Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Español
Français
Polski
Русский
Edit links

Secure Remote Password protocol

From Wikipedia, the free encyclopedia

The **Secure Remote Password protocol** (**SRP**) is an augmented password-authenticated key agreement (PAKE) protocol, specifically designed to work around existing patents.[1]

Like all PAKE protocols, an eavesdropper or man in the middle cannot obtain enough information to be able to brute force guess a password without further interactions with the parties for each guess. This means that strong security can be obtained using weak passwords. Furthermore, being an augmented PAKE protocol, the server does not store password-equivalent data. This means that an attacker who steals the server data cannot masquerade as the client unless they first perform a brute force search for the password.

In layman's terms, during SRP (or any other PAKE protocol) authentication, one party (the "client" or "user") demonstrates to another party (the "server") that they know the password, without sending the password itself nor any other information from which the password can be broken. The password never leaves the client and is unknown to the server.

Contents [hide]

Overview [edit]

The SRP protocol has a number of desirable properties: it allows a user to authenticate themselves to a server, it is resistant to dictionary attacks mounted by an eavesdropper, and it does not require a trusted third party. It effectively conveys a zero-knowledge password proof from the user to the server. In revision 6 of the protocol only one password can be guessed per connection attempt. One of the interesting properties of the protocol is that even if one or two of the cryptographic primitives it uses are attacked, it is still secure. The SRP protocol has been revised several times, and is currently at revision 6a.

The SRP protocol creates a large private key shared between the two parties in a manner similar to Diffie–Hellman key exchange based on the client side having the user password and the server side having a cryptographic verifier derived from the password. The shared public key is derived from two random numbers, one generated by the client, and the other generated by the server, which are unique to the login attempt. In cases where encrypted communications as well as authentication are required, the SRP protocol is more secure than the alternative SSH protocol and faster than using Diffie–Hellman key exchange with signed messages. It is also independent of third parties, unlike Kerberos. The SRP protocol, version 3 is described in RFC 2945. SRP version 6 is also used for strong password authentication in SSL/TLS[2] (in TLS-SRP) and other standards such as EAP[3] and SAML, and is being standardized in IEEE P1363 and ISO/IEC 11770-4.

Protocol [edit]

The following notation is used in this description of the protocol, version 6:

- q and $N = 2q + 1$ are chosen such that both are prime (which makes q a Sophie Germain prime and N a safe prime). N must be large enough so that computing discrete logarithms modulo N is infeasible.
- All arithmetic is performed in the ring of integers modulo N, $\mathbb{Z}_N$. This means that below g^x should be read as $g^x \bmod N$
- g is a generator of the multiplicative group $\mathbb{Z}_N^*$.

- $H()$ is a hash function; e.g., SHA-256.
- k is a parameter derived by both sides; in SRP-6, $k = 3$, while in SRP-6a it is derived from N and g : $k = H(N, g)$. It is used to prevent a 2-for-1 guess when an active attacker impersonates the server.[4][5]
- s is a small salt.
- I is an identifying username.
- p is the user's password.
- v is the host's password verifier, $v = g^x$ where at a minimum $x = H(s, p)$. As x is only computed on the client it is free to choose a stronger algorithm. An implementation could choose to use $x = H(s \mid I \mid p)$ without affecting any steps required of the host. The standard RFC2945 defines $x = H(s \mid H(I \mid ":" \mid p))$. Use of I within x avoids a malicious server from being able to learn if two users share the same password. As the RFC was written by the inventor of SRP it demonstrates that can do anything you like to the stretch the password which is only handled at the client for example PBKDF2 stretch the raw password.
- A and B are random one time ephemeral keys of the user and host respectively.
- | (pipe) denotes concatenation.

All other variables are defined in terms of these.

