Using the SRP protocol as a key exchange method in Secure Shell

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at

Copyright Notice

Copyright (C) The Internet Society (2001). See the Full Copyright Notice below for details.

Abstract

This memo describes an experimental method for authentication and keyexchange in the Secure Shell protocol.

The main virtue of the SRP protocol [SRP] is that it provides authentication based on a small secret (typically a password). It is useful in situations where no authentic host key is known. For Secure Shell, it can be used as a bootstrapping procedure to get the host key of a server in a safe way. SRP also provides authentication of the user, which means that it might make sense to either skip the secsh "ssh-userauth"-service [SSH-USERAUTH] when using SRP, or allow login with the "none" or "external-keyx" method.

Conventions and notations
Some of the conventions used in this document are taken from [SSH-TRANS], others are from [SRP].

C is the client, S is the server; q is a large safe prime, g is a primitive root. V_S is S’s version string; V_C is C’s version string; I_C is C’s KEXINIT message and I_S S’s KEXINIT message which have been exchanged before this part begins. (See [SSH-TRANS] for more information).

The ^ operator is the exponentiation operation, and the mod operator is the integer remainder operation. Most implementations perform the exponentiation and remainder in a single stage to avoid generating unwieldy intermediate results.

The | symbol indicates ssh-style string concatenation: For any strings A and B, A | B is the encoding of

    string A
    string B

Computation takes place in the ring \( \mathbb{Z}/q \). Actually, most of the action takes place in its multiplicative group, which is generated by g. The ring structure is not absolutely essential, what we really need is a group \( G \) and and two mixing operations \(+\) and \(-\), unrelated to the group operation, each mapping \( G \times G \) onto a set that is "almost" equal to \( G \) (in the ring case, the image includes zero, which is outside the multiplicative group. This is not really a problem). We must have \( a = (a + b) - b \), for all \( a, b \) in \( G \) such that also \( a + b \) is in \( G \), and this is why it is convenient to use the ring structure.

Furthermore, HASH is a hash function (currently SHA1), n is the user’s name (used for looking up salt and verifier in the server’s database), p is a password, and s is a random salt string.

\( x \) is constructed from the strings \( n, p \) and \( s \) as \( \text{HASH}(s \mid \text{HASH}(n \mid p)) \), and the verifier is computed as \( g^x \mod q \). S keeps a database containing triples \( \langle n, s, v \rangle \), indexed by \( n \).

Protocol description

1. C generates a random number \( a (\log(q) < a < q-1) \) and computes \( e = g^a \mod q \). C sends \( e \) and \( n \) to S.

2. S uses \( n \) to find \( v \) and \( s \) in its database. S generates a random number \( b, (\log(q) < b < q-1) \) and computes \( f = v + g^b \mod q \). S selects \( u \) as the integer corresponding to the first 32 bits of \( \text{HASH}(f) \). If \( f \) or \( u \) happens to be zero, S must try another \( b \). S computes \( K = (e \times v^u)^b \mod q \). S sends \( s \) and \( f \) to C.
3. C gets the password \( p \) and computes \( x = \text{HASH}(s \mid \text{H}(n \mid p)) \) and \( v = g^x \mod q \). C also computes \( u \) in the same way as S. Finally, C computes \( K = (f - v)^{(a + u \times x)} \mod q \).

Each party must check that \( e \) and \( f \) are in the range \([1, q-1]\). If not, the key exchange fails.

Note the addition in step 2, \( v + g^b \mod q \), and the corresponding subtraction \( f - v \) in step 3, are the only operations that uses the ring structure. C should also check that \( f - v \) is non-zero, i.e. belongs to the multiplicative group generated by \( g \).

At this point C and S have a shared secret \( K \). They must now prove that they know the same value. Even if we’re primarily interested in authenticating the server, the user must prove knowledge of the key *first*. (Otherwise, the server leaks information about the verifier).

To do this, the client sends \( m_1 = \text{HMAC}(K, H) \) to the server, where \( H \) is the "exchange hash" defined below. After verifying the MAC, the server responds by sending \( m_2 = \text{HMAC}(K, e \mid m_1 \mid H) \) to the client. Actually, the purpose of this final message exchange is twofold: (i) to prove knowledge of the shared secret key \( K \), completing the SRP protocol, and (ii) to use the shared key \( K \) to authenticate the exchange hash. The latter is needed in order to protect against attacks on the algorithm negotiation that happens before the SRP exchange, as well as version rollback attacks.

