Network Working Group
Internet Draft: SCRAM-MD5 SASL Mechanism
Document: draft-newman-auth-scram-01.txt

C. Newman
Innosoft

October 1997
Expires in six months

Salted Challenge Response Authentication Mechanism (SCRAM)

Status of this memo

Abstract

SCRAM is a simple passphrase-based authentication mechanism suitable for a wide variety of usage scenarios. It combines the best properties of CRAM-MD5 [CRAM-MD5] and OTP [OTP] without a significant increase in complexity.

This document defines the SCRAM-MD5 SASL mechanism [SASL] using the MD5 [MD5] and HMAC-MD5 [HMAC] algorithms. It is suitable for use directly with protocols such as IMAP [IMAP4] and POP [POP3].

0. Changes from Previous Version

(1) MD5 is used instead of SHA1 as it permits backwards-compatibility with CRAM-MD5 verifiers on the server and it is a more commonly available hash function. This should not effect SCRAM's security significantly.

(2) The server-proof is now independent of the client-verifier. This allows a remote authentication service to efficiently restrict the services an application server may offer to clients.

(3) The salt is included in the client-key which means that a compromised verifier and client-server exchange only gains access to services which use the same salt.

(4) The service identifier is simplified and separated from the extension data which now has defined syntax.

(5) The client-nonce may be omitted for one-way authentication.

(6) Examples and sample source code has been added.

1. How to Read This Document

This document has information for several different audiences. Section 2 describes the highlights of SCRAM. Sections 3-6 are intended for implementors. Section 7 is intended for system administrators. Sections 7-10 are intended for security evaluation.

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. SCRAM Highlights

SCRAM is a simple passphrase-based mechanism, which does not require complicated public key technology. It can be implemented in a few pages of source code.

SCRAM is not plaintext equivalent, so it improves network security over plaintext mechanisms without sacrificing security of the authentication database.

SCRAM includes a salt to prevent global dictionary attacks on the server's authentication database.

SCRAM supports proxy authentication by including an authorization

identity (user to login as) separate from the authentication identity (server authentication database entry to use).

SCRAM supports optional mutual authentication.

SCRAM supports limited server trust. This means a centralized authentication server can restrict the services offered by application servers.

SCRAM-MD5 is backwards-compatible with existing CRAM-MD5 verifiers stored in server authentication databases.

3. Client Implementation of SCRAM-MD5

This section includes a step-by-step guide for client implementors. Although section 6 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-MD5". The mechanism does not provide a security layer.

The client begins by sending a message to the server containing the following three pieces of information. The entire message MUST NOT exceed 1000 characters.

(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) An optional "client nonce." If this is omitted, it indicates a desire for client-only authentication. When present, it is important that this be globally unique. One common technique for generating globally unique identifiers combines a process identifier with the system clock, a sequence number, a random number and the client's domain name. The random number is important as clocks are often synchronized using insecure protocols. Advice for generating strong random numbers can be found in [RANDOM].

The server responds by sending a message containing three pieces of information. The entire message MUST NOT exceed 1000 characters.

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

(5) A service id consisting of the service name of the protocol's SASL profile followed by an "@" followed by the domain name of the server and terminated by NUL. The client SHOULD verify this is correct.

(6) A string containing extension data, terminated by NUL. A client which supports TLS [TLS] or stronger SASL mechanisms can check this to protect against an active downgrade attack. A server which supports TLS or stronger SASL mechanisms SHOULD advertise them here.

(7) A "server nonce".

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 HMAC-MD5 function with (A) as the key and the 8-octet salt as the data, producing a 16-octet result. Once this is done, (A) SHOULD be erased from memory.

(C) Apply the MD5 function to the result of (B). This produces a 16-octet result.

(D) Apply the MD5 function to the result of (C). This produces a 16-octet result.

(E) Create a buffer containing the server's response (4)-(7), immediately followed by the initial client message (1)-(3).

(F) Apply the HMAC-MD5 function with the result of (D) as the key and the buffer from (E) as the data. This produces a 16-octet result.

(G) Create a 16-octet buffer containing the exclusive-or of (C) and (F).

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

(8) The 16-octet result of step (G).