First, to establish a password p with server Steve, client Carol picks a small random salt s, and computes $x = H(s, p)$, $v = g^x$. Steve stores v and s, indexed by I, as Carol's password verifier and salt. Carol must not share with anybody, and safely erase x at this step, because it is equivalent to the plaintext password p. This step is completed before the system is used as part of the user registration with Steve. Note that the salt s is shared and exchanged to negotiate a session key later so the value could be chosen by either side but is done by Carol so that she can register I, s and v in a single registration request. The transmission and authentication of the registration request is not covered in SRP.

Then to perform a proof of password at a later date the following exchange protocol occurs:

1. Carol → Steve: generate random value a; send I and $A = g^a$
2. Steve → Carol: generate random value b; send s and $B = kv + g^b$
3. Both: $u = H(A, B)$
4. Carol: $S_{\text{Carol}} = (B - kg^x)^{(a + ux)} = (kv + g^b - kg^x)^{(a + ux)} = (kg^x - kg^x + g^b)^{(a + ux)} = (g^b)^{(a + ux)}$
5. Carol: $K_{\text{Carol}} = H(S_{\text{Carol}})$
6. Steve: $S_{\text{Steve}} = (Av^u)^b = (g^a v^u)^b = [g^a (g^x)^u]^b = (g^{a + ux})^b = (g^b)^{(a + ux)}$
7. Steve: $K_{\text{Steve}} = H(S_{\text{Steve}}) = K_{\text{Carol}}$

Now the two parties have a shared, strong session key K. To complete authentication, they need to prove to each other that their keys match. One possible way is as follows:

1. Carol → Steve: $M_1 = H[H(N) \text{ XOR } H(g) \mid H(I) \mid s \mid A \mid B \mid K_{\text{Carol}}]$. Steve verifies M_1.
2. Steve → Carol: $M_2 = H(A \mid M_1 \mid K_{\text{Steve}})$. Carol verifies M_2.

This method requires guessing more of the shared state to be successful in impersonation than just the key. While most of the additional state is public, private information could safely be added to the inputs to the hash function, like the server private key.[*clarification needed*]

Alternatively, in a password-only proof the calculation of K can be skipped and the shared S proven with:

1. Carol → Steve: $M_1 = H(A \mid B \mid S_{\text{Carol}})$. Steve verifies M_1.
2. Steve → Carol: $M_2 = H(A \mid M_1 \mid S_{\text{Steve}})$. Carol verifies M_2.

When using SRP to negotiate a shared key K which will be immediately used after the negotiation the verification steps of M_1 and M_2 may be skipped. The server will reject the very first request from the client which it cannot decrypt.

The two parties also employ the following safeguards:

1. Carol will abort if she receives $B = 0 \pmod N$ or $u = 0$.
2. Steve will abort if he receives $A \pmod N = 0$.
3. Carol must show her proof of K (or S) first. If Steve detects that Carol's proof is incorrect, he must abort without showing his own proof of K (or S)