Protocol messages

The name of the method, when listed in the SSH_MSG_KEXINIT message, is "srp-ring1-sha1". The SSH_MSG_KEXINIT negotiation determines which hash function is used, as well as the values of \( q \) and \( g \).

For the "srp-ring1-sha1", \( q \) is equal to \( 2^{1024} - 2^{960} - 1 + 2^{64} \times \text{floor}(2^{894} \pi + 129093) \). This is the same prime used for "diffie-hellman-group1-sha1" in [SSH-TRANS]. Its hexadecimal value is

\[
\text{FFFFFFFF FFFFFFFF C90FDAA2 2168C234 4C6662FB 80DC1CD1 29024E08 8A67CC74 02B3139B22 514A0879 8B3404DD EF95193B CD3A431B 302B0A6D F25F1437 4FE1356D 6551C64E 5E2B67ED EE386BFB 5A989FBA5 AE9F2411 7C4B1FE6 49286651 ECE65381 FFFFFFFF FFFFFFFF.}
\]

In decimal, this value is

\( 179769313486231590770839156793787453197860296048756011706444 \)
The generator used for "srp-ring1-ring1" is $g = 5$. This is different from the generator used in [SSH-TRANS], because we need to generate the entire multiplicative group.

First, the client sends:

```
byte      SSH_MSG_KEXSRP_INIT
string    n
mpint     e
```

The server responds with

```
byte      SSH_MSG_KEXSRP_REPLY
string    s
mpint     f
```

The server MUST NOT send this message until it has received the SSH_MSG_KEXSRP_INIT message.

At this point, both sides can compute the exchange hash $H$, as the \text{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    n, the user name
string    s, the salt
mpint     e, exchange value sent by the client
mpint     f, exchange value sent by the server
mpint     K, the shared secret
```

The client computes $m_1 = \text{HMAC}(K, H)$, and sends it to the server, to prove that it knows the shared key. It sends

```
byte      SSH_MSG_KEXSRP_PROOF
string    m_1
```

[ Would it be possible to instead send the exchange hash in the clear, e.g. use $m_1 = H$? ]

The server verifies that $m_1$ is correct using its own $K$. If they don’t
match, the key exchange fails, and the server MUST NOT send any proof back to the client.

Finally, the server computes $m_2$ as the $\text{HMAC}(K, e | m_1 | H)$ and sends

```c
byte SSH_MSG_KEXSRP_PROOF
string m2
```

to the client. The client verifies that $m_2$ is correct, and if so, the key exchange is successful and its output is $H$ and $K$.

### Message numbers

The following message numbers have been defined in this protocol

```c
/* Numbers 30-49 used for kex packets.
   Different kex methods may reuse message numbers in this range. */
define SSH_MSG_KEXSRP_INIT 30
#define SSH_MSG_KEXSRP_REPLY 31
#define SSH_MSG_KEXSRP_PROOF 32
```

### Ring negotiation

This section sketches the changes needed in order to get away from using a fixed ring. The client MUST not use a ring unless its quality is checked in some way (see next section). I will assume that the client either keeps a list of trusted rings, or makes extensive quality checks at runtime. The name of this key exchange method is "srp-sha1".

Each verifier must be associated with a particular ring, which was used when computing the verifier in the first place. Therefore, the server’s user database will contain entries <n, s, v, q, g> where the first three elements are the name, salt and verifier as before, and $q$ and $g$ determines the ring and the generator.

C initiates the protocol by sending its user name to the server:

```c
byte SSH_MSG_KEXSRP_INIT
string username
```

Note that $e$ can not be computed yet, as the ring is not known. S replies with

