Network Working Group
Internet Draft: SCRAM-SHA1 SASL Mechanism
Document: draft-newman-auth-scram-00.txt

C. Newman
Innosoft

September 1997
Expires in six months

Salted Challenge Response Authentication Mechanism (SCRAM)

Status of this memo

Abstract

SCRAM is a simple passphrase-based authentication mechanism which uses only a publicly available cryptographic hash function to provide authentication for protocols. It is designed to replace plaintext password mechanisms without significant additional complexity, loss of performance or plaintext equivalent verifiers.

CRAM-MD5 [CRAM-MD5], a similar mechanism, has the drawback that the password verifier stored on the server can be used to impersonate the user. Current plaintext password mechanisms do not have this drawback and it is a serious issue for servers which allow remote login or sites which distribute the authentication database to multiple servers via an insecure protocol. SCRAM-SHA1 corrects this drawback with minimal additional complexity.

This document defines the SCRAM-SHA1 SASL mechanism [SASL] using the SHA1 [SHA1] and HMAC-SHA1 [HMAC] algorithms.

0. Open Issues

(1) Although this mechanism has no new concepts, it has not had extensive review. Advice on the completeness of the security considerations is appreciated.

(2) The TWEKE proposal (appendix B) is more secure and more complex. Is it an acceptable or desirable tradeoff? Perhaps SCRAM could get 40% penetration into the plaintext market and TWEKE could get 30%, would it be worthwhile to do TWEKE instead of SCRAM in this case? What if there's a bigger difference?

1. Conventions Used in this Document

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as defined in "Key words for use in RFCs to Indicate Requirement Levels" [KEYWORDS].

2. Client Implementation of SCRAM-SHA1

This section includes a step-by-step guide for client implementors. Although section 7 contains the formal definition of the syntax and is the authoritative reference in case of errors here, this section should be sufficient to build a correct implementation.

When used with SASL the mechanism name is "SCRAM-SHA1". The mechanism does not provide a security layer.

The client begins by sending a message to the server containing three pieces of information:

(1) An authorization identity. When the empty string is used, this defaults to the authentication identity. This is used by system administrators or proxy servers to login with a different user identity. This field may be up to 255 octets and is terminated by a NUL (0) octet. US-ASCII printable characters are preferred, although UTF-8 [UTF-8] printable characters are permitted to support international names. Use of character sets other than US-ASCII and UTF-8 is forbidden.

(2) An authentication identity. The identity whose passphrase will be used. This field may be up to 255 octets and is terminated by a NUL (0) octet. US-ASCII printable characters are preferred, although UTF-8 [UTF-8] printable characters are permitted to support international names. Use of character sets other than US-ASCII and UTF-8 is forbidden.

(3) A "client nonce" of 8 to 256 octets. It is important that this be globally unique and somewhat random. It can be generated by appending the system clock to a random number (advice for generating good random numbers can be found in [RANDOM]) and the client's IP address or domain name.

The server responds by sending a message containing three pieces of information:

(4) An 8-octet salt value, specific to the authentication identity.

(5) A server id consisting of the service name of the protocol's SASL profile followed by a "." followed by the domain name of the server followed by an "@" and optional extension data terminated by NUL. This will not be longer than 512 octets. The client SHOULD verify this is correct.

(6) A "server nonce" of 8 to 32 octets.

The client then does the following:

(A) Create a buffer containing the user's passphrase. The client MUST support passphrases of at least 64 octets. US-ASCII characters are preferred, although UTF-8 characters are permitted. Character sets other than UTF-8 MUST NOT be used.

(B) Apply the SHA1 function to (A), producing a 20 octet result. Once this is done, (A) SHOULD be erased from memory.

(C) Apply the HMAC-SHA1 function with the result of (B) as the key and the 8-octet salt (4) value as the data. This produces a 20 octet result.

(D) Create a buffer containing the server's response (4)-(6), immediately followed by the initial client message (1)-(3).

(E) Apply the HMAC-SHA1 function with the result of (C) as the key and the buffer from (D) as the data. This produces a 20-octet result.

(F) Create a 20-octet buffer containing the exclusive-or of (B) and (E).

The client then sends a message to the server containing the following:

(7) The 20-octet result of step (F).

If authentication is successful, then the server responds with the following:

(8) A 20-octet mutual authentication verifier.