If no client challenge was suppled in step 3, the one-way authentication is now complete.

(9) A 16-octet server authentication verifier.

The client SHOULD verify this with the following procedure:

(H) Apply the HMAC-MD5 function with the result of (B) as the key and the 8-octet salt as the data. This produces a 16-octet result.

(I) Create a buffer containing the initial client message (1)-(3) immediately followed by the initial server response (4)-(7).

(J) Apply the HMAC-MD5 function with the result of (H) as the key and the buffer from (I) as the data.

(K) If the result of (J) matches (9), the server is authenticated.

A secured client MAY store the result of (B) to re-authenticate to services using the same salt, or the intermediate HMAC state from

(B) to re-authenticate to any service. Clients SHOULD NOT store the passphrase itself.

4. Server Implementation of SCRAM-MD5.

The section includes a step-by-step guide for server implementors. Although section 6 contains the formal definition of the syntax and is the authoritative reference in case of errors here, this section in conjunction with section 3 should be sufficient to build a correct implementation.

The server's authentication database contains an 8-octet salt, 16-octet client verifier and a 16-octet server key for each local user. The server MUST support "user@host" syntax for the authentication identity at least to the extent of stripping "@host" when it matches the local hostname or rejecting the authentication immediately after the initial client message if the hostname doesn't match. The server MAY support remote user authentication using this syntax.

The stored client verifier is equal to the result of step (D) above, and the stored server key is equal to the result of step (J) above. To create its initial response, the server simply looks up the authentication identity to fetch the salt, and generates an 8 to 248 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 (E) above.

(b) Apply the HMAC-MD5 function with the stored client verifier as the key and the result of (a) as the data. This produces a 16-octet result equal to step (F) above.

(c) Exclusive-or the result of (b) with message (8) from the client. This produces a 16-octet result which should be equal to the result of step (C) above.

(d) Apply the MD5 function to the output of step (c). This

produces a 16-octet result which should be equal to the result of step (D) above.

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

If no client challenge was provided in step (3), the server is now done and responds with the appropriate status code.

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

(g) Apply the HMAC-MD5 function with the stored verifier as the key and the buffer from (f) as the data. This produces a 16-octet result.

The result of (g) is sent to the client to authenticate the server.

5. Example

The following is an example of the SCRAM-MD5 mechanism using the IMAP [IMAP4] profile of SASL. Note that base64 encoding and the lack of an initial client reponse with the first command are characteristics of the IMAP profile of SASL and not characteristics of SASL or SCRAM-MD5.

In this example, "C:" represents lines sent from the client to the server and "S:" represents lines sent from the server to the client. The wrapped lines are for editorial clarity -- there are no actual newlines in the middle of the messages.

```
C: a001 AUTHENTICATE SCRAM-MD5
S: +
```



```
C: AGNocmlzADxwNVIxZTBWTzNLdFZBNEZITDdudWRRQGVsZWFub3Iua
   W5ub3NvZnQuY29tPg==
S: AeYw5Ugm+blpbWFwQGVsZWFub3IuaW5ub3NvZnQuY29tAAA8b1JNa
   nFFekYvL1J5WnhFMlF2cDNzd0BlbGVhbm9yLmlubm9zb2Z0LmNvbT4=
C: 5cZpsA9pODOVwuNU1xmJHA==
S: a001 OK [vJ1FEfRHulPALMwSb/UC9g==] AUTHENTICATE completed
```

For this example, the user "chris", with an empty authorization

identity is using the passphrase "secret stuff". The client nonce is "<p5R1e0VO3KtVA4FHL7nudQ@eleanor.innosoft.com>" and the server nonce is "<oRMjqEzF//RyZxE2Qvp3sw@eleanor.innosoft.com>". The service identity is "imap@eleanor.innosoft.com" and the server extension data is empty. The salt, in hexidecimal, is "01e6 30e5 4826 f9b9". The complete 40 octet verifier stored on the server, in base64, is "AeYw5Ugm+bkHTj2Ouau2II2etDOwYVEXkVsKPP0Q6pV9hbFaweymdg", and in hexidecimal it is "01e6 30e5 4826 f9b9 074e 3d8e b9ab b620 8d9e b433 b061 5117 915b 0a3c fd10 ea95 7d85 b15a c1ec a676".