Implementation example in Python [edit]

```
# An example SRP authentication
# WARNING: Do not use for real cryptographic purposes beyond testing.
# based on http://srp.stanford.edu/design.html
import hashlib
```

```
import random

def global_print(*names):
    x = lambda s: ["{}", "0x{:x}"] [hasattr(s, 'real')].format(s)
    print("".join("{} = {}\n".format(name, x(globals()[name])) for name in
names))

# note: str converts as is, str( [1,2,3,4] ) will convert to "[1,2,3,4]"
def H(*args):  # a one-way hash function
    a = ':'.join(str(a) for a in args)
    return int(hashlib.sha256(a.encode('utf-8')).hexdigest(), 16)

def cryptrand(n=1024):
    return random.SystemRandom().getrandbits(n) % N

# A large safe prime (N = 2q+1, where q is prime)
# All arithmetic is done modulo N
# (generated using "openssl dhparam -text 1024")
N = '''00:c0:37:c3:75:88:b4:32:98:87:e6:1c:2d:a3:32:
       4b:1b:a4:b8:1a:63:f9:74:8f:ed:2d:8a:41:0c:2f:
       c2:1b:12:32:f0:d3:bf:a0:24:27:6c:fd:88:44:81:
       97:aa:e4:86:a6:3b:fc:a7:b8:bf:77:54:df:b3:27:
       c7:20:1f:6f:d1:7f:d7:fd:74:15:8b:d3:1c:e7:72:
       c9:f5:f8:ab:58:45:48:a9:9a:75:9b:5a:2c:05:32:
       16:2b:7b:62:18:e8:f1:42:bc:e2:c3:0d:77:84:68:
       9a:48:3e:09:5e:70:16:18:43:79:13:a8:c3:9c:3d:
       d0:d4:ca:3c:50:0b:88:5f:e3'''
N = int(''.join(N.split()).replace(':', ''), 16)
g = 2        # A generator modulo N

k = H(N, g)  # Multiplier parameter (k=3 in legacy SRP-6)

print("#. H, N, g, and k are known beforehand to both client and server:")
global_print("H", "N", "g", "k")

print("0. server stores (I, s, v) in its password database")

# the server must first generate the password verifier
I = "person"         # Username
p = "password1234"   # Password
s = cryptrand(64)    # Salt for the user
x = H(s, I, p)       # Private key
v = pow(g, x, N)     # Password verifier
global_print("I", "p", "s", "x", "v")

print("1. client sends username I and public ephemeral value A to the server")
a = cryptrand()
A = pow(g, a, N)
global_print("I", "A")  # client->server (I, A)

print("2. server sends user's salt s and public ephemeral value B to client")
b = cryptrand()
B = (k * v + pow(g, b, N)) % N
global_print("s", "B")  # server->client (s, B)

print("3. client and server calculate the random scrambling parameter")
u = H(A, B)  # Random scrambling parameter
global_print("u")

print("4. client computes session key")
x = H(s, I, p)
S_c = pow(B - k * pow(g, x, N), a + u * x, N)
K_c = H(S_c)
global_print("S_c", "K_c")

print("5. server computes session key")
S_s = pow(A * pow(v, u, N), b, N)
K_s = H(S_s)
global_print("S_s", "K_s")

print("6. client sends proof of session key to server")
M_c = H(H(N) ^ H(g), H(I), s, A, B, K_c)
```

```
global_print("M_c")
# client->server (M_c) ; server verifies M_c

print("7. server sends proof of session key to client")
M_s = H(A, M_c, K_s)
global_print("M_s")
# server->client (M_s) ;  client verifies M_s
```