```c
byte SSH_MSG_KEXSRP_REPLY
mpint q
mpint g
```
string s, salt

C computes e, and sends it to S:

byte SSH_MSG_KEXSRP_VALUE
mpint e

S computes f and K, and responds with

byte SSH_MSG_KEXSRP_VALUE
mpint f

The server MUST NOT send this message until after it has received e from the client.

Now the client can compute K. Both sides compute the exchange hash as the 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 n, the user name
string s, the salt
mpint q
mpint g
mpint e, exchange value sent by the client
mpint f, exchange value sent by the server
mpint K, the shared secret

The final exchange of SSH_MSG_KEXSRP_PROOF is unchanged. Note that the ability use different rings costs one more roundtrip.

Security Considerations

This entire draft discusses an authentication and key-exchange system that protects passwords and exchanges keys across an untrusted network. Most of this section is taken from [SRP], which also provides more details.

Knowledge of the verifier enables an attacker to mount an offline search (also known as a “dictionary attack”) on the user’s password, as well as to impersonate the server. So the verifier should be kept secret. The <name, salt, verifier> entry can be created on the user’s machine and transferred to the server, just like a user’s public key, or it could be created on the server. The former approach has the advantage that the cleartext password is not even temporarily known by the server.
SRP has been designed not only to counter the threat of casual password-sniffing, but also to prevent a determined attacker equipped with a dictionary of passwords from guessing at passwords using captured network traffic. The SRP protocol itself also resists active network attacks, and implementations can use the securely exchanged keys to protect the session against hijacking and provide confidentiality.

The SRP key exchange was originally designed primarily as a user authentication method, but it also provides a peculiar form of host authentication. If SRP succeeds, using a particular user name and password, the client can be confident that the remote server knows some verifier corresponding to that password. But if the same password is used with several servers, the client can’t distinguish them from each other, even if the actual verifiers are not shared between servers.

As some of the best known algorithms for computing discrete logarithms use extensive precomputations, it is desirable not to depend on a single fixed group like the multiplicative group used with "srp-rings1-sha1". However, care must be taken whenever the a client starts to use a new ring. Consider an attacker that knows how to compute discrete logarithms in the multiplicative group of a particular ring, and can convince the client to use that group. According to Tom Wu, the worst the attacker can do is getting information that enables him to verify guessed passwords.

In "diffie-hellman-group-exchange-sha1" [PROVOS] the client knows the server’s hostkey a priori, and uses that to authenticate the groups the server proposes.

With SRP, authenticating a proposed ring seems more difficult; if the ring is weak, authenticating it using the negotiated session key proves nothing.

SRP also has the added advantage of permitting the host to store passwords in a form that is not directly useful to an attacker. Even if the host’s password database were publicly revealed, the attacker would still need an expensive dictionary search to obtain any passwords. The exponential computation required to validate a guess in this case is much more time-consuming than the hash currently used by most UNIX systems. Hosts are still advised, though, to try their best to keep their password files secure.

At the time of this writing, SRP is still quite a new protocol, and it is too early to say definitely that it is secure. It is therefore recommended not to use SRP for general remote access that lets the client to execute arbitrary programs on the server.
SRP can be used for read-only access to public files (such as the server’s host key, or a user's known_hosts file). Used in this way, SRP can be used to obtain an authentic public key for the server, while a more conservative authentication mechanism is used for further access.

Further questions

This document should give a list of rings that can be used, which should include the rings used by libsrp (is there any specification, besides the source code, that lists these rings)? In general, to what extent should the protocol be compatible with libsrp?

Rings can be transmitted either by sending modulo and generator explicitly, like above, or by identifying rings with names or numbers.

It may be a good idea to optionally include the server’s host key in the SSH_MSG_KEXSRP_REPLY above, and in the exchange hash. It is not needed for the SRP exchange, but it is a convenient way to transmit an authentic host key, and it is useful for key re-exchanges later on.

To strengthen host authentication, in the case that a user has the same password on several servers, it may be a good idea to include the hostname somewhere in the computation of x, either in the user name or in the salt.

One can also consider adding the description of the group as another element in the computation of x, to add robustness in the "middle-man-sends-booby-trapped-group" scenario. More analysis is needed to say if adding the group description would really help, though.

Author’s Address

Niels Møller
LSH author
Slåtbaksvågen 48
120 51 Ärsta
Sweden

EMail: nisse@lysator.liu.se

References

RFC 2945

Draft, draft-ietf-secsh-architecture-07.txt

Draft, draft-ietf-secsh-transport-09.txt

Internet Draft, draft-ietf-secsh-userauth-09.txt

Full Copyright Statement

Copyright (C) The Internet Society (1997). All Rights Reserved.

This document and translations of it may be copied and furnished to
others, and derivative works that comment on or otherwise explain it
or assist in its implementation may be prepared, copied, published and
distributed, in whole or in part, without restriction of any kind,
provided that the above copyright notice and this paragraph are
included on all such copies and derivative works. However, this
document itself may not be modified in any way, such as by removing
the copyright notice or references to the Internet Society or other
Internet organizations, except as needed for the purpose of developing
Internet standards in which case the procedures for copyrights defined
in the Internet Standards process must be followed, or as required to
translate it into languages other than English.

The limited permissions granted above are perpetual and will not be
revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an
"AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING
TASK FORCE DISCLAIMS 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."