Network Working Group
Request for Comments: 4462
Category: Standards Track

J. Hutzelman
CMU
J. Salowey
Cisco Systems
J. Galbraith
Van Dyke Technologies, Inc.
V. Welch
U Chicago / ANL

May 2006

Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol

Status of This Memo

Copyright Notice

Abstract

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

The Generic Security Service Application Program Interface (GSS-API) provides security services to callers in a mechanism-independent fashion.

This memo describes methods for using the GSS-API for authentication and key exchange in SSH. It defines an SSH user authentication method that uses a specified GSS-API mechanism to authenticate a user, and a family of SSH key exchange methods that use GSS-API to authenticate a Diffie-Hellman key exchange.

This memo also defines a new host public key algorithm that can be used when no operations are needed using a host's public key, and a new user authentication method that allows an authorization name to be used in conjunction with any authentication that has already occurred as a side-effect of GSS-API-based key exchange.

Table of Contents

1. Introduction

This document describes the methods used to perform key exchange and user authentication in the Secure Shell protocol using the GSS-API. To do this, it defines a family of key exchange methods, two user authentication methods, and a new host key algorithm. These definitions allow any GSS-API mechanism to be used with the Secure Shell protocol.

This document should be read only after reading the documents describing the SSH protocol architecture [SSH-ARCH], transport layer protocol [SSH-TRANSPORT], and user authentication protocol [SSH-USERAUTH]. This document freely uses terminology and notation from the architecture document without reference or further explanation.

1.1. SSH Terminology

The data types used in the packets are defined in the SSH architecture document [SSH-ARCH]. It is particularly important to note the definition of string allows binary content.

The SSH_MSG_USERAUTH_REQUEST packet refers to a service; this service name is an SSH service name and has no relationship to GSS-API service names. Currently, the only defined service name is "ssh-connection", which refers to the SSH connection protocol [SSH-CONNECT].

1.2. Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [KEYWORDS].

2. GSS-API-Authenticated Diffie-Hellman Key Exchange

This section defines a class of key exchange methods that combine the Diffie-Hellman key exchange from Section 8 of [SSH-TRANSPORT] with mutual authentication using GSS-API.

Since the GSS-API key exchange methods described in this section do not require the use of public key signature or encryption algorithms, they MAY be used with any host key algorithm, including the "null" algorithm described in Section 5.

2.1. Generic GSS-API Key Exchange

The following symbols are used in this description:

- C is the client, and S is the server
- p is a large safe prime, g is a generator for a subgroup of GF(p), and q is the order of the subgroup
- V_S is S's version string, and V_C is C's version string
- I_C is C's KEXINIT message, and I_S is S's KEXINIT message

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

2. C calls GSS_Init_sec_context(), using the most recent reply token received from S during this exchange, if any. For this call, the client MUST set mutual_req_flag to "true" to request that mutual authentication be performed. It also MUST set integ_req_flag to "true" to request that per-message integrity protection be supported for this context. In addition, deleg_req_flag MAY be set to "true" to request access delegation, if requested by the user. Since the key exchange process authenticates only the host, the setting of anon_req_flag is immaterial to this process. If the client does not support the "gssapi-keyex" user authentication method described in Section 4, or does not intend to use that method in conjunction with the GSS-API context established during key exchange, then anon_req_flag SHOULD be set to "true". Otherwise, this flag MAY be set to true if the client wishes to hide its identity. Since the key exchange process will involve the exchange of only a single token once the context has been established, it is not necessary that the GSS-API context support detection of replayed or out-of-sequence tokens. Thus, replay_det_req_flag and sequence_req_flag need not be set for this process. These flags SHOULD be set to "false".

   * If the resulting major_status code is GSS_S_COMPLETE and the mutual_state flag is not true, then mutual authentication has not been established, and the key exchange MUST fail.

   * If the resulting major_status code is GSS_S_COMPLETE and the integ_avail flag is not true, then per-message integrity protection is not available, and the key exchange MUST fail.

   * If the resulting major_status code is GSS_S_COMPLETE and both the mutual_state and integ_avail flags are true, the resulting output token is sent to S.

   * If the resulting major_status code is GSS_S_CONTINUE_NEEDED, the output_token is sent to S, which will reply with a new token to be provided to GSS_Init_sec_context().

   * The client MUST also include "e" with the first message it sends to the server during this process; if the server receives more than one "e" or none at all, the key exchange fails.

   * It is an error if the call does not produce a token of non-zero length to be sent to the server. In this case, the key exchange MUST fail.

3. S calls GSS_Accept_sec_context(), using the token received from C.

   * If the resulting major_status code is GSS_S_COMPLETE and the mutual_state flag is not true, then mutual authentication has not been established, and the key exchange MUST fail.

   * If the resulting major_status code is GSS_S_COMPLETE and the integ_avail flag is not true, then per-message integrity protection is not available, and the key exchange MUST fail.

   * If the resulting major_status code is GSS_S_COMPLETE and both the mutual_state and integ_avail flags are true, then the security context has been established, and processing continues with step 4.

   * If the resulting major_status code is GSS_S_CONTINUE_NEEDED, then the output token is sent to C, and processing continues with step 2.