6. Formal Syntax of SCRAM-MD5 Messages

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

```
client-msg-1     = [authorize-id] NUL authenticate-id NUL [nonce]
                   ;; MUST NOT exceed 1000 octets

server-msg-1     = salt service-id NUL server-ext-data NUL nonce
                   ;; MUST NOT exceed 1000 octets

client-msg-2     = client-proof

server-msg-2     = server-proof

passphrase       = *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

service-id       = service-name "@" server-domain

service-name     = *US-ASCII-PRINT
                   ;; a SASL/GSSAPI service name
```



```
server-domain    = *US-ASCII-PRINT
```

```
                    ;; an internet domain name

server-ext-data  = *(sasl-mech-list / tls-avail / ext-list)

sasl-mech-list   = "SASL:" *(CFWS sasl-mech) CRLF

CFWS             = [CRLF] 1*SPACE

sasl-mech        = 1*20(ALPHA / DIGIT / "-" / "_")

tls-avail        = "TLS:" SPACE "yes"

ext-list         = 1*UTF8PRINT ":" *(CFWS 1*UTF8PRINT) CRLF

nonce            = 8*OCTET

salt             = 8OCTET

client-proof     = 16OCTET

server-proof     = 16OCTET

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
```

7. System Administrator Advice

This section includes advice for system administrators using this mechanism.

SCRAM stores three pieces of information for each user: a salt, a client verifier and server key. The latter two are derived from the salt and the user's passphrase.

The salt prevents global dictionary attacks, similar to the salt used in Unix /etc/passwd files. As the 12 bits of salt in Unix /etc/passwd has proved to be insufficient, SCRAM uses 64 bits of salt. See [SCHNEIER] for a good discussion of salt and dictionary attacks. In a multi-server site, security can be increased by using a different salt on each server.

Although the verifiers used by SCRAM-MD5 have roughly comparable security to those used by current plaintext mechanisms (such as Unix /etc/passwd), it is still very important to keep them secret. Just as tools exist to try common passwords against Unix /etc/passwd files, it is also possible to build such tools for SCRAM-MD5. In addition, once a SCRAM-MD5 verifier is stolen, a passive (undetectable) evesdropper of that user logging in gains the output of step (C) above, which is sufficient to impersonate the user to all services with the same salt. This is far better than current plaintext mechanisms where a passive evesdropper 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-MD5 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-MD5 doesn't protect the integrity or privacy of data exchanged after authentication. Use of TLS [TLS] or a stronger SASL mechanism such as Kerberos is encouraged if this functionality is needed.

8. SCRAM-MD5 Functional Notation

This section is designed to provide a quick understanding of SCRAM-MD5 for those who like functional notation.

+	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 is the same as AU, a single NUL is used instead.
service	is the name of the service and server (NUL terminated)
ext-attr	server extension attributes (NUL terminated)
pass	is the plaintext passphrase
H(x)	is a one-way hash function applied to "x", such as MD5
MAC(x,y)	is a message authentication code (MAC) such as HMAC-MD5 "y" is the key and "x" is the text signed by the key.
salt	is a per-user salt value the server stores
Us	is a unique nonce the server sends to the client
Uc	is a unique nonce the client sends to the server

The following computed values are used in the exchange:

```
client-chal     = AZ + AU + Uc
server-chal     = salt + service + ext-attr + Us
salted-pass     = MAC(salt, pass)
client-key      = H(salted-pass)
client-verifier = H(client-key)
shared-key      = MAC(server-chal + client-chal, client-verifier)
client-proof    = client-key XOR shared-key
server-key      = MAC(salt, salted-pass)
server-proof    = MAC(client-chal + server-chal, server-key)
```

The SCRAM exchange is as follows:

```
(1) client -> server: client-chal
(2) server -> client: server-chal
(3) client -> server: client-proof
(4) server -> client: server-proof
```

The server stores the salt, client-verifier and server-key. It authenticates the client by computing:

H(client-proof XOR shared-key)

after step 3 and comparing it to the stored client-verifier.

