Network Working Group
Internet-Draft
Intended Status: Proposed Standard
Expires: September 2009

Abhijit Menon-Sen
Oryx Mail Systems GmbH
Chris Newman
Sun Microsystems
Alexey Melnikov
Isode Ltd
Simon Josefsson
SJD AB
March 9, 2009

Salted Challenge Response Authentication Mechanism (SCRAM) as a GSS-API Mechanism

draft-newman-auth-scram-gs2-01.txt

Status of this Memo

Copyright Notice

Abstract

The secure authentication mechanism most widely deployed and used by Internet application protocols is the transmission of clear-text passwords over a channel protected by Transport Layer Security (TLS). There are some significant security concerns with that mechanism, which could be addressed by the use of a challenge response authentication mechanism protected by TLS. Unfortunately, the challenge response mechanisms presently on the standards track all fail to meet requirements necessary for widespread deployment, and have had success only in limited use.

This specification describes an authentication mechanism called the Salted Challenge Response Authentication Mechanism (SCRAM), which addresses the security concerns and meets the deployability requirements. When used in combination with TLS or an equivalent security layer, SCRAM could improve the status-quo for application protocol authentication and provide a suitable choice for a mandatory-to-implement mechanism for future application protocol standards.

The purpose of this document is to describe the general SCRAM protocol, and how it is used in the GSS-API environment. Through GS2, this makes the protocol available in the SASL environment as well.

1.0. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Formal syntax is defined by [RFC5234] including the core rules defined in Appendix B of [RFC5234].

Example lines prefaced by "C:" are sent by the client and ones prefaced by "S:" by the server. If a single "C:" or "S:" label applies to multiple lines, then the line breaks between those lines are for editorial clarity only, and are not part of the actual protocol exchange.

1.1. Terminology

This document uses several terms defined in [RFC4949] ("Internet Security Glossary") including the following: authentication, authentication exchange, authentication information, brute force, challenge-response, cryptographic hash function, dictionary attack, eavesdropping, hash result, keyed hash, man-in-the-middle, nonce, one-way encryption function, password, replay attack and salt. Readers not familiar with these terms should use that glossary as a reference.

Some clarifications and additional definitions follow:

- Authentication information: Information used to verify an identity claimed by a SCRAM client. The authentication information for a SCRAM identity consists of salt, iteration count, the "StoredKey" and "ServerKey" (as defined in the algorithm overview) for each supported cryptographic hash function.

- Authentication database: The database used to look up the authentication information associated with a particular identity. For application protocols, LDAPv3 (see [RFC4510]) is frequently used as the authentication database. For network-level protocols such as PPP or 802.11x, the use of RADIUS is more common.

- Base64: An encoding mechanism defined in [RFC4648] which converts an octet string input to a textual output string which can be easily displayed to a human. The use of base64 in SCRAM is restricted to the canonical form with no whitespace.

- Octet: An 8-bit byte.

- Octet string: A sequence of 8-bit bytes.

- Salt: A random octet string that is combined with a password before applying a one-way encryption function. This value is used to protect passwords that are stored in an authentication database.

1.2. Notation

The pseudocode description of the algorithm uses the following notations:

- ":=": The variable on the left hand side represents the octet string resulting from the expression on the right hand side.

- "+": Octet string concatenation.

- "[ ]": A portion of an expression enclosed in "[" and "]" may not be included in the result under some circumstances. See the associated text for a description of those circumstances.

- HMAC(key, str): Apply the HMAC keyed hash algorithm (defined in [RFC2104]) using the octet string represented by "key" as the key and the octet string "str" as the input string. The size of the result is the hash result size for the hash function in use. For example, it is 20 octets for SHA-1 (see [RFC3174]).

- H(str): Apply the cryptographic hash function to the octet string "str", producing an octet string as a result. The size of the result depends on the hash result size for the hash function in use.

- XOR: Apply the exclusive-or operation to combine the octet string on the left of this operator with the octet string on the right of this operator. The length of the output and each of the two inputs will be the same for this use.

- Hi(str, salt):

```
U0   := HMAC(str, salt + INT(1))
U1   := HMAC(str, U0)
U2   := HMAC(str, U1)
...
Ui-1 := HMAC(str, Ui-2)
Ui   := HMAC(str, Ui-1)

Hi := U0 XOR U1 XOR U2 XOR ... XOR Ui
```

where "i" is the iteration count, "+" is the string concatenation operator and INT(g) is a four-octet encoding of the integer g, most significant octet first.

