Internet Engineering Task Force
Internet-Draft
Updates: 4462 (if approved)
Intended status: Standards Track
Expires: April 3, 2021

H. Kario
Red Hat, Inc.
Sep 30, 2020

Quantum-Resistant GSS-API Key Exchange for SSH
draft-kario-gss-qr-kex-00

Abstract

This document specifies additions and amendments to RFC4462. It defines a new key exchange method that uses GSS-API in a way to provide key exchange method that is resistant to attacks by quantum computers. The purpose of this specification is to provide an easy-to-implement upgrade to environments that require resistance against quantum computers before widely accepted post-quantum cryptography algorithms are established.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

SSH GSS-API Methods [RFC4462] allows the use of GSSAPI for authentication and key exchange in SSH. Unfortunately for resistance against quantum computers all of the methods in RFC 4462 as well as all of the new methods introduced in SSH GSS-API SHA-2 Methods [RFC8732] derive their security from Finite-Field Diffie-Hellman or Elliptic Curve Diffie-Hellman key exchanges. Both FFDH and ECDH are believed to be vulnerable to Shor's algorithm running on quantum computers. This document updates RFC4462 with new methods intended for use in environments where use of quantum resistant algorithms is more important that the forward secrecy provided by FFDH and ECDH.

2. Rationale

Due to security concerns with FFDH and ECDH against attacks using quantum computers, we propose a new key exchange method that does not use FFDH or ECDH to agree on a shared secret to derive later encryption keys but rather uses GSS-API as a secure communication channel to exchange secrets that are then used to derive encryption keys.

To provide resistance against quantum computer attacks the connection needs to also carefully select encryption ciphers, and host authentication methods.

3. Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 RFC2119 [RFC2119] RFC8174 [RFC8174] when, and only when, they appear in all capitals, as shown here.

4. New Quantum Resistant Key Exchange Methods

This document adopts the same naming convention as defined in [RFC4462] to define families of methods that cover any GSS-API mechanism used with a specific SHA-2 Hash. It also reuses much of the the scheme defined in Section 2.1 of [RFC4462].

The following new key exchange algorithms are defined:

Key Exchange Method Name	Implementation Recommendations
gss-qr-sha256-*	SHOULD/RECOMMENDED
gss-qr-sha512-*	MAY/OPTIONAL

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

Each method in any family of methods specifies GSS-API-authenticated exchanges as described in Section 2.1 of [RFC4462]. The method name for each method is the concatenation of the family name prefix with the Base64 encoding of the MD5 hash [RFC1321] of the ASN.1 DER encoding [ISO-IEC-8825-1] of the underlying GSS-API mechanism's OID. Base64 encoding is described in Section 6.8 of [RFC2045].

Family method references

Family Name prefix	Hash Function
gss-qr-sha256-	SHA-256
gss-qr-sha512-	SHA-512

4.1. Generic Quantum Resistant GSS-API key Exchange

This section reuses much of the scheme defined in Section 2.1 of [RFC4462] though it does not transport FFDH key shares in the exchanged messages.

This section defers to [RFC7546] as the source of information on GSS-API context establishment operations, Section 3 being the most relevant. All security considerations described in [RFC7546] apply here too.

The parties generate nonces in the key exchange. The generated nonces MUST be at least 256 bits long and come from a quantum safe CSPRNG. The nonces MUST NOT be reused in other key exchanges.

The client initiates negotiation by calling GSS_Init_sec_context() and the server responds to it by calling GSS_Accept_sec_context(). For the negotiation, client MUST set the mutual_req_flag, conf_req_flag, and integ_req_flag flag to "true". 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 of [RFC4462], 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. This key exchange process will exchange only a single message token once the context has been established; therefore, the replay_det_req_flag and sequence_req_flag SHOULD be set to "false".

During GSS context establishment, multiple tokens may be exchanged by the client and the server. When the GSS context is established (major_status is GSS_S_COMPLETE), the parties check that mutual_state and integ_avail are both "true". If not, the key exchange MUST fail.