The client verifies the server by computing the server-proof directly and comparing.

9. Usage Scenarios

Single Sign-on
: As with CRAM, the intermediate HMAC context from step (B) can be used for a single sign-on client model. SCRAM also adds the ability to store the output of step (B) for the single sign-on only to services with the same salt. If expiration is needed, it can be built into the single sign-on facility. Kerberos is necessary if server-enforced expiration is needed. OTP is necessary if expiration after a fixed number of authentications is necessary.

Simplicity
: CRAM, SCRAM and OTP are all suitable for use in lightweight clients or servers. Kerberos and public key technology are not simple enough.

Frequent Authentications
: Protocols such as IMAP or POP result in frequent authentications by the same user. OTP, by itself, is not suitable for such services as the sequence needs to be reset. OTP with OTP extended responses still does not permit simulatanous connections by the same user.

Location Independent User
: SCRAM, CRAM and OTP are all suitable for users which move between many clients. Public key client authentication will not be suitable for such uses until there is a worldwide smart card standard.

Scalability / Server Efficiency

CRAM and SCRAM are believed to be scalable to large numbers of users. OTP is less scalable due to the need to provide per-user locking and update services to the authentication database.

Client Efficiency
SCRAM has about half the client efficiency of CRAM. OTP's client efficiency varies with the sequence number and it is usually many times slower than SCRAM. Public key systems are even slower.

Limited Server Trust
The server-key may be stored on a central SCRAM server which limits the services an application server is permitted to offer. CRAM and OTP do not offer this functionality.

Multi-Vendor Scenarios
SCRAM does not require cooperation between multiple services on the same server or multiple applications on the same client. This has proved to be a deployment problem for Kerberos.

Proxy Authentication
SCRAM supports proxy authentication by including a separate authentication and authorization identifiers. CRAM lacks this facility.

10. Security Considerations

Security considerations are discussed throughout this document. The security considerations of MD5 [MD5] and HMAC [HMAC] also apply. SCRAM relies primarily on the one-way characteristic of MD5 and HMAC-MD5 for cryptographic security.

An analysis of different attacks follows:

Passive Network Attacks

SCRAM is resistant to replay attacks as long as the appropriate nonce is unique.

SCRAM is not resistant to passive dictionary attacks. User education, passphrase setting policy and TLS [TLS] may be used to protect against such attacks.

SCRAM does not protect against session data evesdropping. TLS [TLS] may be used to protect against this.

Active Network Attacks

SCRAM protects against server impersonation with the optional server authentication. CRAM and OTP do not have this facility.

SCRAM can protect against client impersonation by providing no information beyond the salt to a client which fails verification. Note that in order to prevent revealing the existance of a user to such attackers, the server will have to invent consistant salt between attempts and fail after the second client message. One way to do this would be to store a random secret and use HMAC(user-name, random-secret) to generate consistant salt values.

The server-ext-data may be used to protect against downgrade active attacks (where the active attacker changes the advertised protocol

security services). CRAM and OTP do not have this facility.

SCRAM protects against account hijacking by an active attacker. Use of OTP extended response [OTP-EXT] to reset the sequence is susceptible to account hijacking.

SCRAM does not protect against connection hijacking or corruption. TLS [TLS] or IPAUTH [IPAUTH] may be used to protect against this.

SCRAM does not protect against active denial of service attacks.

Server Attacks

SCRAM verifiers by themselves can not be used to impersonate the user. CRAM verifiers have this weakness.

An attacker which has access to both the SCRAM verifier and a client exchange for a particular user gains the ability to

impersonate that user to other servers using the same salt. TLS [TLS] can make it more difficult to obtain the client exchange, but does not defend against installation of a trojan horse server.

Client Attacks

As SCRAM is designed for user entry of a plaintext passphrase, it is vulnerable to passphrase hijacking by trojan horse clients. OTP [OTP] with an independent OTP calculator can be used to limit the vulnerability to a single session.

11. 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 salt to protect against global dictionary attacks dates back to at least the Unix /etc/password system. There is some discussion of this in [SCHNEIER].