   * If the resulting major_status code is GSS_S_COMPLETE, but a non-zero-length reply token is returned, then that token is sent to the client.

4. S generates a random number y (0 < y < q) and computes f = g^y mod p. It computes K = e ^ y mod p, and H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K). It then calls GSS_GetMIC() to obtain a GSS-API message integrity code for H. S then sends f and the message integrity code (MIC) to C.

5. This step is performed only (1) if the server's final call to GSS_Accept_sec_context() produced a non-zero-length final reply token to be sent to the client and (2) if no previous call by the client to GSS_Init_sec_context() has resulted in a major_status of GSS_S_COMPLETE. Under these conditions, the client makes an

additional call to GSS_Init_sec_context() to process the final reply token. This call is made exactly as described above. However, if the resulting major_status is anything other than GSS_S_COMPLETE, or a non-zero-length token is returned, it is an error and the key exchange MUST fail.

6. C computes K = f^x mod p, and H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K). It then calls GSS_VerifyMIC() to verify that the MIC sent by S matches H. If the MIC is not successfully verified, the key exchange MUST fail.

Either side MUST NOT send or accept e or f values that are not in the range [1, p-1]. If this condition is violated, the key exchange fails.

If any call to GSS_Init_sec_context() or GSS_Accept_sec_context() returns a major_status other than GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED, or any other GSS-API call returns a major_status other than GSS_S_COMPLETE, the key exchange fails. In this case, several mechanisms are available for communicating error information to the peer before terminating the connection as required by [SSH-TRANSPORT]:

- If the key exchange fails due to any GSS-API error on the server (including errors returned by GSS_Accept_sec_context()), the server MAY send a message informing the client of the details of the error. In this case, if an error token is also sent (see below), then this message MUST be sent before the error token.

- If the key exchange fails due to a GSS-API error returned from the server's call to GSS_Accept_sec_context(), and an "error token" is also returned, then the server SHOULD send the error token to the client to allow completion of the GSS security exchange.

- If the key exchange fails due to a GSS-API error returned from the client's call to GSS_Init_sec_context(), and an "error token" is also returned, then the client SHOULD send the error token to the server to allow completion of the GSS security exchange.

As noted in Section 9, it may be desirable under site security policy to obscure information about the precise nature of the error; thus, it is RECOMMENDED that implementations provide a method to suppress these messages as a matter of policy.

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 group used for Diffie-Hellman key exchange and the underlying GSS-API mechanism are also defined by the method name.

After the client's first call to GSS_Init_sec_context(), it sends the following:

```
byte      SSH_MSG_KEXGSS_INIT
string    output_token (from GSS_Init_sec_context())
mpint     e
```

Upon receiving the SSH_MSG_KEXGSS_INIT message, the server MAY send the following message, prior to any other messages, to inform the client of its host key.

```
byte      SSH_MSG_KEXGSS_HOSTKEY
string    server public host key and certificates (K_S)
```

Since this key exchange method does not require the host key to be used for any encryption operations, this message is OPTIONAL. If the "null" host key algorithm described in Section 5 is used, this message MUST NOT be sent. If this message is sent, the server public host key(s) and/or certificate(s) in this message are encoded as a single string, in the format specified by the public key type in use (see [SSH-TRANSPORT], Section 6.6).

In traditional SSH deployments, host keys are normally expected to change infrequently, and there is often no mechanism for validating host keys not already known to the client. As a result, the use of a new host key by an already-known host is usually considered an indication of a possible man-in-the-middle attack, and clients often present strong warnings and/or abort the connection in such cases.

By contrast, when GSS-API-based key exchange is used, host keys sent via the SSH_MSG_KEXGSS_HOSTKEY message are authenticated as part of the GSS-API key exchange, even when previously unknown to the client. Further, in environments in which GSS-API-based key exchange is used heavily, it is possible and even likely that host keys will change much more frequently and/or without advance warning.

Therefore, when a new key for an already-known host is received via the SSH_MSG_KEXGSS_HOSTKEY message, clients SHOULD NOT issue strong warnings or abort the connection, provided the GSS-API-based key exchange succeeds.

In order to facilitate key re-exchange after the user's GSS-API credentials have expired, client implementations SHOULD store host keys received via SSH_MSG_KEXGSS_HOSTKEY for the duration of the session, even when such keys are not stored for long-term use.

Each time the server's call to GSS_Accept_sec_context() returns a major_status code of GSS_S_CONTINUE_NEEDED, it sends the following reply to the client:

```
byte      SSH_MSG_KEXGSS_CONTINUE
string    output_token (from GSS_Accept_sec_context())
```

If the client receives this message after a call to GSS_Init_sec_context() has returned a major_status code of GSS_S_COMPLETE, a protocol error has occurred and the key exchange MUST fail.

Each time the client receives the message described above, it makes another call to GSS_Init_sec_context(). It then sends the following:

```
byte      SSH_MSG_KEXGSS_CONTINUE
string    output_token (from GSS_Init_sec_context())
```

The server and client continue to trade these two messages as long as the server's calls to GSS_Accept_sec_context() result in major_status codes of GSS_S_CONTINUE_NEEDED. When a call results in a major_status code of GSS_S_COMPLETE, it sends one of two final messages.