The client SHOULD verify this with the following procedure:

(G) Create a buffer containing the initial client message (1)-(3) immediately followed by the initial server response (4)-(6).

(H) Apply the HMAC-SHA1 function with the result of (C) as the key and the buffer from (G) as the data.

(I) If the result of (H) matches (8), the server is authenticated.

A secured client MAY store the result of (B) to re-authenticate. Permanent storage of (B) by the client is discouraged although it is preferable to storing the actual passphrase.

3. Server Implementation of SCRAM-SHA1.

The section includes a step-by-step guide for server implementors. Although section 7 contains the formal definition of the syntax and is the authoritative reference in case of errors here, this section in conjunction with section 2 should be sufficient to build a correct implementation.

The server's authentication database contains an 8-octet salt and 20-octet verifier for each local user. The server MAY support remote users using the syntax "user@host" for the authentication identity, but if it doesn't it MUST truncate the authentication identity at the "@" sign prior to lookup in the authentication database.

The authentication verifier is equal to the result of step (C) above. To create its initial response, the server simply looks up the authentication identity to fetch the salt, and generates an 8 to 32 octet nonce. This nonce MUST be unique to prevent replay attacks. It can be generated by appending a system clock to a random number [RANDOM]. To verify the client's credentials, the server preforms the following steps:

(a) Generate a buffer identical to step (D) for the client.

(b) Apply the HMAC-SHA1 function with the stored verifier as the key and the result of (a) as the data. This produces a 20-octet result equal to step (E) above.

(c) Exclusive-or the result of (b) with message (7) from the client. This produces a 20-octet result which should be equal to the output of step (B) above.

(d) Apply the HMAC-SHA1 function with (c) as the key and the stored salt as the data. This produces a 20-octet result.

(e) if the result of (d) is equal to the stored verifier, then the user is authenticated.

(f) Generate a buffer identical to step (G) above.

(g) Apply the HMAC-SHA1 function with the stored verifier as the key and the buffer from (f) as the data. This produces a 20-octet result.

The result of (g) is sent to the client as the mutual authentication step.

4. Example

XXX: to be done

5. System Administrator Advice

This section includes advice for system administrators using this mechanism.

Although the verifiers used by SCRAM-SHA1 are probably more secure than those used by current plaintext mechanisms (such as Unix /etc/password), it is still very important to keep them secret. Just as tools exist to try common passwords against Unix /etc/password files, it is also possible to build such tools for SCRAM-SHA1. In addition, once a SCRAM-SHA1 verifier is stolen, a passive (undetectable) snoop of that user logging in will result in the output of step (B) above, which is sufficient to impersonate a user. This is far better than current plaintext mechanisms where a passive snoop always recovers the user's password, but is still a serious concern.

Verifiers SHOULD be kept hidden from all users on the server. Sites which distribute verifiers among multiple servers, SHOULD encrypt them when distributing them.

SCRAM-SHA1 is only a good mechanism if passphrases are well chosen. For this reason, implementations should use the term "passphrase"

rather than "password" and when a user's passphrase is set, site policy restrictions should be applied. A reasonable site policy would require passphrases of at least 10 characters with at least one non-alphanumeric character.

SCRAM-SHA1 doesn't protect the integrity or privacy of data exchanged after authentication. Use of an external encryption layer or a stronger authentication mechanism such as Kerberos is encouraged if this functionality is needed.

6. SCRAM-SHA1 Functional Notation

This section is designed to provide a quick understanding of SCRAM-SHA1 for the mathematically inclined.

+	octet concatenation
XOR	the exclusive-or function
AU	is the authentication user identity (NUL terminated)
AZ	is the authorization user identity (NUL terminated) if AZ would be the same as AU, a single NUL is used instead.
SV	is the name of the service and server
p	is the plaintext passphrase
H(x)	is a one-way hash function applied to "x", such as SHA-1
M(x,y)	is a message authentication code (MAC) such as HMAC-SHA1 "y" is the key and "x" is the text signed by the key.
V	is a per-user verifier the server stores
s	is a per-user salt value the server stores
P	is the proof the client sends the server
Us	is a unique nonce the server sends to the client
Uc	is a unique nonce the client sends to the server

The verifier (V) is computed by applying the hash function to the plaintext passphrase, then using the result to sign the salt. Thus:

V = M(s, H(p))

The proof (P) is computed as follows:

P = H(p) XOR M(s + SV + Us + AZ + AU + Uc, V)

The SCRAM exchange is as follows:

client -> server: AZ + AU + Uc
server -> client: s + SV + Us
client -> server: P
server -> client: M(AZ + AU + Uc + s + SV + Us, V)

The server verifies P by checking that the following is equal to V:

M(s, P XOR M(s + SV + Us + AZ + AU + Uc, V))

The client verifies the server's identity by performing the same computation the server does and comparing it to the server's result.