This is, essentially, PBKDF2 [RFC2898] with HMAC() as the PRF and with dkLen == output length of HMAC() == output length of H().

2. Introduction

This specification describes an authentication mechanism called the Salted Challenge Response Authentication Mechanism (SCRAM) which addresses the requirements necessary to deploy a challenge-response mechanism more widely than past attempts. When used in combination with Transport Layer Security (TLS, see [TLS]) or an equivalent security layer, a mechanism from this family could improve the status-quo for application protocol authentication and provide a suitable choice for a mandatory-to-implement mechanism for future application protocol standards.

<<For simplicity, this mechanism does not presently include negotiation of a security layer. It is intended to be used with an external security layer such as that provided by TLS or SSH.>>

SCRAM provides the following protocol features:

- The authentication information stored in the authentication database is not sufficient by itself to impersonate the client. The information is salted to prevent a pre-stored dictionary attack if the database is stolen.

- The server does not gain the ability to impersonate the client to other servers (with an exception for server-authorized proxies).

- The mechanism permits the use of a server-authorized proxy without requiring that proxy to have super-user rights with the back-end server.

- A standard attribute is defined to enable storage of the authentication information in LDAPv3 (see [RFC4510]).

- Both the client and server can be authenticated by the protocol.

For an in-depth discussion of why other challenge response mechanisms are not considered sufficient, see appendix A. For more information about the motivations behind the design of this mechanism, see appendix B.

Comments regarding this draft may be sent either to the ietf-sasl@imc.org mailing list or to the authors.

3. SCRAM Algorithm and Protocol Overview

Note that this section omits some details, such as client and server nonces. See Section 5 for more details.

To begin with, the client is in possession of a username and password. It sends the username to the server, which retrieves the corresponding authentication information, i.e. a salt, StoredKey, ServerKey and the iteration count i. (Note that a server implementation may chose to use the same iteration count for all account.) The server sends the salt and the iteration count to the client, which then computes the following values and sends a ClientProof to the server:

```
SaltedPassword  := Hi(password, salt)
ClientKey       := H(SaltedPassword)
StoredKey       := H(ClientKey)
AuthMessage     := client-first-message + "," +
                   server-first-message + "," +
                   client-final-message-without-proof
ClientSignature := HMAC(StoredKey, AuthMessage)
ClientProof     := ClientKey XOR ClientSignature
ServerKey       := HMAC(SaltedPassword, salt)
ServerSignature := HMAC(ServerKey, AuthMessage)

ScramKey        := HMAC(ClientKey, AuthMessage)
MicKey          := HMAC(ScramKey, "SCRAM MIC constant")
ClientMic       := HMAC(MicKey, client-gs2-to-be-protected)
ServerMic       := HMAC(MicKey, server-gs2-to-be-protected)
```

The server authenticates the client by computing the ClientSignature, exclusive-ORing that with the ClientProof to recover the ClientKey and verifying the correctness of the ClientKey by applying the hash function and comparing the result to the StoredKey. If the ClientKey is correct, this proves that the client has access to the user's password.

Similarly, the client authenticates the server by computing the ServerSignature and comparing it to the value sent by the server. If the two are equal, it proves that the server had access to the user's ServerKey.

Once authentication is successful both the client and the server are in possesion of the ClientKey. The ClientKey is used to construct

the shared SCRAM key (ScramKey), which is then used to produce the MicKey. The MicKey is used to verify channel binding and authorization identity by the server, and to confirm that the channel binding information was verified by the client.

The AuthMessage is computed by concatenating messages from the authentication exchange. client-gs2-to-be-protected and server-gs2-to-be-protected are also parts of the authentication exchange. The format of these messages is defined in the Formal Syntax section.

4. Use of SCRAM in GSS-API and SASL

The SCRAM protocol defined in this document is not specific to a particular authentication framework, such as GSS-API, SASL or EAP. The purpose of this section is to describe how the SCRAM protocol is implemented within a particular framework. The focus here is on GSS-API and SASL. If desirable, it may be possible to write similar mappings for other authentication frameworks in the future (e.g., EAP).

4.1 Use of SCRAM in GSS-API

Context establishment consists of sending and receiving the SCRAM Authentication Exchange protocol. The GSS-API OID allocated for SCRAM is 1.3.6.1.4.1.11591.4.2. The PROT_READY should be set after the authentication exchange completed. When the context has been established, message integrity services through GSS_Wrap/GSS_Unwrap are implemented by using the ClientMic and ServerMic keys derived from the authentication protocol.