If the server's final call to GSS_Accept_sec_context() (resulting in a major_status code of GSS_S_COMPLETE) returns a non-zero-length token to be sent to the client, it sends the following:

```
byte      SSH_MSG_KEXGSS_COMPLETE
mpint     f
string    per_msg_token (MIC of H)
boolean   TRUE
string    output_token (from GSS_Accept_sec_context())
```

If the client receives this message after a call to GSS_Init_sec_context() has returned a major_status code of GSS_S_COMPLETE, a protocol error has occurred and the key exchange MUST fail.

If the server's final call to GSS_Accept_sec_context() (resulting in a major_status code of GSS_S_COMPLETE) returns a zero-length token or no token at all, it sends the following:

```
byte      SSH_MSG_KEXGSS_COMPLETE
mpint     f
string    per_msg_token (MIC of H)
boolean   FALSE
```

If the client receives this message when no call to GSS_Init_sec_context() has yet resulted in a major_status code of GSS_S_COMPLETE, a protocol error has occurred and the key exchange MUST fail.

If either the client's call to GSS_Init_sec_context() or the server's call to GSS_Accept_sec_context() returns an error status and produces an output token (called an "error token"), then the following SHOULD be sent to convey the error information to the peer:

```
byte      SSH_MSG_KEXGSS_CONTINUE
string    error_token
```

If a server sends both this message and an SSH_MSG_KEXGSS_ERROR message, the SSH_MSG_KEXGSS_ERROR message MUST be sent first, to allow clients to record and/or display the error information before processing the error token. This is important because a client processing an error token will likely disconnect without reading any further messages.

In the event of a GSS-API error on the server, the server MAY send the following message before terminating the connection:

```
byte      SSH_MSG_KEXGSS_ERROR
uint32    major_status
uint32    minor_status
string    message
string    language tag
```

The message text MUST be encoded in the UTF-8 encoding described in [UTF8]. Language tags are those described in [LANGTAG]. Note that the message text may contain multiple lines separated by carriage return-line feed (CRLF) sequences. Application developers should take this into account when displaying these messages.

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

```
string    V_C, the client's version string (CR, NL excluded)
string    V_S, the server's version string (CR, NL 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. If no SSH_MSG_KEXGSS_HOSTKEY message has been sent by the server or received by the client, then the empty string is used in place of K_S when computing the exchange hash.

The GSS_GetMIC call MUST be applied over H, not the original data.

2.2. Group Exchange

This section describes a modification to the generic GSS-API-authenticated Diffie-Hellman key exchange to allow the negotiation of the group to be used, using a method based on that described in [GROUP-EXCHANGE].

The server keeps a list of safe primes and corresponding generators that it can select from. These are chosen as described in Section 3 of [GROUP-EXCHANGE]. The client requests a modulus from the server, indicating the minimum, maximum, and preferred sizes; the server responds with a suitable modulus and generator. The exchange then proceeds as described in Section 2.1 above.

This description uses the following symbols, in addition to those defined above:

- n is the size of the modulus p in bits that the client would like to receive from the server
- min and max are the minimal and maximal sizes of p in bits that are acceptable to the client

1. C sends "min || n || max" to S, indicating the minimal acceptable group size, the preferred size of the group, and the maximal group size in bits the client will accept.
2. S finds a group that best matches the client's request, and sends "p || g" to C.
3. The exchange proceeds as described in Section 2.1 above, beginning with step 1, except that the exchange hash is computed as described below.

Servers and clients SHOULD support groups with a modulus length of k bits, where 1024 <= k <= 8192. The recommended values for min and max are 1024 and 8192, respectively.

This is implemented using the following messages, in addition to those described above:

First, the client sends:

```
byte      SSH_MSG_KEXGSS_GROUPREQ
uint32    min, minimal size in bits of an acceptable group
uint32    n, preferred size in bits of the group the server
          should send
uint32    max, maximal size in bits of an acceptable group
```

The server responds with:

```
byte      SSH_MSG_KEXGSS_GROUP
mpint     p, safe prime
mpint     g, generator for subgroup in GF(p)
```

This is followed by the message exchange described above in Section 2.1, except that the exchange hash H is computed as the HASH hash of the concatenation of the following:

```
string    V_C, the client's version string (CR, NL excluded)
string    V_S, the server's version string (CR, NL 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
uint32    min, minimal size in bits of an acceptable group
uint32    n, preferred size in bits of the group the server
          should send
uint32    max, maximal size in bits of an acceptable group
mpint     p, safe prime
mpint     g, generator for subgroup in GF(p)
mpint     e, exchange value sent by the client
mpint     f, exchange value sent by the server
mpint     K, the shared secret
```

2.3. gss-group1-sha1-*

Each of these methods specifies GSS-API-authenticated Diffie-Hellman key exchange as described in Section 2.1 with SHA-1 as HASH, and the group defined in Section 8.1 of [SSH-TRANSPORT]. The method name for each method is the concatenation of the string "gss-group1-sha1-" with the Base64 encoding of the MD5 hash [MD5] of the ASN.1 Distinguished Encoding Rules (DER) encoding [ASN1] of the underlying GSS-API mechanism's Object Identifier (OID). Base64 encoding is described in Section 6.8 of [MIME].

Each and every such key exchange method is implicitly registered by this specification. The IESG is considered to be the owner of all such key exchange methods; this does NOT imply that the IESG is considered to be the owner of the underlying GSS-API mechanism.

2.4. gss-group14-sha1-*

Each of these methods specifies GSS-API authenticated Diffie-Hellman key exchange as described in Section 2.1 with SHA-1 as HASH, and the group defined in Section 8.2 of [SSH-TRANSPORT]. The method name for each method is the concatenation of the string "gss-group14-sha1-" with the Base64 encoding of the MD5 hash [MD5] of the ASN.1 DER encoding [ASN1] of the underlying GSS-API mechanism's OID. Base64 encoding is described in Section 6.8 of [MIME].

Each and every such key exchange method is implicitly registered by this specification. The IESG is considered to be the owner of all such key exchange methods; this does NOT imply that the IESG is considered to be the owner of the underlying GSS-API mechanism.

2.5. gss-gex-sha1-*

Each of these methods specifies GSS-API-authenticated Diffie-Hellman key exchange as described in Section 2.2 with SHA-1 as HASH. The method name for each method is the concatenation of the string "gss-gex-sha1-" with the Base64 encoding of the MD5 hash [MD5] of the ASN.1 DER encoding [ASN1] of the underlying GSS-API mechanism's OID. Base64 encoding is described in Section 6.8 of [MIME].

Each and every such key exchange method is implicitly registered by this specification. The IESG is considered to be the owner of all such key exchange methods; this does NOT imply that the IESG is considered to be the owner of the underlying GSS-API mechanism.

2.6. Other GSS-API Key Exchange Methods

Key exchange method names starting with "gss-" are reserved for key exchange methods that conform to this document; in particular, for those methods that use the GSS-API-authenticated Diffie-Hellman key exchange algorithm described in Section 2.1, including any future methods that use different groups and/or hash functions. The intent is that the names for any such future methods be defined in a similar manner to that used in Section 2.3.

3. GSS-API User Authentication

This section describes a general-purpose user authentication method based on [GSSAPI]. It is intended to be run over the SSH user authentication protocol [SSH-USERAUTH].

The authentication method name for this protocol is "gssapi-with-mic".

3.1. GSS-API Authentication Overview

GSS-API authentication must maintain a context. Authentication begins when the client sends an SSH_MSG_USERAUTH_REQUEST, which specifies the mechanism OIDs the client supports.

If the server supports any of the requested mechanism OIDs, the server sends an SSH_MSG_USERAUTH_GSSAPI_RESPONSE message containing the mechanism OID.

After the client receives SSH_MSG_USERAUTH_GSSAPI_RESPONSE, the client and server exchange SSH_MSG_USERAUTH_GSSAPI_TOKEN packets until the authentication mechanism either succeeds or fails.

If at any time during the exchange the client sends a new SSH_MSG_USERAUTH_REQUEST packet, the GSS-API context is completely discarded and destroyed, and any further GSS-API authentication MUST restart from the beginning.

If the authentication succeeds and a non-empty user name is presented by the client, the SSH server implementation verifies that the user name is authorized based on the credentials exchanged in the GSS-API exchange. If the user name is not authorized, then the authentication MUST fail.

3.2. Initiating GSS-API Authentication

The GSS-API authentication method is initiated when the client sends an SSH_MSG_USERAUTH_REQUEST:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name (in ISO-10646 UTF-8 encoding)
string    service name (in US-ASCII)
string    "gssapi-with-mic" (US-ASCII method name)
uint32    n, the number of mechanism OIDs client supports
string[n] mechanism OIDs
```

Mechanism OIDs are encoded according to the ASN.1 Distinguished Encoding Rules (DER), as described in [ASN1] and in Section 3.1 of

[GSSAPI]. The mechanism OIDs MUST be listed in order of preference, and the server must choose the first mechanism OID on the list that it supports.

The client SHOULD send GSS-API mechanism OIDs only for mechanisms that are of the same priority, compared to non-GSS-API authentication methods. Otherwise, authentication methods may be executed out of order. Thus, the client could first send an SSH_MSG_USERAUTH_REQUEST for one GSS-API mechanism, then try public key authentication, and then try another GSS-API mechanism.

If the server does not support any of the specified OIDs, the server MUST fail the request by sending an SSH_MSG_USERAUTH_FAILURE packet.

The user name may be an empty string if it can be deduced from the results of the GSS-API authentication. If the user name is not empty, and the requested user does not exist, the server MAY disconnect or MAY send a bogus list of acceptable authentications but never accept any. This makes it possible for the server to avoid disclosing information about which accounts exist. In any case, if the user does not exist, the authentication request MUST NOT be accepted.

Note that the 'user name' value is encoded in ISO-10646 UTF-8. It is up to the server how it interprets the user name and determines whether the client is authorized based on his GSS-API credentials. In particular, the encoding used by the system for user names is a matter for the ssh server implementation. However, if the client reads the user name in some other encoding (e.g., ISO 8859-1 - ISO Latin1), it MUST convert the user name to ISO-10646 UTF-8 before transmitting, and the server MUST convert the user name to the encoding used on that system for user names.

Any normalization or other preparation of names is done by the ssh server based on the requirements of the system, and is outside the scope of SSH. SSH implementations which maintain private user databases SHOULD prepare user names as described by [SASLPREP].

