Network Working Group
Request for Comments: 4419
Category: Standards Track

M. Friedl
N. Provos
W. Simpson

March 2006

Diffie-Hellman Group Exchange for 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

This memo describes a new key exchange method for the Secure Shell (SSH) protocol. It allows the SSH server to propose new groups on which to perform the Diffie-Hellman key exchange to the client. The proposed groups need not be fixed and can change with time.

1. Overview and Rationale

SSH [RFC4251] is a very common protocol for secure remote login on the Internet. Currently, SSH performs the initial key exchange using the "diffie-hellman-group1-sha1" method [RFC4253]. This method prescribes a fixed group on which all operations are performed.

The Diffie-Hellman key exchange provides a shared secret that cannot be determined by either party alone. Furthermore, the shared secret is known only to the participant parties. In SSH, the key exchange is signed with the host key to provide host authentication.

The security of the Diffie-Hellman key exchange is based on the difficulty of solving the Discrete Logarithm Problem (DLP). Since we expect that the SSH protocol will be in use for many years in the future, we fear that extensive precomputation and more efficient algorithms to compute the discrete logarithm over a fixed group might pose a security threat to the SSH protocol.

The ability to propose new groups will reduce the incentive to use precomputation for more efficient calculation of the discrete logarithm. The server can constantly compute new groups in the background.

2. Requirements Notation

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 [RFC2119].

3. Diffie-Hellman Group and Key Exchange

The server keeps a list of safe primes and corresponding generators that it can select from. A prime p is safe if p = 2q + 1 and q is prime. New primes can be generated in the background.

The generator g should be chosen such that the order of the generated subgroup does not factor into small primes; that is, with p = 2q + 1, the order has to be either q or p - 1. If the order is p - 1, then the exponents generate all possible public values, evenly distributed throughout the range of the modulus p, without cycling through a smaller subset. Such a generator is called a "primitive root" (which is trivial to find when p is "safe").

The client requests a modulus from the server indicating the preferred size. In the following description (C is the client, S is the server, the modulus p is a large safe prime, and g is a generator for a subgroup of GF(p), min is the minimal size of p in bits that is acceptable to the client, n is the size of the modulus p in bits that the client would like to receive from the server, max is the maximal size of p in bits that the client can accept, V_S is S's version string, V_C is C's version string, K_S is S's public host key, I_C is C's KEXINIT message, and I_S is S's KEXINIT message that has been exchanged before this part begins):

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. C generates a random number x, where 1 < x < (p-1)/2. It computes e = g^x mod p, and sends "e" to S.

4. S generates a random number y, where 0 < y < (p-1)/2, 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 || min || n || max || p || g || 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.

5. C verifies that K_S really is the host key for S (e.g., using certificates or a local database to obtain the public key). 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 || min || n || max || p || g || e || f || K), and verifies the signature s on H.

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.

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. To prevent confinement attacks, they MUST accept the shared secret K only if 1 < K < p - 1.

The server should return the smallest group it knows that is larger than the size the client requested. If the server does not know a group that is larger than the client request, then it SHOULD return the largest group it knows. In all cases, the size of the returned group SHOULD be at least 1024 bits.

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 KEXINIT messages.

First, the client sends:

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

The server responds with

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

The client responds with:

```
byte    SSH_MSG_KEX_DH_GEX_INIT
mpint   e
```

The server responds with:

```
byte    SSH_MSG_KEX_DH_GEX_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 version string (CR and NL excluded)
string  V_S, the server's version string (CR and 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 will send
uint32  max, maximal size in bits of an acceptable group
mpint   p, safe prime
mpint   g, generator for subgroup
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 as per [RFC4253].

4. Key Exchange Methods

This document defines two new key exchange methods: "diffie-hellman-group-exchange-sha1" and "diffie-hellman-group-exchange-sha256".

4.1. diffie-hellman-group-exchange-sha1

The "diffie-hellman-group-exchange-sha1" method specifies Diffie-Hellman Group and Key Exchange with SHA-1 [FIPS-180-2] as HASH.

4.2. diffie-hellman-group-exchange-sha256

The "diffie-hellman-group-exchange-sha256" method specifies Diffie-Hellman Group and Key Exchange with SHA-256 [FIPS-180-2] as HASH.

Note that the hash used in key exchange (in this case, SHA-256) must also be used in the key derivation pseudo-random function (PRF), as per the requirement in the "Output from Key Exchange" section in [RFC4253].

5. Summary of Message Numbers

The following message numbers have been defined in this document. They are in a name space private to this document and not assigned by IANA.