SCRAM combines these techniques. The author of this specification

first proposed this on a public mailing list July 16, 1997.

12. 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 2195, MCI, September 1997.

<ftp://ds.internic.net/rfc/rfc2195.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>

[IPAUTH] Atkinson, "IP Authentication Header", RFC 1826, Naval Research Laboratory, August 1995.

<ftp://ds.internic.net/rfc/rfc1826.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>

[MD5] Rivest, "The MD5 Message Digest Algorithm", RFC 1321, MIT Laboratory for Computer Science, April 1992.

<ftp://ds.internic.net/rfc/rfc1321.txt>

[OTP] Haller, Metz, "A One-Time Password System", RFC 1938, Bellcore, Kaman Sciences Corporation, May 1996.

<ftp://ds.internic.net/rfc/rfc1938.txt>

[OTP-EXT] Metz, "OTP Extended Responses", work in progress.

[POP3] Myers, J., Rose, M., "Post Office Protocol - Version 3", RFC 1939, Carnegie Mellon, Dover Beach Consulting, Inc., May 1996.

<ftp://ds.internic.net/rfc/rfc1939.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.

[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>

13. Author's Address

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

Email: chris.newman@innosoft.com

A. 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.

(8.5) Client sends Y.

and would modify steps (F), (J), (b) and (g) to include the value K. This would result in a protocol safe from passive network attacks. The expense would be reduced performance, the need for a bignum math library and possibly a requirement that an export license be obtained from certain governments (included the United States). This would not defend against active attacks unless an encryption service was added. This may be free of patent restrictions.

TWEKE would be harder to deploy than SCRAM due to the higher math, the use of public key technology and the performance loss.

B. 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 service-id and server-key are included to facilite the remote authentication database service. Otherwise these issues are deferred for future work.

C. Appendix - HMAC-MD5 Sample Source Code

The following sample C source code is calls the source code in [MD5] and is derived from the source code in [HMAC]. It is needed by the SCRAM source code in the next section.

```
/* hmac-md5.h -- HMAC-MD5 functions
 */
```

```
#define HMAC_MD5_SIZE 16
```



```
/* intermediate MD5 context */
typedef struct HMAC_MD5_CTX_s {
    MD5_CTX ictx, octx;
} HMAC_MD5_CTX;

/* One step hmac computation
 *
 * digest may be same as text or key
 */
void hmac_md5(const unsigned char *text, int text_len,
              const unsigned char *key, int key_len,
              unsigned char digest[HMAC_MD5_SIZE]);

/* create intermediate result from key
 */
void hmac_md5_init(HMAC_MD5_CTX *hmac,
                   const unsigned char *key, int key_len);

#define hmac_md5_update(hmac, text, text_len) \
  MD5Update(&(hmac)->ictx, (text), (text_len))

/* finish hmac from intermediate result.
 * Intermediate result is zeroed.
 */
void hmac_md5_final(unsigned char digest[HMAC_MD5_SIZE],
                    HMAC_MD5_CTX *hmac);
```

```
/* hmac-md5.c -- Keyed-Hashing
 *  derived from RFC 2104 by H. Krawczyk, M. Bellare, R.Canetti
 */

#include <stdio.h>
#include <string.h>
#include "md5.h"
#include "hmac-md5.h"

/* MD5 block size */
#define BLOCK_SIZE 64
```

```
void hmac_md5_init(HMAC_MD5_CTX *hmac,
                   const unsigned char *key, int key_len)
{
    unsigned char k_pad[BLOCK_SIZE];    /* padded key */
    int i;
```



```
    /* if key longer than BLOCK_SIZE bytes reset it to MD5(key) */
    if (key_len > BLOCK_SIZE) {
        MD5Init(&hmac->ictx);
        MD5Update(&hmac->ictx, key, key_len);
        MD5Final(k_pad, &hmac->ictx);
        key = k_pad;
        key_len = HMAC_MD5_SIZE;
    }

    /* XOR padded key with inner pad value */
    for (i = 0; i < key_len; i++) {
        k_pad[i] = key[i] ^ 0x36;
    }
    while (i < BLOCK_SIZE) {
        k_pad[i++] = 0x36;
    }

    /* Begin inner MD5 */
    MD5Init(&hmac->ictx);
    MD5Update(&hmac->ictx, k_pad, BLOCK_SIZE);

    /* XOR padded key with outer pad value */
    for (i = 0; i < BLOCK_SIZE; ++i) {
        k_pad[i] ^= (0x36 ^ 0x5c);
    }

    /* Begin outer MD5 */
    MD5Init(&hmac->octx);
    MD5Update(&hmac->octx, k_pad, BLOCK_SIZE);

    /* clean up workspace */
    memset(k_pad, 0, BLOCK_SIZE);
}
```

```
void hmac_md5_final(unsigned char digest[HMAC_MD5_SIZE],
                    HMAC_MD5_CTX *hmac)
{
    /* finish inner MD5 */
    MD5Final(digest, &hmac->ictx);
    /* finish outer MD5 */
    MD5Update(&hmac->octx, digest, HMAC_MD5_SIZE);
    MD5Final(digest, &hmac->octx);
    /* MD5Final zeros context */
}

void hmac_md5(const unsigned char *text, int text_len,
              const unsigned char *key, int key_len,
              unsigned char digest[HMAC_MD5_SIZE])
```



```
{
    HMAC_MD5_CTX hmac;

    hmac_md5_init(&hmac, key, key_len);
    hmac_md5_update(&hmac, text, text_len);
    hmac_md5_final(digest, &hmac);
}
```