The client MAY at any time continue with a new SSH_MSG_USERAUTH_REQUEST message, in which case the server MUST abandon the previous authentication attempt and continue with the new one.

3.3. Initial Server Response

The server responds to the SSH_MSG_USERAUTH_REQUEST with either an SSH_MSG_USERAUTH_FAILURE if none of the mechanisms are supported or with an SSH_MSG_USERAUTH_GSSAPI_RESPONSE as follows:

```
byte        SSH_MSG_USERAUTH_GSSAPI_RESPONSE
string      selected mechanism OID
```

The mechanism OID must be one of the OIDs sent by the client in the SSH_MSG_USERAUTH_REQUEST packet.

3.4. GSS-API Session

Once the mechanism OID has been selected, the client will then initiate an exchange of one or more pairs of SSH_MSG_USERAUTH_GSSAPI_TOKEN packets. These packets contain the tokens produced from the 'GSS_Init_sec_context()' and 'GSS_Accept_sec_context()' calls. The actual number of packets exchanged is determined by the underlying GSS-API mechanism.

```
byte        SSH_MSG_USERAUTH_GSSAPI_TOKEN
string      data returned from either GSS_Init_sec_context()
            or GSS_Accept_sec_context()
```

If an error occurs during this exchange on server side, the server can terminate the method by sending an SSH_MSG_USERAUTH_FAILURE packet. If an error occurs on client side, the client can terminate the method by sending a new SSH_MSG_USERAUTH_REQUEST packet.

When calling GSS_Init_sec_context(), the client MUST set integ_req_flag to "true" to request that per-message integrity protection be supported for this context. In addition, deleg_req_flag MAY be set to "true" to request access delegation, if requested by the user.

Since the user authentication process by its nature authenticates only the client, the setting of mutual_req_flag is not needed for this process. This flag SHOULD be set to "false".

Since the user authentication process will involve the exchange of only a single token once the context has been established, it is not necessary that the context support detection of replayed or out-of-sequence tokens. Thus, the setting of replay_det_req_flag and sequence_req_flag are not needed for this process. These flags SHOULD be set to "false".

Additional SSH_MSG_USERAUTH_GSSAPI_TOKEN messages are sent if and only if the calls to the GSS-API routines produce send tokens of non-zero length.

Any major status code other than GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED SHOULD be a failure.

3.5. Binding Encryption Keys

In some cases, it is possible to obtain improved security by allowing access only if the client sends a valid message integrity code (MIC) binding the GSS-API context to the keys used for encryption and integrity protection of the SSH session. With this extra level of protection, a "man-in-the-middle" attacker who has convinced a client of his authenticity cannot then relay user authentication messages between the real client and server, thus gaining access to the real server. This additional protection is available when the negotiated GSS-API context supports per-message integrity protection, as indicated by the setting of the integ_avail flag on successful return from GSS_Init_sec_context() or GSS_Accept_sec_context().

When the client's call to GSS_Init_sec_context() returns GSS_S_COMPLETE with the integ_avail flag set, the client MUST conclude the user authentication exchange by sending the following message:

```
byte      SSH_MSG_USERAUTH_GSSAPI_MIC
string    MIC
```

This message MUST be sent only if GSS_Init_sec_context() returned GSS_S_COMPLETE. If a token is also returned, then the SSH_MSG_USERAUTH_GSSAPI_TOKEN message MUST be sent before this one.

The contents of the MIC field are obtained by calling GSS_GetMIC() over the following, using the GSS-API context that was just established:

```
string    session identifier
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service
string    "gssapi-with-mic"
```

If this message is received by the server before the GSS-API context is fully established, the server MUST fail the authentication.

If this message is received by the server when the negotiated GSS-API context does not support per-message integrity protection, the server MUST fail the authentication.

3.6. Client Acknowledgement

Some servers may wish to permit user authentication to proceed even when the negotiated GSS-API context does not support per-message integrity protection. In such cases, it is possible for the server

to successfully complete the GSS-API method, while the client's last call to GSS_Init_sec_context() fails. If the server simply assumed success on the part of the client and completed the authentication service, it is possible that the client would fail to complete the authentication method, but not be able to retry other methods because the server had already moved on. To protect against this, a final message is sent by the client to indicate it has completed authentication.

When the client's call to GSS_Init_sec_context() returns GSS_S_COMPLETE with the integ_avail flag not set, the client MUST conclude the user authentication exchange by sending the following message:

```
byte      SSH_MSG_USERAUTH_GSSAPI_EXCHANGE_COMPLETE
```

This message MUST be sent only if GSS_Init_sec_context() returned GSS_S_COMPLETE. If a token is also returned, then the SSH_MSG_USERAUTH_GSSAPI_TOKEN message MUST be sent before this one.

If this message is received by the server before the GSS-API context is fully established, the server MUST fail the authentication.

If this message is received by the server when the negotiated GSS-API context supports per-message integrity protection, the server MUST fail the authentication.

It is a site policy decision for the server whether or not to permit authentication using GSS-API mechanisms and/or contexts that do not support per-message integrity protection. The server MAY fail the otherwise valid gssapi-with-mic authentication if per-message integrity protection is not supported.