```
#define SSH_MSG_KEX_DH_GEX_REQUEST_OLD  30
#define SSH_MSG_KEX_DH_GEX_REQUEST      34
#define SSH_MSG_KEX_DH_GEX_GROUP        31
#define SSH_MSG_KEX_DH_GEX_INIT         32
#define SSH_MSG_KEX_DH_GEX_REPLY        33
```

SSH_MSG_KEX_DH_GEX_REQUEST_OLD is used for backward compatibility. Instead of sending "min || n || max", the client only sends "n". In addition, the hash is calculated using only "n" instead of "min || n || max".

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

6. Implementation Notes

6.1. Choice of Generator

One useful technique is to select the generator, and then limit the modulus selection sieve to primes with that generator:

2	when p (mod 24) = 11.
5	when p (mod 10) = 3 or 7.

It is recommended to use 2 as generator, because it improves efficiency in multiplication performance. It is usable even when it is not a primitive root, as it still covers half of the space of possible residues.

6.2. Private Exponents

To increase the speed of the key exchange, both client and server may reduce the size of their private exponents. It should be at least twice as long as the key material that is generated from the shared secret. For more details, see the paper by van Oorschot and Wiener [VAN-OORSCHOT].

7. Security Considerations

This protocol aims to be simple and uses only well-understood primitives. This encourages acceptance by the community and allows for ease of implementation, which hopefully leads to a more secure system.

The use of multiple moduli inhibits a determined attacker from precalculating moduli exchange values, and discourages dedication of resources for analysis of any particular modulus.

It is important to employ only safe primes as moduli, as they allow us to find a generator g so that the order of the generated subgroup does not factor into small primes, that is, with p = 2q + 1, the order has to be either q or p - 1. If the order is p - 1, then the exponents generate all possible public values, evenly distributed throughout the range of the modulus p, without cycling through a smaller subset. Van Oorshot and Wiener note that using short private exponents with a random prime modulus p makes the computation of the discrete logarithm easy [VAN-OORSCHOT]. However, they also state that this problem does not apply to safe primes.

The least significant bit of the private exponent can be recovered when the modulus is a safe prime [MENEZES]. However, this is not a problem if the size of the private exponent is big enough. Related to this, Waldvogel and Massey note: When private exponents are chosen independently and uniformly at random from {0,...,p-2}, the key entropy is less than 2 bits away from the maximum, lg(p-1) [WALDVOGEL].

The security considerations in [RFC4251] also apply to this key exchange method.

8. Acknowledgements

The document is derived in part from "SSH Transport Layer Protocol" [RFC4253] by T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen.

Markku-Juhani Saarinen pointed out that the least significant bit of the private exponent can be recovered efficiently when using safe primes and a subgroup with an order divisible by two.

Bodo Moeller suggested that the server send only one group, reducing the complexity of the implementation and the amount of data that needs to be exchanged between client and server.

Appendix A: Generation of Safe Primes

The "Handbook of Applied Cryptography" [MENEZES] lists the following algorithm to generate a k-bit safe prime p. It has been modified so that 2 is a generator for the multiplicative group mod p.

1. Do the following:
    1. Select a random (k-1)-bit prime q, so that q mod 12 = 5.
    2. Compute p := 2q + 1, and test whether p is prime (using, e.g., trial division and the Rabin-Miller test).
2. Repeat until p is prime.

If an implementation uses the OpenSSL libraries, a group consisting of a 1024-bit safe prime and 2 as generator can be created as follows:

```
DH *d = NULL;
d = DH_generate_parameters(1024, DH_GENERATOR_2, NULL, NULL);
BN_print_fp(stdout, d->p);
```

The order of the subgroup generated by 2 is q = p - 1.

References

Normative References

[FIPS-180-2] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-2, August 2002.

[RFC4251] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[RFC4253] Lonvick, C., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

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

Informative References

[MENEZES] Menezes, A., van Oorschot, P., and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, p. 537, 1996.

[VAN-OORSCHOT]	van Oorschot, P. and M. Wiener, "On Diffie-Hellman key agreement with short exponents", Advances in Cryptology -EUROCRYPT'96, LNCS 1070, Springer-Verlag, pp. 332-343, 1996.
[WALDVOGEL]	Waldvogel, C. and J. Massey, "The probability distribution of the Diffie-Hellman key", Proceedings of AUSCRYPT 92, LNCS 718, Springer-Verlag, pp. 492-504, 1993.

Authors' Addresses

Markus Friedl
EMail: markus@openbsd.org

Niels Provos
EMail: provos@citi.umich.edu

William A. Simpson
EMail: wsimpson@greendragon.com

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).