D. Appendix - SCRAM sample source code

The following sample source code implements SCRAM itself for both server and client.

Please note the comments marked "/*XXX ... */" as they need to be translated from English to computer readable code.

A client implementation simply calls scram_md5_generate() with the passphrase after receiving the first server reply. The cproof parameter will hold the message to send to the server and the sproof parameter will hold the expected server mutual authentication. A client may also call cram_md5_cred() to turn a passphrase into CRAM/SCRAM credentials for later use in scram_md5_generate().

A server implementation simply calls scram_md5_generate() with the

stored verifier, the second client message and the SCRAM_VERIFY option. Server verifiers are generated by creating a random salt and calling scram_md5_vgen() with either the passphrase or CRAM/SCRAM credentials.

```
/* scram.h -- scram utility functions
 */

/* size of CRAM_MD5 verifier and CRAM_MD5/SCRAM_MD5 credentials */
#define CRAM_MD5_SIZE 32

/* size of SCRAM_MD5 salt and verifier */
#define SCRAM_MD5_SALTSIZE 8
#define SCRAM_MD5_DATASIZE 16

/* SCRAM verifier */
typedef struct SCRAM_MD5_VRFY_s {
    unsigned char salt[SCRAM_MD5_SALTSIZE];
    unsigned char clidata[SCRAM_MD5_DATASIZE];
    unsigned char svrdata[SCRAM_MD5_DATASIZE];
} SCRAM_MD5_VRFY;
```



```
/* prepare CRAM-MD5 credentials/verifier also SCRAM-MD5 credential
 *  buf       -- must be aligned and have room for CRAM_MD5_SIZE
 *  pass      -- passphrase or verifier
 *  passlen   -- len of pass/verifier (0 ok if NUL terminated)
 */
void cram_md5_cred(char *buf, const char *pass, int passlen);

/* generate SCRAM-MD5 verifier
 *  vptr      -- gets result
 *  salt      -- contains salt of SCRAM_MD5_SALTSIZE
 *  pass      -- passphrase or verifier
 *  passlen   -- len of pass/verifier (0 ok if NUL terminated)
 *  plainflag -- 1 = plaintext passphrase,
 *               0 = result of cram_md5_cred()
 *  clientkey -- cache for client proof, usually NULL
 */
void scram_md5_vgen(SCRAM_MD5_VRFY *vptr,
                    const unsigned char *salt,
                    const char *pass, int passlen, int plainflag,
```

```
                        unsigned char *clientkey);

     /* scram secret action type
      */
     #define SCRAM_CREDENTIAL 0 /* generate replies using credentials */
     #define SCRAM_PLAINTEXT  1 /* generate replies using plaintext */
     #define SCRAM_VERIFY     2 /* use SCRAM_MD5_VRFY to verify client,
                                   and generate server reply */

     /* generate or verify SCRAM-MD5
      * input params:
      *  cchal     -- client challenge string
      *  cchallen  -- length of client challenge
      *  schal     -- server challenge string
      *  schallen  -- length of server challenge
      *  secret    -- passphrase, credentials or verifier
      *  secretlen -- length of passphrase (0 ok if NUL terminated)
      *  action    -- see above
      * in/out:
      *  cproof    -- client proof of length SCRAM_MD5_DATASIZE
      * output:
      *  sproof    -- server proof of length SCRAM_MD5_DATASIZE
      * returns:
      *  -2 if params invalid
      *  -1 if verify fails