<<describe syntax of gss_wrap/gss_unwrap output better>>

4.2 Use of SCRAM in SASL via GS2.

Through GS2, each GSS-API mechanism is supported in SASL. To use SCRAM in SASL, we must derive the SASL mechanism name using the algorithm described in GS2. The DER encoding of the OID is (in hex) 06 09 2B 06 01 04 01 DA 47 04 02. The SHA-1 hash is 29 06 29 12 AB 25 83 CD 02 92 1B 4E 2D D8 6A 40 CD D0 5D C2. Convert the first ten octets to binary, and re-group them in groups of 5, and convert them back to decimal, which results in these computations:

```
hex:
29 06 29 12 AB 25 83 CD 02 92

binary:
00101001 00000110 00101001 00010010 10101011
00100101 10000011 11001101 00000010 10010010

binary in groups of 5:
00101 00100 00011 00010 10010 00100 10101 01011
00100 10110 00001 11100 11010 00000 10100 10010

decimal of each group:
5 4 3 2 18 4 21 11 4 22 1 28 26 0 20 18

base32 encoding:
F E D C S E V L E W B 4 2 A U S
```

The last step translate each decimal value using table 3 in Base32 [RFC4648]. Thus the SASL mechanism name for SCRAM is "GS2-FEDCSEVLEWB42AUS".

The wire syntax of SCRAM in SASL is described normatively in [GS2], based on the wire format describe above for GSS-API.

5. SCRAM Authentication Exchange

SCRAM is a text protocol where the client and server exchange messages containing one or more attribute-value pairs separated by commas. Each attribute has a one-letter name. The messages and their attributes are described in section 5.1, and defined in the Formal Syntax section.

This is a simple example of a authentication exchange:

```
C: n=Chris Newman,r=ClientNonce[^A]
S: r=ClientNonceServerNonce,s=PxR/wv+epq,i=128[^A]
C: r=ClientNonceServerNonce,p=WxPv/siO5l+qxN4[^A]mic=<<base64>>,
   d=qop=none
S: v=WxPv/siO5l+qxN4[^A]mic=<<base64>>,d=qop=none
```

<< oidgunk required at the beginning of the first client message? However we can assume GS2 compression as discuss on the mailing list >>

<<+cbgood in the last server step implies that the channel binding was verified. But is it optional?>>

With channel bindings this might look like:

```
C: n=Chris Newman,r=ClientNonce[^A]
S: r=ClientNonceServerNonce,s=PxR/wv+epq,i=128[^A]
C: r=ClientNonceServerNonce,p=WxPv/siO5l+qxN4[^A]mic=<<base64>>,
   d=qop=none,cbqop=none,c=<<base64>>
S: v=WxPv/siO5l+qxN4[^A]mic=<<base64>>,d=qop=none+cbgood
```

Note that [^A] here represents 1 octet with value %x01.

<<This text needs to be updated to match ABNF:>>

First, the client sends a message containing the username, and a random, unique nonce. In response, the server sends the user's iteration count i, the user's salt, and appends its own nonce to the client-specified one. The client then responds with the same nonce and a ClientProof computed using the selected hash function as explained earlier. In this step the client can also include an optional authorization identity. <<The server verifies the nonce and the proof, verifies that the authorization identity (if supplied by the client in the second message) is authorized to act as the authentication identity, and, finally, it responds with a ServerSignature, concluding the authentication exchange>>. <<The client then authenticates the server by computing the ServerSignature and comparing it to the value sent by the server.>> If the two are different, the client MUST consider the authentication exchange to be unsuccessful and it might have to drop the connection.

5.1 SCRAM attributes

This section describes the permissible attributes, their use, and the format of their values. All attribute names are single US-ASCII letters and are case-sensitive.

- a: This optional attribute specifies an authorization identity. A client may include it in its second message to the server if it wants to authenticate as one user, but subsequently act as a different user. This is typically used by an administrator to perform some management task on behalf of another user, or by a proxy in some situations (<<see appendix A for more details>>).

  Upon the receipt of this value the server verifies its correctness and makes the authorization decision. Failed verification results in failed authentication exchange.

If this attribute is omitted (as it normally would be), or specified with an empty value, the authorization identity is assumed to be derived from the username specified with the (required) "n" attribute.

The server always authenticates the user specified by the "n" attribute. If the "a" attribute specifies a different user, the server associates that identity with the connection after successful authentication and authorization checks.

The syntax of this field is the same as that of the "n" field with respect to quoting of %x01, '=' and ','.