3.7. Completion

As with all SSH authentication methods, successful completion is indicated by an SSH_MSG_USERAUTH_SUCCESS if no other authentication is required, or an SSH_MSG_USERAUTH_FAILURE with the partial success flag set if the server requires further authentication. This packet SHOULD be sent immediately following receipt of the SSH_MSG_USERAUTH_GSSAPI_EXCHANGE_COMPLETE packet.

3.8. Error Status

In the event that a GSS-API error occurs on the server during context establishment, the server MAY send the following message to inform the client of the details of the error before sending an SSH_MSG_USERAUTH_FAILURE message:

```
byte      SSH_MSG_USERAUTH_GSSAPI_ERROR
uint32    major_status
uint32    minor_status
string    message
string    language tag
```

The message text MUST be encoded in the UTF-8 encoding described in [UTF8]. Language tags are those described in [LANGTAG]. Note that the message text may contain multiple lines separated by carriage return-line feed (CRLF) sequences. Application developers should take this into account when displaying these messages.

Clients receiving this message MAY log the error details and/or report them to the user. Any server sending this message MUST ignore any SSH_MSG_UNIMPLEMENTED sent by the client in response.

3.9. Error Token

In the event that, during context establishment, a client's call to GSS_Init_sec_context() or a server's call to GSS_Accept_sec_context() returns a token along with an error status, the resulting "error token" SHOULD be sent to the peer using the following message:

```
byte        SSH_MSG_USERAUTH_GSSAPI_ERRTOK
string      error token
```

This message implies that the authentication is about to fail, and is defined to allow the error token to be communicated without losing synchronization.

When a server sends this message, it MUST be followed by an SSH_MSG_USERAUTH_FAILURE message, which is to be interpreted as applying to the same authentication request. A client receiving this message SHOULD wait for the following SSH_MSG_USERAUTH_FAILURE message before beginning another authentication attempt.

When a client sends this message, it MUST be followed by a new authentication request or by terminating the connection. A server receiving this message MUST NOT send an SSH_MSG_USERAUTH_FAILURE in reply, since such a message might otherwise be interpreted by a client as a response to the following authentication sequence.

Any server sending this message MUST ignore any SSH_MSG_UNIMPLEMENTED sent by the client in response. If a server sends both this message and an SSH_MSG_USERAUTH_GSSAPI_ERROR message, the SSH_MSG_USERAUTH_GSSAPI_ERROR message MUST be sent first, to allow the client to store and/or display the error status before processing the error token.

4. Authentication Using GSS-API Key Exchange

This section describes a user authentication method building on the framework described in [SSH-USERAUTH]. This method performs user authentication by making use of an existing GSS-API context established during key exchange.

The authentication method name for this protocol is "gssapi-keyex".

This method may be used only if the initial key exchange was performed using a GSS-API-based key exchange method defined in accordance with Section 2. The GSS-API context used with this method is always that established during an initial GSS-API-based key exchange. Any context established during key exchange for the purpose of rekeying MUST NOT be used with this method.

The server SHOULD include this user authentication method in the list of methods that can continue (in an SSH_MSG_USERAUTH_FAILURE) if the initial key exchange was performed using a GSS-API-based key exchange method and provides information about the user's identity that is useful to the server. It MUST NOT include this method if the initial key exchange was not performed using a GSS-API-based key exchange method defined in accordance with Section 2.

The client SHOULD attempt to use this method if it is advertised by the server, initial key exchange was performed using a GSS-API-based key exchange method, and this method has not already been tried. The client SHOULD NOT try this method more than once per session. It MUST NOT try this method if initial key exchange was not performed using a GSS-API-based key exchange method defined in accordance with Section 2.

If a server receives a request for this method when initial key exchange was not performed using a GSS-API-based key exchange method defined in accordance with Section 2, it MUST return SSH_MSG_USERAUTH_FAILURE.

This method is defined as a single message:

```
byte        SSH_MSG_USERAUTH_REQUEST
string      user name
string      service
string      "gssapi-keyex"
string      MIC
```

The contents of the MIC field are obtained by calling GSS_GetMIC over the following, using the GSS-API context that was established during initial key exchange:

```
string      session identifier
byte        SSH_MSG_USERAUTH_REQUEST
string      user name
string      service
string      "gssapi-keyex"
```

Upon receiving this message when initial key exchange was performed using a GSS-API-based key exchange method, the server uses GSS_VerifyMIC() to verify that the MIC received is valid. If the MIC is not valid, the user authentication fails, and the server MUST return SSH_MSG_USERAUTH_FAILURE.

If the MIC is valid and the server is satisfied as to the user's credentials, it MAY return either SSH_MSG_USERAUTH_SUCCESS or SSH_MSG_USERAUTH_FAILURE with the partial success flag set, depending on whether additional authentications are needed.

5. Null Host Key Algorithm

The "null" host key algorithm has no associated host key material and provides neither signature nor encryption algorithms. Thus, it can be used only with key exchange methods that do not require any public-key operations and do not require the use of host public key material. The key exchange methods described in Section 2 are examples of such methods.