      *   0 on success
      */
     int scram_md5_generate(const char *cchal, int cchallen,
                            const char *schal, int schallen,
```



```
                            const char *secret, int secretlen,
                            int action, unsigned char *cproof,
                            unsigned char *sproof);


     /* scram.c -- routines for SCRAM-MD5 calculations
      */

     #include <stdio.h>
     #include <stdlib.h>
     #include <string.h>
     #include "md5.h"
```

```
#include "hmac-md5.h"
#include "scram.h"

/* for htonl() and ntohl() */
#include <sys/types.h>
#include <netinet/in.h>

/* MD5 block size */
#define BLOCK_SIZE 64

/* intermediate CRAM context
 *  values stored in network byte order (Big Endian)
 */
typedef struct CRAM_MD5_CTX_s {
    UINT4 istate[4];
    UINT4 ostate[4];
} CRAM_MD5_CTX;

void cram_md5_cred(char *buf, const char *pass, int passlen)
{
    HMAC_MD5_CTX hctx;
    CRAM_MD5_CTX *ctx = (CRAM_MD5_CTX *) buf;

    if (passlen == 0) passlen = strlen(pass);
    hmac_md5_init(&hctx, (const unsigned char *) pass, passlen);
    ctx->istate[0] = htonl(hctx.ictx.state[0]);
    ctx->istate[1] = htonl(hctx.ictx.state[1]);
    ctx->istate[2] = htonl(hctx.ictx.state[2]);
    ctx->istate[3] = htonl(hctx.ictx.state[3]);
    ctx->ostate[0] = htonl(hctx.octx.state[0]);
    ctx->ostate[1] = htonl(hctx.octx.state[1]);
    ctx->ostate[2] = htonl(hctx.octx.state[2]);
    ctx->ostate[3] = htonl(hctx.octx.state[3]);
    memset(&hctx, 0, sizeof (hctx));
}
```



```
/* extract hmac context from CRAM-MD5 credentials
 */
void hmac_cram_md5_init(HMAC_MD5_CTX *hctx, CRAM_MD5_CTX *ctx)
{
    hctx->ictx.state[0] = ntohl(ctx->istate[0]);
```

```
        hctx->ictx.state[1] = ntohl(ctx->istate[1]);
        hctx->ictx.state[2] = ntohl(ctx->istate[2]);
        hctx->ictx.state[3] = ntohl(ctx->istate[3]);
        hctx->octx.state[0] = ntohl(ctx->ostate[0]);
        hctx->octx.state[1] = ntohl(ctx->ostate[1]);
        hctx->octx.state[2] = ntohl(ctx->ostate[2]);
        hctx->octx.state[3] = ntohl(ctx->ostate[3]);
        hctx->ictx.count[0] = hctx->octx.count[0] = BLOCK_SIZE << 3;
        hctx->ictx.count[1] = hctx->octx.count[1] = 0;
    }

    void scram_md5_vgen(SCRAM_MD5_VRFY *vptr,
                        const unsigned char *salt,
                        const char *pass, int passlen, int plainflag,
                        unsigned char *clientkey)
    {
        HMAC_MD5_CTX hctx;

        if (clientkey == NULL) clientkey = vptr->clidata;

        /* get context */
        if (plainflag) {
            if (passlen == 0) passlen = strlen(pass);
            hmac_md5_init(&hctx, (const unsigned char *) pass,
                          passlen);
        } else {
            hmac_cram_md5_init(&hctx, (CRAM_MD5_CTX *) pass);
        }

        /* generate salted passphrase */
        hmac_md5_update(&hctx, salt, SCRAM_MD5_SALTSIZE);
        hmac_md5_final(vptr->clidata, &hctx);

        /* generate server proof */
        hmac_md5(salt, SCRAM_MD5_SALTSIZE, vptr->clidata,
                 sizeof (vptr->clidata), vptr->svrdata);

        /* generate client key and client verifier */
        MD5Init(&hctx.ictx);