Outputs [edit]

```
#. H, N, g, and k are known beforehand to both client and server:
H = <function H at 0x101f1fa60>
N =
0xc037c37588b4329887e61c2da3324b1ba4b81a63f9748fed2d8a410c2fc21b1232f0d3bfa024276cfd
88448197aae486a63bfca7b8bf7754dfb327c7201f6fd17fd7fd74158bd31ce772c9f5f8ab584548a99a
759b5a2c0532162b7b6218e8f142bce2c30d7784689a483e095e701618437913a8c39c3dd0d4ca3c500b
885fe3
g = 0x2
k = 0xb317ec553cb1a52201d79b7c12d4b665d0dc234fdbfd5a06894c1a194f818c4a

0. server stores (I, s, v) in its password database
I = person
p = password1234
s = 0x23c52769f89b02c0
x = 0x28a914ef69978f5fe544f030bea89eab675bcaa2ec79cd36efa1d410d27d5215
v =
0xa636254492ec0f7391d6b596ec926b91866775072dfd758c6ebc51bf7277ec6ca97f6cf0316d7fa90a
2b9e87366cf813a53dcdc6ab303fcc932a5783f62affb7e0275189f165b8b919a2067404e6f2aa0534c9
9a3224a6365c1367dcd9ef005376d6f20a2b300c307f7afcedea08fb2d7a3340f13b5b9e35d52f0b8267
0ab17e

1. client sends username I and public ephemeral value A to the server
I = person
A =
0x48147d013e3a2e08ace222a0ab914a7ed67c704b2480716b53f9d229243d1725473cf4451904658597
f487b0fa8bc7d544671b25563f095bad384cbb8da7f58f7f13c8fa8bb9d6aade5fe02df288f2b38d71d5
1036ede52802645f82cd7216535c0c978f90230e0f878163a638cf57ad11968169c26e467b8ee14eb2ca
5b1614

2. server sends user's salt s and public ephemeral value B to client
s = 0x23c52769f89b02c0
B =
0x709f340738e62e46184634acd2cd7c861a7d92c5fde9eb43ac120226a0eb6601ee5d1a0b92ffb62541
70d91fb451c3c02bbf8b41f9e7e3e885d709f0dc4808048e595c68448a2111b45eefaa1e2d6a4814d99a
e038a5f2371c753b312c529cada66b23e6512c7ef814683f4cfe2a4c5413c434e21bc689d869fc969141
b84a61

3. client and server calculate the random scrambling parameter
u = 0x78e4f2723b9ee5f69c7225469c70263cb39580dd4414b82ab9960def0ac9ef68

4. client computes session key
S_c =
0x94ea4b72b61d4330cf44f31e5c710491d41abdd6dd5b66b277bc517addbe89d9aa002645897567ae77
96d1574f5d7f62cf96d2246dabfbc919cf1444d69097ceaf5476bc3964cacd52697e346f5e5a424c2c89
b661d2eba34e5c71955734421956114 97f606fa49639f873f385d0f6cdb9308fe2b0777d1a89bbaebe9d
f237a4
K_c = 0x3f1e089e02b3770a5e4ab27b3a04415e54826fe4b729cd37b86fbe59b9e0d3c6

5. server computes session key
S_s =
0x94ea4b72b61d4330cf44f31e5c710491d41abdd6dd5b66b277bc517addbe89d9aa002645897567ae77
96d1574f5d7f62cf96d2246dabfbc919cf1444d69097ceaf5476bc3964cacd52697e346f5e5a424c2c89
b661d2eba34e5c71955734421956114 97f606fa49639f873f385d0f6cdb9308fe2b0777d1a89bbaebe9d
f237a4
K_s = 0x3f1e089e02b3770a5e4ab27b3a04415e54826fe4b729cd37b86fbe59b9e0d3c6

6. client sends proof of session key to server
M_c = 0x21d1546a18f923907b975091341316ca03bacf9cfd61b33f66d87e07eacff18

7. server sends proof of session key to client
M_s = 0x937ee2752d2d0a18eea2e7d4c5aa0dd0df54970f4c99fc13c75c5db3bba45643
```

Implementations [edit]

- OpenSSL version 1.0.1 or later.
- Botan (the C++ crypto library) contains an implementation of SRP-6a
- TLS-SRP is a set of ciphersuites for transport layer security that uses SRP.
- srp-client SRP-6a implementation in JavaScript (compatible with RFC 5054), open source, Mozilla Public License (MPL) licensed.
- The JavaScript Crypto Library includes a JavaScript implementation of the SRP protocol, open source, BSD licensed.
- Gnu Crypto provide a Java implementation licensed under the GNU General Public License with the "library exception", which permits its use as a library in conjunction with non-Free software.
- The Legion of the Bouncy Castle provides Java and C# implementations under the MIT License.
- Nimbus SRP is a Java library providing a verifier generator, client and server-side sessions. Includes interfaces for custom password key, client and server evidence message routines. No external dependencies. Released under the Apache 2.0 license.
- srplibcpp is a C++ implement base on MIRACL.
- DragonSRP is a C++ modular implementation currently works with OpenSSL
- Json2Ldap provides SRP-6a authentication to LDAP directory servers.
- csrp SRP-6a implementation in C.
- Crypt-SRP SRP-6a implementation in Perl.
- pysrp SRP-6a implementation in Python (compatible with csrp).
- py3srp SRP-6a implementation in pure Python3.
- Meteor web framework's Accounts system implements SRP for password authentication.
- srp-rb SRP-6a implementation in Ruby
- srp-6a-demo SRP-6a implementation in PHP and JavaScript
- thinbus-srp-js SRP-6a implementation in JavaScript. Comes with compatible Java classes which use Nimbus SRP a demonstration app using Spring Security. There is also a demonstration application performing authentication to a PHP server. Released under the Apache License.
- Stanford JavaScript Crypto Library (SJCL) implements SRP for key exchange
- node-srp is a JavaScript client and server (node.js) implementation of SRP
- SRP6 for C# and Java implementation in C# and Java
- ALOSRPAuth is an Objective-C implementation of SRP-6a
- go-srp is a Go implementation of SRP-6a