- n: This attribute specifies the name of the user whose password is used for authentication. A client must include it in its first message to the server. If the "a" attribute is not specified (which would normally be the case), this username is also the identity which will be associated with the connection subsequent to authentication and authorization.

  Before sending the username to the server, the client MUST prepare the username using the "SASLPrep" profile [SASLPrep] of the "stringprep" algorithm [RFC3454]. If the preparation of the username fails or results in an empty string, the client SHOULD abort the authentication exchange (*).

  (*) An interactive client can request a repeated entry of the username value.

  Upon receipt of the username by the server, the server SHOULD prepare it using the "SASLPrep" profile [SASLPrep] of the "stringprep" algorithm [RFC3454]. If the preparation of the username fails or results in an empty string, the server SHOULD abort the authentication exchange.

  The characters %x01, ',' or '=' in usernames are sent as '=01', '=2C' and '=3D' respectively. If the server receives a username which contains '=' not followed by either '01', '2C' or '3D', then the server MUST fail the authentication.

- m: This attribute is reserved for future extensibility. In this version of SCRAM, its presence in a client or a server message MUST cause authentication failure when the attribute is parsed by the other end.

- r: This attribute specifies a sequence of random printable characters excluding ',' which forms the nonce used as input to the hash function. No quoting is applied to this string (unless

the binding of SCRAM to a particular protocol states otherwise). As described earlier, the client supplies an initial value in its first message, and the server augments that value with its own nonce in its first response. It is important that this be value different for each authentication. The client MUST verify that the initial part of the nonce used in subsequent messages is the same as the nonce it initially specified. The server MUST verify that the nonce sent by the client in the second message is the same as the one sent by the server in its first message.

- c: This optional attribute specifies base64-encoded channel-binding data. It is sent by the client in the second step. If specified by the client, if the server supports the specified channel binding type and if the server can't verify it, then the server MUST fail the authentication exchange. Whether this attribute is included, and the meaning and contents of the channel-binding data depends on the external security layer in use. This is necessary to detect a man-in-the-middle attack on the security layer.

- s: This attribute specifies the base64-encoded salt used by the server for this user. It is sent by the server in its first message to the client.

- i: This attribute specifies an iteration count for the selected hash function and user, and must be sent by the server along with the user's salt.

  Servers SHOULD announce a hash iteration-count of at least 128.

- p: This attribute specifies a base64-encoded ClientProof. The client computes this value as described in the overview and sends it to the server.

- v: This attribute specifies a base64-encoded ServerSignature. It is sent by the server in its final message, and may be used by the client to verify that the server has access to the user's authentication information. This value is computed as explained in the overview.