        MD5Update(&hctx.ictx, vptr->clidata, sizeof (vptr->clidata));
        MD5Final(clientkey, &hctx.ictx);
        MD5Init(&hctx.ictx);
        MD5Update(&hctx.ictx, clientkey, SCRAM_MD5_DATASIZE);
```

```
    MD5Final(vptr->clidata, &hctx.ictx);

    /* copy salt to verifier */
    if (salt != vptr->salt) {
        memcpy(vptr->salt, salt, SCRAM_MD5_SALTSIZE);
    }
}

int scram_md5_generate(const char *cchal, int cchallen,
                       const char *schal, int schallen,
                       const char *secret, int secretlen,
                       int action, unsigned char *cproof,
                       unsigned char *sproof)
{
    SCRAM_MD5_VRFY verifier, *vptr;
    HMAC_MD5_CTX hctx;
    unsigned char clientkey[HMAC_MD5_SIZE];
    unsigned char sharedkey[HMAC_MD5_SIZE];
    int i, result = 0;

    /* check params */
    if ((action == SCRAM_CREDENTIAL && secretlen != CRAM_MD5_SIZE)
        || (action == SCRAM_VERIFY
            && secretlen != sizeof (verifier))
        || schallen < SCRAM_MD5_SALTSIZE) {
        return (-2);
    }

    /* get verifier */
    if (action == SCRAM_VERIFY) {
        vptr = (SCRAM_MD5_VRFY *) secret;
    } else {
        scram_md5_vgen(&verifier, (const unsigned char *) schal,
                       secret, secretlen, action, clientkey);
        vptr = &verifier;
    }

    /* calculate shared key */
    hmac_md5_init(&hctx, vptr->clidata, sizeof (vptr->clidata));
    hmac_md5_update(&hctx, (unsigned char *) schal, schallen);
    hmac_md5_update(&hctx, (unsigned char *) cchal, cchallen);
    hmac_md5_final(sharedkey, &hctx);

    if (action == SCRAM_VERIFY) {
        /* verify client proof */
        for (i = 0; i < HMAC_MD5_SIZE; ++i) {
            /*XXX: the line which belongs here is omitted due to U.S.
              export regulations, but it exclusive-ors the
```

```
                "sharedkey" with the "cproof" and places the result in
                "clientkey" (see step (c) above) */
            }
            MD5Init(&hctx.ictx);
            MD5Update(&hctx.ictx, clientkey, sizeof (clientkey));
            MD5Final(clientkey, &hctx.ictx);
            if (memcmp(clientkey, vptr->clidata,
                       sizeof (clientkey)) != 0) {
                result = -1;
            }
        } else {
            /* generate client proof */
            for (i = 0; i < HMAC_MD5_SIZE; ++i) {
                /*XXX: the line which belongs here is omitted due to U.S.
                  export regulations, but it exclusive-ors the
                  "sharedkey" with the "clientkey" and places the result
                  in "cproof" (see step (G) above) */
            }
        }

        /* calculate server result */
        if (result == 0) {
            hmac_md5_init(&hctx, vptr->svrdata, HMAC_MD5_SIZE);
            hmac_md5_update(&hctx, (unsigned char *) schal, schallen);
            hmac_md5_update(&hctx, (unsigned char *) cchal, cchallen);
            hmac_md5_final(sproof, &hctx);
        }

        /* cleanup workspace */
        memset(clientkey, 0, sizeof (clientkey));
        memset(sharedkey, 0, sizeof (sharedkey));
        if (vptr == &verifier) memset(&verifier, 0, sizeof (verifier));

        return (result);
    }
```