7. Formal Syntax of SCRAM-SHA1 Messages

This is the formal syntactic definition of the client and server messages. This uses the ABNF [ABNF] notation.

```
client-msg-1     = [authorize-id] NUL authenticate-id NUL client-nonce

server-msg-1     = salt server-id NUL server-nonce

client-msg-2     = proof

server-msg-2     = mutual-auth

passphrase       = 8*UTF8-SAFE
                   ;; At least 64 octets MUST be supported


authorize-id     = *UTF8-PRINT
                   ;; No more than 255 octets

authenticate-id  = *UTF8-PRINT
                   ;; No more than 255 octets

server-id        = service-name "." server-domain
                   "@" [ server-ext-data ]
                   ;; No more that 511 octets total

service-name     = *USASCII-PRINT
                   ;; a GSSAPI service name

server-domain    = *USASCII-PRINT
                   ;; an internet domain name

server-ext-data  = *UTF8-SAFE
                   ;; extension data

server-id        = *UTF8-PRINT
                   ;; No more than 511 octets
```

```
client-nonce     = 8*256OCTET

server-nonce     = 8*32OCTET

salt             = 8OCTET

proof            = 20OCTET

mutual-auth      = 20OCTET

NUL              = %x00  ;; US-ASCII NUL character

US-ASCII-SAFE    = %x01-09 / %x0B-0C / %x0E-7F
                    ;; US-ASCII except CR, LF, NUL

US-ASCII-PRINT   = %x20-7E
                    ;; printable US-ASCII including SPACE

UTF8-SAFE        = US-ASCII-SAFE / UTF8-1 / UTF8-2 / UTF8-3
                      / UTF8-4 / UTF8-5

UTF8-PRINT      = US-ASCII-PRINT / UTF8-1 / UTF8-2 / UTF8-3
                      / UTF8-4 / UTF8-5

UTF8-CONT        = %x80..BF

UTF8-1           = %xC0..DF UTF8-CONT

UTF8-2           = %xE0..EF 2UTF8-CONT

UTF8-3           = %xF0..F7 3UTF8-CONT

UTF8-4           = %xF8..FB 4UTF8-CONT

UTF8-5           = %xFC..FD 5UTF8-CONT
```

8. Security Considerations

Security considerations are discussed throughout this document. The security considerations of SHA1 [SHA1] and HMAC [HMAC] also apply.

SCRAM-SHA1 is conjectured to be a reasonably strong mechanism as long as passphrases are well chosen and verifiers are kept secret. Making a SCRAM-SHA1 verifier public is believed to be no worse than making a Unix /etc/password verifier public when a plaintext-only mechanism is used.

There are two particularly dangerous attacks against SCRAM-SHA1. The first is to passively record an authentication session (or steal the verifier) and perform an offline dictionary attack to find the passphrase. This type of attack is estimated to be about 40% effective at typical sites with current behavior patterns [SCHNEIER]. Use of the term "passphrase", enforcement of site policy when passphrases are changed and user education may improve this to acceptable levels for many sites.

The second attack is to both steal the verifier for a user and passively record an authentication session by that user. This results in the ability to impersonate that user and more than doubles the speed of a dictionary attack or brute-force attack to recover the actual passphrase. For this reason, verifiers should be kept well-protected.

This mechanism provides no protection for the session after authentication. A passive observer can see information transmitted, and an active attacker can hijack the session. Use of an external encryption layer such as TLS [TLS] can address this problem.

This mechanism uses a hash-function combined with exclusive-or as a simple single-block cipher. [SCHNEIER] expresses reservations about ciphers built using one-way hash functions, although not all of his reservations may apply to this limited use.

9. Intellectual Property Issues and Prior Art

The author is not aware of any patents which apply to this mechanism.