To verify the integrity of the handshake both peers use the Hash Function defined by the selected Key Exchange method to calculate the running hash of exchanged messages, H_S and H_C.

```
H_S = hash(V_C || V_S || I_C || KC_S || ... || KC_C).

H_C = hash(V_C || V_S || I_C || KC_S || ... || KC_C || KC).
```

The GSS_wrap() call is used by the server and client to encrypt the calculated hash and the selected nonce. The peers use the

GSS_unwrap() to decrypt the value used to check if the other peer has received the same messages and to get the nonce it selected.

Peers MUST verify if the length of the selected nonce is not shorter than 32 octets. If the received nonce is shorter, the key exchange MUST fail.

The following is an overview of the key exchange process:

```
     Client                                                Server
     ------                                                ------
     Calls GSS_Init_sec_context().
     SSH_MSG_KEXGSS_INIT  --------------->

 (Loop)
 |                                 Calls GSS_Accept_sec_context().
 |                           <------------ SSH_MSG_KEXGSS_CONTINUE
 |   Calls GSS_Init_sec_context().
 |   SSH_MSG_KEXGSS_CONTINUE ------------>

                                   Calls GSS_Accept_sec_context().
                                        Generates ephemeral nonce.
                                                Computes hash H_S.
                                 Calls GSS_wrap( H_S || nonce_S ).
                             <------------ SSH_MSG_KEXGSS_COMPLETE

     Computes hash H_S.
     Calls GSS_unwrap().
     Verifies that computed H_S matches received value.
     Computes hash H_C.
     Generates ephemeral nonce.
     Calls GSS_wrap( H_C || nonce_C ).
     SSH_MSG_KEXGSS_COMPLETE ------------>
                                                Computes hash H_C.
                                               Calls GSS_unwrap().
                  Verifies that computed H_C matches received value.
```

This is implemented with the following messages:

The client sends:

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

The server sends:

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

Each time the client receives the message described above, it makes another call to GSS_Init_sec_context().

The client sends:

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

The final server message is either:

```
byte      SSH_MSG_KEXGSS_COMPLETE
string    enc_nonce (GSS_wrap() of H_S and nonce_S)
boolean   TRUE
string    output_token (from GSS_Accept_sec_context())
```

Or the following if no output_token is available:

```
byte      SSH_MSG_KEXGSS_COMPLETE
string    enc_nonce (GSS_wrap() of H_S and nonce_S)
boolean   FALSE
```

As the final message the client sends either:

```
byte      SSH_MSG_KEXGSS_COMPLETE
string    enc_nonce (GSS_wrap() of H_C and nonce_C)
boolean   TRUE
string    output_token (from GSS_Accept_sec_context())
```

Or the following if no output_token is available:

```
byte      SSH_MSG_KEXGSS_COMPLETE
string    enc_nonce (GSS_wrap() of H_C and nonce_C)
boolean   FALSE
```