This algorithm is used when, as a matter of configuration, the host does not have or does not wish to use a public key. For example, it can be used when the administrator has decided as a matter of policy to require that all key exchanges be authenticated using Kerberos [KRB5], and thus the only permitted key exchange method is the GSS-API-authenticated Diffie-Hellman exchange described above, with Kerberos V5 as the underlying GSS-API mechanism. In such a configuration, the server implementation supports the "ssh-dss" key algorithm (as required by [SSH-TRANSPORT]), but could be prohibited by configuration from using it. In this situation, the server needs some key exchange algorithm to advertise; the "null" algorithm fills this purpose.

Note that the use of the "null" algorithm in this way means that the server will not be able to interoperate with clients that do not support this algorithm. This is not a significant problem, since in the configuration described, it will also be unable to interoperate with implementations that do not support the GSS-API-authenticated key exchange and Kerberos.

Any implementation supporting at least one key exchange method that conforms to Section 2 MUST also support the "null" host key algorithm. Servers MUST NOT advertise the "null" host key algorithm unless it is the only algorithm advertised.

6. Summary of Message Numbers

The following message numbers have been defined for use with GSS-API-based key exchange methods:

```
#define SSH_MSG_KEXGSS_INIT                      30
#define SSH_MSG_KEXGSS_CONTINUE                  31
#define SSH_MSG_KEXGSS_COMPLETE                  32
#define SSH_MSG_KEXGSS_HOSTKEY                   33
#define SSH_MSG_KEXGSS_ERROR                     34
#define SSH_MSG_KEXGSS_GROUPREQ                  40
#define SSH_MSG_KEXGSS_GROUP                     41
```

The numbers 30-49 are specific to key exchange and may be redefined by other kex methods.

The following message numbers have been defined for use with the 'gssapi-with-mic' user authentication method:

```
#define SSH_MSG_USERAUTH_GSSAPI_RESPONSE          60
#define SSH_MSG_USERAUTH_GSSAPI_TOKEN             61
#define SSH_MSG_USERAUTH_GSSAPI_EXCHANGE_COMPLETE 63
#define SSH_MSG_USERAUTH_GSSAPI_ERROR             64
#define SSH_MSG_USERAUTH_GSSAPI_ERRTOK            65
#define SSH_MSG_USERAUTH_GSSAPI_MIC               66
```

The numbers 60-79 are specific to user authentication and may be redefined by other user auth methods. Note that in the method described in this document, message number 62 is unused.

7. GSS-API Considerations

7.1. Naming Conventions

In order to establish a GSS-API security context, the SSH client needs to determine the appropriate targ_name to use in identifying the server when calling GSS_Init_sec_context(). For this purpose, the GSS-API mechanism-independent name form for host-based services is used, as described in Section 4.1 of [GSSAPI].

In particular, the targ_name to pass to GSS_Init_sec_context() is obtained by calling GSS_Import_name() with an input_name_type of GSS_C_NT_HOSTBASED_SERVICE, and an input_name_string consisting of the string "host@" concatenated with the hostname of the SSH server.

Because the GSS-API mechanism uses the targ_name to authenticate the server's identity, it is important that it be determined in a secure fashion. One common way to do this is to construct the targ_name from the hostname as typed by the user; unfortunately, because some GSS-API mechanisms do not canonicalize hostnames, it is likely that this technique will fail if the user has not typed a fully-qualified, canonical hostname. Thus, implementers may wish to use other methods, but should take care to ensure they are secure. For example, one should not rely on an unprotected DNS record to map a host alias to the primary name of a server, or an IP address to a hostname, since an attacker can modify the mapping and impersonate the server.

Implementations of mechanisms conforming to this document MUST NOT use the results of insecure DNS queries to construct the targ_name. Clients MAY make use of a mapping provided by local configuration or use other secure means to determine the targ_name to be used. If a client system is unable to securely determine which targ_name to use, then it SHOULD NOT use this mechanism.

7.2. Channel Bindings

This document recommends that channel bindings SHOULD NOT be specified in the calls during context establishment. This document does not specify any standard data to be used as channel bindings, and the use of network addresses as channel bindings may break SSH in environments where it is most useful.

7.3. SPNEGO

The use of the Simple and Protected GSS-API Negotiation Mechanism [SPNEGO] in conjunction with the authentication and key exchange methods described in this document is both unnecessary and undesirable. As a result, mechanisms conforming to this document MUST NOT use SPNEGO as the underlying GSS-API mechanism.

Since SSH performs its own negotiation of authentication and key exchange methods, the negotiation capability of SPNEGO alone does not provide any added benefit. In fact, as described below, it has the potential to result in the use of a weaker method than desired.

Normally, SPNEGO provides the added benefit of protecting the GSS-API mechanism negotiation. It does this by having the server compute a MIC of the list of mechanisms proposed by the client, and then checking that value at the client. In the case of key exchange, this protection is not needed because the key exchange methods described here already perform an equivalent operation; namely, they generate a MIC of the SSH exchange hash, which is a hash of several items including the lists of key exchange mechanisms supported by both sides. In the case of user authentication, the protection is not needed because the negotiation occurs over a secure channel, and the host's identity has already been proved to the user.