This is primarily a derivative of simple hash-based challenge response systems. The hash-based challenge response idea has existed since at least 1992, when the RIPE project published the SKID algorithm according to [SCHNEIER].

The repeated-hash idea used to verify the client's authenticator is derived from S/KEY [SKEY].

The idea of using a hash function to construct a cipher (with exclusive-or) was originally invented by Peter Gutmann in 1993 according to [SCHNEIER].

The idea of using salt to protect against global dictionary attacks dates back to the unix /etc/password system or before. There is some discussion of this in [SCHNEIER].

SCRAM combines these four techniques. The author of this specification first proposed this publicly on a mailing list July 16, 1997. There is nothing new about this mechanism beyond the idea of combining these existing techniques.

The SCRAM algorithm includes a single-block cipher capable of encrypting 20 octets of authentication data. The author does not believe this will cause problems for export restrictions, but checking with the appropriate government(s) should be considered. Computer readable source code for cryptographic hash functions such as MD5 and SHA1 have been exported from the United States without problems.

10. References

[ABNF] Crocker, D., "Augmented BNF for Syntax Specifications: ABNF", Work in progress: draft-ietf-drums-abnf-xx.txt

[CRAM-MD5] Klensin, Catoe, Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", RFC 2095, MCI, January 1997.

<ftp://ds.internic.net/rfc/rfc2095.txt>

[HMAC] Krawczyk, Bellare, Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, IBM, UCSD, February 1997.

<ftp://ds.internic.net/rfc/rfc2104.txt>

[IMAP4] Crispin, M., "Internet Message Access Protocol - Version 4rev1", RFC 2060, University of Washington, December 1996.

<ftp://ds.internic.net/rfc/rfc2060.txt>

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

<ftp://ds.internic.net/rfc/rfc2119.txt>

[RANDOM] Eastlake, Crocker, Schiller, "Randomness Recommendations for Security", RFC 1750, DEC, Cybercash, MIT, December 1994.

<ftp://ds.internic.net/rfc/rfc1750.txt>

[SASL] Myers, "Simple Authentication and Security Layer (SASL)", work in progress.

[SCHNEIER] Schneier, "Applied Cryptography: Protocols, Algorithms and Source Code in C," John Wiley and Sons, Inc., 1996.

[SHA1] NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.

[SKEY] Haller, Neil M. "The S/Key One-Time Password System", RFC 1760, Bellcore, February 1995.

<ftp://ds.internic.net/rfc/rfc1760.txt>

[TLS] Dierks, Allen, "The TLS Protocol Version 1.0", Work in progress.

[UTF8] Yergeau, F. "UTF-8, a transformation format of Unicode and ISO 10646", RFC 2044, Alis Technologies, October 1996.

<ftp://ds.internic.net/rfc/rfc2044.txt>

11. Author's Address

Chris Newman
Innosoft International, Inc.
1050 Lakes Drive
West Covina, CA 91790 USA

Email: chris.newman@innosoft.com

A. Appendix - Sample Source Code

XXX: to be done

B. Appendix - TWEKE Proposal

Tom Wu has proposed adding a Diffie-Hellman key exchange to this mechanism. Diffie-Hellman works roughly as follows:

Server picks g, n and x. Server computes X = g^x mod n.

server -> client: g, n, X

Client picks y and computes Y = g^y mod n and K = X^y mod n.

client -> server: Y

Server computes K = Y^x mod n (which is the same as the client's K).

If g, n, x and y are sufficiently big and have the right

characteristics, then both the client and server share K which is very difficult for a passive evesdropper to obtain.

The TWEKE proposal would add the following steps:

(4.5) Server sends g, n, X.

(7.5) Client sends Y.

and would modify steps (D) and (a) to include the value K. This would result in a protocol safe from passive attacks. The expense would be reduced performance, the need for a bignum math library and a requirement that an export license be obtained from certain governments (included the United States). This would not defend against active attacks, but should be free of patent restrictions after October 6th, 1997.

TWEKE might be harder to deploy than SCRAM due to the higher math and the use of public key technology.

C. Appendix - Additional Services

Several additional services are needed to make SCRAM useful in various usage scenarios. These include remote authentication database support for servers, authentication database APIs for servers, remote passphrase change support for clients, single-sign-on APIs for clients and management tools. The server-id is included to facilite the remote authentication database service. Otherwise these issues are deferred for future work.