA are in [SSH-NUMBERS].

11.2. Ignored Data Message

```
byte      SSH_MSG_IGNORE
string    data
```

All implementations MUST understand (and ignore) this message at any time (after receiving the identification string). No implementation is required to send them. This message can be used as an additional protection measure against advanced traffic analysis techniques.

11.3. Debug Message

```
byte      SSH_MSG_DEBUG
boolean   always_display
string    message in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

All implementations MUST understand this message, but they are allowed to ignore it. This message is used to transmit information that may help debugging. If 'always_display' is TRUE, the message SHOULD be displayed. Otherwise, it SHOULD NOT be displayed unless debugging information has been explicitly requested by the user.

The 'message' doesn't need to contain a newline. It is, however, allowed to consist of multiple lines separated by CRLF (Carriage Return - Line Feed) pairs.

If the 'message' string is displayed, the terminal control character filtering discussed in [SSH-ARCH] should be used to avoid attacks by sending terminal control characters.

11.4. Reserved Messages

An implementation MUST respond to all unrecognized messages with an SSH_MSG_UNIMPLEMENTED message in the order in which the messages were received. Such messages MUST be otherwise ignored. Later protocol versions may define other meanings for these message types.

```
byte      SSH_MSG_UNIMPLEMENTED
uint32    packet sequence number of rejected message
```

12. Summary of Message Numbers

The following is a summary of messages and their associated message number.

```
SSH_MSG_DISCONNECT             1
SSH_MSG_IGNORE                 2
SSH_MSG_UNIMPLEMENTED          3
SSH_MSG_DEBUG                  4
SSH_MSG_SERVICE_REQUEST        5
SSH_MSG_SERVICE_ACCEPT         6
SSH_MSG_KEXINIT                20
SSH_MSG_NEWKEYS                21
```

Note that numbers 30-49 are used for kex packets. Different kex methods may reuse message numbers in this range.

13. IANA Considerations

This document is part of a set. The IANA considerations for the SSH protocol as defined in [SSH-ARCH], [SSH-USERAUTH], [SSH-CONNECT], and this document, are detailed in [SSH-NUMBERS].

14. Security Considerations

This protocol provides a secure encrypted channel over an insecure network. It performs server host authentication, key exchange, encryption, and integrity protection. It also derives a unique session ID that may be used by higher-level protocols.

Full security considerations for this protocol are provided in [SSH-ARCH].

15. References

15.1. Normative References

[SSH-ARCH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[SSH-USERAUTH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, January 2006.

[SSH-CONNECT] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

[SSH-NUMBERS] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, January 2006.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm ", RFC 1321, April 1992.

[RFC1950] Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2144] Adams, C., "The CAST-128 Encryption Algorithm", RFC 2144, May 1997.

[RFC2409] Harkins, D. and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC2440] Callas, J., Donnerhacke, L., Finney, H., and R. Thayer, "OpenPGP Message Format", RFC 2440, November 1998.

[RFC3066] Alvestrand, H., "Tags for the Identification of Languages", BCP 47, RFC 3066, January 2001.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, May 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[FIPS-180-2] US National Institute of Standards and Technology, "Secure Hash Standard (SHS)", Federal Information Processing Standards Publication 180-2, August 2002.

[FIPS-186-2] US National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication 186-2, January 2000.

[FIPS-197] US National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication 197, November 2001.

[FIPS-46-3] US National Institute of Standards and Technology, "Data Encryption Standard (DES)", Federal Information Processing Standards Publication 46-3, October 1999.

[SCHNEIER] Schneier, B., "Applied Cryptography Second Edition: protocols algorithms and source in code in C", John Wiley and Sons, New York, NY, 1996.

[TWOFISH] Schneier, B., "The Twofish Encryptions Algorithm: A 128-Bit Block Cipher, 1st Edition", March 1999.

15.2. Informative References

[RFC0894] Hornig, C., "Standard for the transmission of IP datagrams over Ethernet networks", STD 41, RFC 894, April 1984.

[RFC1661] Simpson, W., "The Point-to-Point Protocol (PPP)", STD 51, RFC 1661, July 1994.

[RFC2412] Orman, H., "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.

[ssh-1.2.30] Ylonen, T., "ssh-1.2.30/RFC", File within compressed tarball ftp://ftp.funet.fi/pub/unix/security/login/ssh/ssh-1.2.30.tar.gz, November 1995.

Authors' Addresses

Tatu Ylonen
SSH Communications Security Corp
Valimotie 17
00380 Helsinki
Finland

EMail: ylo@ssh.com

Chris Lonvick (editor)
Cisco Systems, Inc.
12515 Research Blvd.
Austin 78759
USA

EMail: clonvick@cisco.com

Trademark Notice

"ssh" is a registered trademark in the United States and/or other countries.

Full Copyright Statement

Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).