6. Formal Syntax

The following syntax specification uses the Augmented Backus-Naur Form (ABNF) notation as specified in [RFC5234]. "UTF8-2", "UTF8-3" and "UTF8-4" non-terminal are defined in [UTF-8].

```
attr-val        = ALPHA "=" value
```

```
value           = *(value-char)

value-safe-char = %02-2B / %2D-3C / %3E-7F /
                  UTF8-2 / UTF-3 / UTF8-4
                  ;; UTF8-char except NUL, %x01 (CTRL+A), "=",
                  ;; and ",".

value-char      = value-safe-char / "="

base64-char     = ALPHA / DIGIT / "/" / "+"

base64-4        = 4*4(base64-char)

base64-3        = 3*3(base64-char) "="

base64-2        = 2*2(base64-char) "=="

base64          = *(base64-4) [base64-3 / base64-2]

posit-number = (%x31-39) *DIGIT
                  ;; A positive number

saslname        = 1*(value-safe-char / "=01" / "=2C" / "=3D")
                  ;; Conforms to <value>

authzid         = "a=" saslname
                  ;; Protocol specific.

username        = "n=" saslname
                  ;; Usernames are prepared using SASLPrep.

reserved-mext  = "m=" 1*(value-char)
                  ;; Reserved for signalling mandatory extensions.
                  ;; The exact syntax will be defined in
                  ;; the future.

channel-binding = "c=" base64

proof           = "p=" base64

nonce           = "r=" c-nonce [s-nonce]
                  ;; Second part provided by server.

c-nonce         = value

s-nonce         = value

salt            = "s=" base64
```

```
   verifier        = "v=" base64
                     ;; base-64 encoded ServerSignature.

   iteration-count = "i=" posit-number

   delim = %x01

   client-first-message =
                     scram-client-first-message delim

   server-first-message =
                     scram-server-first-message delim

   client-final-message =
                     scram-client-final-message-without-proof ","
                     proof delim
                     gss-mic client-gss-wrap
                     ;; <<GS2 extensions omitted after "gss-mic">>

   server-final-message =
                     scram-server-final-message delim
                     gss-mic server-gss-wrap
                     ;; <<GS2 extensions omitted after "gss-mic">>

   gss-mic = "mic=" base64 ","
                     ;; base-64 encoding of ClientMic
                     ;; for the client and ServerMic
                     ;; for the server

   client-gss-wrap = "d=" client-gs2-to-be-protected
                     ;; A particular case of <gss-wrap>

   client-gs2-to-be-protected = "qop=none" [",cbqop=none," channel-
                     binding]
                     ["," authzid]
                     ;; A particular case of <gs2-to-be-protected>

   server-gss-wrap = "d=" server-gs2-to-be-protected
                     ;; A particular case of <gss-wrap>

   server-gs2-to-be-protected = "qop=none" [ "+cbgood" ]
                     ;; A particular case of <gs2-to-be-protected>
                     ;; Note that "+cbgood" is included if
                     ;; channel binding verification succeeded.



   gss-wrap = "d=" gs2-to-be-protected
```

```
      gs2-to-be-protected = qop ["," maxbuf]
                        ["," cbqop "," channel-binding] ["," authzid]
                        ;; <<GS2- specific extensions -
                        ;; "["," extensions]"
                        ;; omitted at the end>>

      qop = "qop=" qopvalue *( "+" qopvalue)

      qopvalue = "none" ; no security layer
                            / "integ" ; integrity protection
                            / "conf" ; confidentiality protection
                            / "cbgood" ; channel binding validated
                                       ; (server to client)

      maxbuf = "maxbuf=" posit-number

      cbqop = "cbqop=" qopvalue *( "+" qopvalue)
                        ;; QOPs that can be used if channel binding
                        ;; succeeds




      scram-client-first-message =
                        [reserved-mext ","] username "," nonce
                        ["," extensions]

      scram-server-first-message =
                        [reserved-mext ","] nonce "," salt ","
                        iteration-count ["," extensions]

      scram-client-final-message-without-proof =
                        nonce ["," extensions]
                        ;; <<Note, we used to have GSS-API
                        ;; channel-binding here, but the GS2
                        ;; spec says it MUST be NULL>>

      scram-server-final-message =
                        verifier ["," extensions]

      extensions = attr-val *("," attr-val)
                        ;; All extensions are optional,
                        ;; i.e. unrecognized attributes
                        ;; not defined in this document
                        ;; MUST be ignored.
```

7. Security Considerations

If the authentication exchange is performed without a strong security layer, then a passive eavesdropper can gain sufficient information to mount an offline dictionary or brute-force attack which can be used to recover the user's password. The amount of time necessary for this attack depends on the cryptographic hash function selected, the strength of the password and the iteration count supplied by the server. An external security layer with strong encryption will prevent this attack.

If the external security layer used to protect the SCRAM exchange uses an anonymous key exchange, then the SCRAM channel binding mechanism can be used to detect a man-in-the-middle attack on the security layer and cause the authentication to fail as a result. However, the man-in-the-middle attacker will have gained sufficient information to mount an offline dictionary or brute-force attack. For this reason, SCRAM includes the ability to increase the iteration count over time.

If the authentication information is stolen from the authentication database, then an offline dictionary or brute-force attack can be used to recover the user's password. The use of salt mitigates this attack somewhat by requiring a separate attack on each password. Authentication mechanisms which protect against this attack are available (e.g., the EKE class of mechanisms), but the patent situation is presently unclear.

If an attacker obtains the authentication information from the authentication repository and either eavesdrops on one authentication exchange or impersonates a server, the attacker gains the ability to impersonate that user to all servers providing SCRAM access using the same hash function, password, iteration count and salt. For this reason, it is important to use randomly-generated salt values.

If the server detects (from the value of the client-specified "h" attribute) that both endpoints support a stronger hash function that the one the client actually chooses to use, then it SHOULD treat this as a downgrade attack and reject the authentication attempt.

A hostile server can perform a computational denial-of-service attack on clients by sending a big iteration count value.

8. IANA considerations

None.

9. Acknowedgements

The authors would like to thank Dave Cridland for his contributions to this document.

10. Normative References

[RFC4648] Josefsson, "The Base16, Base32, and Base64 Data Encodings", RFC 4648, SJD, October 2006.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC2104] Krawczyk, Bellare, Canetti, "HMAC: Keyed-Hashing for Message Authentication", IBM, February 1997.