References [edit]

1. ^ "What is SRP?". Stanford University.
2. ^ Taylor, David; Tom Wu; Nikos Mavrogiannopoulos; Trevor Perrin (November 2007). "Using the Secure Remote Password (SRP) Protocol for TLS Authentication". RFC 5054
3. ^ Carlson, James; Bernard Aboba; Henry Haverinen (July 2001). "EAP SRP-SHA1 Authentication Protocol". IETF. Draft.
4. ^ Wu, Tom (October 29, 2002). "SRP-6: Improvements and Refinements to the Secure Remote Password Protocol".
5. ^ "SRP Protocol Design".

See also [edit]

- Challenge–response authentication
- Password-authenticated key agreement
- Salted Challenge Response Authentication Mechanism (SCRAM)
- Simple Password Exponential Key Exchange
- Zero-knowledge password proof

External links [edit]

- Official website
- SRP License—BSD like open source.
- US6539479 - SRP Patent (Expired on May 12, 2015 due to failure to pay maintenance fees (according to Google Patents). Originally set to expire in July 2018).

Manual pages [edit]

- pppd(8): Point-to-Point Protocol Daemon
- srptool(1): Simple SRP password tool

RFCs [edit]

- RFC 2944 - Telnet Authentication: SRP
- RFC 2945 - The SRP Authentication and Key Exchange System
- RFC 3720 - Internet Small Computer Systems Interface (iSCSI)
- RFC 3723 - Securing Block Storage Protocols over IP
- RFC 3669 - Guidelines for Working Groups on Intellectual Property Issues
- RFC 5054 - Using the Secure Remote Password (SRP) Protocol for TLS Authentication

Other links [edit]

- IEEE 1363
- SRP Intellectual Property Slides (Dec 2001 - possible deprecated) The EKE patents mentioned expired in 2011 and 2013.

v · t · e	Public-key cryptography	
Algorithms	Integer factorization	Benaloh · Blum–Goldwasser · Cayley–Purser · Damgård–Jurik · GMR · Goldwasser–Micali · Naccache–Stern · Paillier · Rabin · RSA · Okamoto–Uchiyama · Schmidt–Samoa
	Discrete logarithm	BLS · Cramer–Shoup · DH · DSA · ECDH · ECDSA · EdDSA · EKE · ElGamal (signature scheme) · MQV · Schnorr · SPEKE · **SRP** · STS
	Lattice/SVP/CVP/LWE/SIS	NTRUEncrypt · NTRUSign · RLWE-KEX · RLWE-SIG · BLISS
	Others	AE · CEILIDH · EPOC · HFE · IES · Lamport · McEliece · Merkle–Hellman · Naccache–Stern knapsack cryptosystem · Three-pass protocol · XTR
Theory	Discrete logarithm · Elliptic-curve cryptography · Non-commutative cryptography · RSA problem · Trapdoor function	
Standardization	CRYPTREC · IEEE P1363 · NESSIE · NSA Suite B · Post-Quantum Cryptography Standardization	
Topics	Digital signature · OAEP · Fingerprint · PKI · Web of trust · Key size · Post-quantum cryptography	

v · t · e	Cryptography
History of cryptography · Cryptanalysis · Outline of cryptography	
Symmetric-key algorithm · Block cipher · Stream cipher · Public-key cryptography · Cryptographic hash function · Message authentication code · Random numbers · Steganography	
Category · Portal · WikiProject	

Categories: Key-agreement protocols | Password authentication