The hashes H_S and H_C are 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, server's version string (CR, NL excluded)
string    I_C, payload of the client's SSH_MSG_KEXINIT
string    I_S, payload of the server's SSH_MSG_KEXINIT
string    KC_S, payload of the server's SSH_MSG_KEXGSS_CONTINUE
string    KC_C, payload of the client's SSH_MSG_KEXGSS_CONTINUE
string    KC_S, payload of the server's second SSH_MSG_KEXGSS_CONTINUE
string    KC_C, payload of the client's second SSH_MSG_KEXGSS_CONTINUE
...
string    KC, payload of the server's SSH_MSG_KEXGSS_COMPLETE
```

Those values are called exchange hashes, and they are used to authenticate the key exchange. The exchange hashes SHOULD be kept secret. If no SSH_MSG_KEXGSS_CONTINUE messages have been sent by the server or received by the client, then an empty string is used in place of KC_S and KC_C when computing the exchange hash. When multiple SSH_MSG_KEXGSS_CONTINUE messages have been sent by either side, then they should all be included in the exchange hash, in order they have been processed by both sides of the connection. For the H_S hash, the KC is an empty string.

Once a party has both the server nonce (nonce_S) and the client nonce (nonce_C) it concatenates them, in this order, to compute the used shared secret K:

K = nonce_S || nonce_C

If the client receives a SSH_MSG_KEXGSS_CONTINUE 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 client receives a SSH_MSG_KEXGSS_COMPLETE message and a call to GSS_Init_sec_context() does not result in a major_status code of GSS_S_COMPLETE, a protocol error has occurred and the key exchange MUST fail.

5. IANA Considerations

This document augments the SSH Key Exchange Method Names in [RFC4462].

IANA is requested to update the SSH Protocol Parameters [IANA-KEX-NAMES] registry with the following entries:

Key Exchange Method Name	Reference
gss-qr-sha256-*	This draft
gss-qr-sha512-*	This draft

6. Security Considerations

6.1. Symmetric cipher security

Current understanding of quantum computer capabilities suggest that symmetric ciphers with keys smaller than 256 bits will require less than the current recommended minimal work factor of 2^128 operations.

As such, connections that use this key exchange methods MUST use ciphers with at least 256 bit keys to retain quantum resistance.

6.2. User authentication

For the connection to remain resistant against quantum computers, the user authentication needs to also use quantum resistant algorithms. In particular, it's RECOMMENDED that connections use gssapi-keyex for client authentication. The publickey mechanism MUST NOT be used unless the asymmetric keys used for it use post-quantum algorithms. DSA, ECDSA, and RSA keys MUST NOT be used.

6.3. Used GSSAPI Mechanisms

The security of the key exchange depends on the security of the used GSSAPI mechanism. The described key exchange will be quantum resistant only in case the used GSSAPI mechanism is quantum resistant.

For example, the Kerberos 5 mechanism is quantum resistant only when it's used together with algorithms and key sizes that are quantum resistant. Quantum safe algorithm SHOULD be used throught the kerberos infrastructure, both for authentication and encryption. Currently aes256-cts-hmac-sha384-192 mechanism defined in [RFC8009] for encryption is an example of such an algorithm.

6.4. GSSAPI Delegation

Some GSSAPI mechanisms can act on a request to delegate credentials to the target host when the deleg_req_flag is set. In this case, extra care must be taken to ensure that the acceptor being authenticated matches the target the user intended. Some mechanisms implementations (like commonly used krb5 libraries) may use insecure DNS resolution to canonicalize the target name; in these cases spoofing a DNS response that points to an attacker-controlled machine may results in the user silently delegating credentials to the attacker, who can then impersonate the user at will.

7. References

7.1. Normative References

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, DOI 10.17487/RFC1321, April 1992, <https://www.rfc-editor.org/info/rfc1321>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4462] Hutzelman, J., Salowey, J., Galbraith, J., and V. Welch, "Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol", RFC 4462, DOI 10.17487/RFC4462, May 2006, <https://www.rfc-editor.org/info/rfc4462>.

[RFC7546] Kaduk, B., "Structure of the Generic Security Service (GSS) Negotiation Loop", RFC 7546, DOI 10.17487/RFC7546, May 2015, <https://www.rfc-editor.org/info/rfc7546>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8732] Sorce, S. and H. Kario, "Generic Security Service Application Program Interface (GSS-API) Key Exchange with SHA-2", RFC 8732, DOI 10.17487/RFC8732, February 2020, <https://www.rfc-editor.org/info/rfc8732>.

7.2. Informative References

[IANA-KEX-NAMES]
Internet Assigned Numbers Authority, "Secure Shell (SSH) Protocol Parameters: Key Exchange Method Names", June 2005, <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>.

[ISO-IEC-8825-1]
International Organization for Standardization / International Electrotechnical Commission, "ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1, November 2015, <http://standards.iso.org/ittf/PubliclyAvailableStandards/c068345_ISO_IEC_8825-1_2015.zip>.

[NIST-SP-800-131Ar1]
National Institute of Standards and Technology, "Transitions: Recommendation for Transitioning of the Use of Cryptographic Algorithms and Key Lengths", NIST Special Publication 800-131A Revision 1, November 2015, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf>.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <https://www.rfc-editor.org/info/rfc6194>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC8009] Jenkins, M., Peck, M., and K. Burgin, "AES Encryption with HMAC-SHA2 for Kerberos 5", RFC 8009, DOI 10.17487/RFC8009, October 2016, <https://www.rfc-editor.org/info/rfc8009>.

Author's Address

Hubert Kario
Red Hat, Inc.
Purkynova 115
Brno 612 00
Czech Republic

Email: hkario@redhat.com