The use of SPNEGO combined with GSS-API mechanisms used without SPNEGO can lead to interoperability problems. For example, a client that supports key exchange using the Kerberos V5 GSS-API mechanism [KRB5-GSS] only underneath SPNEGO will not interoperate with a server that supports key exchange only using the Kerberos V5 GSS-API mechanism directly. As a result, allowing GSS-API mechanisms to be used both with and without SPNEGO is undesirable.

If a client's policy is to first prefer GSS-API-based key exchange method X, then non-GSS-API method Y, then GSS-API-based method Z, and if a server supports mechanisms Y and Z but not X, then an attempt to use SPNEGO to negotiate a GSS-API mechanism might result in the use of method Z when method Y would have been preferable. As a result, the use of SPNEGO could result in the subversion of the negotiation algorithm for key exchange methods as described in Section 7.1 of [SSH-TRANSPORT] and/or the negotiation algorithm for user authentication methods as described in [SSH-USERAUTH].

8. IANA Considerations

Consistent with Section 8 of [SSH-ARCH] and Section 4.6 of [SSH-NUMBERS], this document makes the following registrations:

The family of SSH key exchange method names beginning with "gss-group1-sha1-" and not containing the at-sign ('@'), to name the key exchange methods defined in Section 2.3.

The family of SSH key exchange method names beginning with "gss-gex-sha1-" and not containing the at-sign ('@'), to name the key exchange methods defined in Section 2.5.

All other SSH key exchange method names beginning with "gss-" and not containing the at-sign ('@'), to be reserved for future key exchange methods defined in conformance with this document, as noted in Section 2.6.

The SSH host public key algorithm name "null", to name the NULL host key algorithm defined in Section 5.

The SSH user authentication method name "gssapi-with-mic", to name the GSS-API user authentication method defined in Section 3.

The SSH user authentication method name "gssapi-keyex", to name the GSS-API user authentication method defined in Section 4.

The SSH user authentication method name "gssapi" is to be reserved, in order to avoid conflicts with implementations supporting an earlier version of this specification.

The SSH user authentication method name "external-keyx" is to be reserved, in order to avoid conflicts with implementations supporting an earlier version of this specification.

This document creates no new registries.

9. Security Considerations

This document describes authentication and key-exchange protocols. As such, security considerations are discussed throughout.

This protocol depends on the SSH protocol itself, the GSS-API, any underlying GSS-API mechanisms that are used, and any protocols on which such mechanisms might depend. Each of these components plays a part in the security of the resulting connection, and each will have its own security considerations.

The key exchange method described in Section 2 depends on the underlying GSS-API mechanism to provide both mutual authentication and per-message integrity services. If either of these features is not supported by a particular GSS-API mechanism, or by a particular implementation of a GSS-API mechanism, then the key exchange is not secure and MUST fail.

In order for the "external-keyx" user authentication method to be used, it MUST have access to user authentication information obtained as a side-effect of the key exchange. If this information is unavailable, the authentication MUST fail.

Revealing information about the reason for an authentication failure may be considered by some sites to be an unacceptable security risk for a production environment. However, having that information available can be invaluable for debugging purposes. Thus, it is RECOMMENDED that implementations provide a means for controlling, as a matter of policy, whether to send SSH_MSG_USERAUTH_GSSAPI_ERROR, SSH_MSG_USERAUTH_GSSAPI_ERRTOK, and SSH_MSG_KEXGSS_ERROR messages, and SSH_MSG_KEXGSS_CONTINUE messages containing a GSS-API error token.

10. Acknowledgements

The authors would like to thank the following individuals for their invaluable assistance and contributions to this document:

- Sam Hartman
- Love Hornquist-Astrand
- Joel N. Weber II
- Simon Wilkinson
- Nicolas Williams

Much of the text describing DH group exchange was borrowed from [GROUP-EXCHANGE], by Markus Friedl, Niels Provos, and William A. Simpson.

11. References

11.1. Normative References

[ASN1] ISO/IEC, "ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690 (1997), ISO/IEC 8825-1:1998, November 1998.

[GROUP-EXCHANGE] Friedl, M., Provos, N., and W. Simpson, "Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol", RFC 4419, March 2006.

[GSSAPI] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

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

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

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

[MIME] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

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

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

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

[SSH-TRANSPORT] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

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

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

11.2. Informative References

[KRB5] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[KRB5-GSS] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, July 2005.

[SASLPREP] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.

[SPNEGO] Zhu, L., Leach, P., Jaganathan, K., and W. Ingersoll, "The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism", RFC 4178, October 2005.

Authors' Addresses

Jeffrey Hutzelman
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
US

Phone: +1 412 268 7225
EMail: jhutz+@cmu.edu
URI: http://www.cs.cmu.edu/~jhutz/

Joseph Salowey
Cisco Systems
2901 Third Avenue
Seattle, WA 98121
US

Phone: +1 206 256 3380
EMail: jsalowey@cisco.com

Joseph Galbraith
Van Dyke Technologies, Inc.
4848 Tramway Ridge Dr. NE
Suite 101
Albuquerque, NM 87111
US

EMail: galb@vandyke.com

Von Welch
University of Chicago & Argonne National Laboratory
Distributed Systems Laboratory
701 E. Washington
Urbana, IL 61801
US

EMail: welch@mcs.anl.gov

Full Copyright Statement

Intellectual Property

Acknowledgement

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