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

[RFC3174] Eastlake, Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, Motorola, September 2001

[RFC5234] Crocker, Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 5234, January 2008.

[RFC4422] Melnikov, Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, Isode Limited, June 2006.

[SASLPrep] Zeilenga, K., "SASLprep: Stringprep profile for user names and passwords", RFC 4013, February 2005.

[RFC3454] Hoffman, P., Blanchet, M., "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.

[SASL-GS2] Josefsson, S., "Using GSS-API Mechanisms in SASL: The GS2 Mechanism Family", work in progress, draft-ietf-sasl-gs2-10.txt, July 2008. <<Can we avoid making this a normative reference?>>

11. Informative References

[RFC2195] Klensin, Catoe, Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", RFC 2195, MCI, September 1997.

[RFC2202] Cheng, Glenn, "Test Cases for HMAC-MD5 and HMAC-SHA-1", RFC 2202, IBM, September 1997

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

[TLS] Dierks, Rescorla, "The Transport Layer Security (TLS) Protocol, Version 1.2", RFC 5246, August 2008.

[RFC4949] Shirey, "Internet Security Glossary, Version 2", RFC 4949, FYI 0036, August 2007.

[RFC4086] Eastlake, Schiller, Crocker, "Randomness Requirements for Security", RFC 4086, BCP 0106, Motorola Laboratories, June 2005.

[RFC4510] Zeilenga, "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map", RFC 4510, June 2006.

[DIGEST-MD5] Leach, P. and C. Newman , "Using Digest Authentication as a SASL Mechanism", RFC 2831, May 2000. <<Also draft-ietf-sasl-rfc2831bis-12.txt>>

[DIGEST-HISTORIC] Melnikov, "Moving DIGEST-MD5 to Historic", work in progress, draft-ietf-sasl-digest-to-historic-00.txt, July 2008

[CRAM-HISTORIC] Zeilenga, "CRAM-MD5 to Historic", work in progress, draft-ietf-sasl-crammd5-to-historic-00.txt, November 2008.

[PLAIN] Zeilenga, "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism" RFC 4616, August 2006.

12. Authors' Addresses

Abhijit Menon-Sen
Oryx Mail Systems GmbH

Email: ams@oryx.com

Alexey Melnikov
Isode Ltd

EMail: Alexey.Melnikov@isode.com

Chris Newman
Sun Microsystems
1050 Lakes Drive
West Covina, CA 91790
USA

Email: chris.newman@sun.com

Simon Josefsson
Email: simon@josefsson.org

Appendix A: Other Authentication Mechanisms

The DIGEST-MD5 [DIGEST-MD5] mechanism has proved to be too complex to implement and test, and thus has poor interoperability. The security layer is often not implemented, and almost never used; everyone uses TLS instead. For a more complete list of problems with DIGEST-MD5 which lead to the creation of SCRAM see [DIGEST-HISTORIC].

The CRAM-MD5 SASL mechanism, while widely deployed has also some problems, in particular it is missing some modern SASL features such as support for internationalized usernames and passwords, support for passing of authorization identity, support for channel bindings. It also doesn't support server authentication. For a more complete list of problems with CRAM-MD5 see [CRAM-HISTORIC].

The PLAIN [PLAIN] SASL mechanism allows a malicious server or eavesdropper to impersonate the authenticating user to any other server for which the user has the same password. It also sends the password in the clear over the network, unless TLS is used. Server authentication is not supported.

Appendix B: Design Motivations

The following design goals shaped this document. Note that some of the goals have changed since the initial version of the document.

The SASL mechanism has all modern SASL features: support for

internationalized usernames and passwords, support for passing of authorization identity, support for channel bindings.

Both the client and server can be authenticated by the protocol.

The authentication information stored in the authentication database is not sufficient by itself to impersonate the client.

<<The server does not gain the ability to impersonate the client to other servers (with an exception for server-authorized proxies).>>

The mechanism is extensible, but [hopefully] not overengineered in this respect.

Easier to implement than DIGEST-MD5 in both clients and servers.

On the wire compatibility with GS2 [SASL-GS2].

Appendix C: SCRAM Examples

<<To be written.>>

(RFC Editor: Please delete everything after this point)

Open Issues

- The appendices need to be written.

- Should the server send a base64-encoded ServerSignature for the value of the "v" attribute, or should it compute a ServerProof the way the client computes a ClientProof?