Salted Challenge Response (SCRAM) SASL Mechanism

draft-newman-auth-scram-08.txt

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

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

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


This Internet-Draft expires in July 2009.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.
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 a family of authentication mechanisms 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, 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.

1. 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.

- \( Hi(str, salt) \):

  \[
  \begin{align*}
  U_0 & := \text{HMAC}(str, salt \ || \ \text{INT}(1)) \\
  U_1 & := \text{HMAC}(str, U_0) \\
  U_2 & := \text{HMAC}(str, U_1) \\
  \cdots \\
  U_{i-1} & := \text{HMAC}(str, U_{i-2}) \\
  U_i & := \text{HMAC}(str, U_{i-1})
  \end{align*}
  \]

  \[ Hi := U_0 \oplus U_1 \oplus U_2 \oplus \cdots \oplus U_i \]

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

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

2. Introduction

This specification describes a family of authentication mechanisms 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 family of 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 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. (Note that a server implementation may chose to use the same iteration count for all account.) The server sends the salt and an iteration count to the client, which then computes the following values and sends a ClientProof to the server:

\[
\begin{align*}
\text{SaltedPassword} &= \text{Hi}(\text{password}, \text{salt}) \\
\text{ClientKey} &= \text{H}((\text{SaltedPassword}) \\
\text{StoredKey} &= \text{H}((\text{ClientKey}) \\
\text{AuthMessage} &= \text{client-first-message} + "," + \\
& \quad \text{server-first-message} + "," + \\
& \quad \text{final-client-message-without-proof} \\
\text{ClientSignature} &= \text{HMAC}((\text{StoredKey}, \text{AuthMessage}) \\
\text{ClientProof} &= \text{ClientKey} \oplus \text{ClientSignature} \\
\text{ServerKey} &= \text{HMAC}((\text{SaltedPassword}, \text{salt})
\end{align*}
\]
ServerSignature := HMAC(ServerKey, AuthMessage)

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 SaltedPassword.

The AuthMessage is computed by concatenating messages from the authentication exchange. The format of these messages is defined in the Formal Syntax section.

4. SCRAM mechanism names

A SCRAM mechanism name is a string "SCRAM-HMAC-" followed by the uppercased name of the underlying hashed function taken from the IANA "Hash Function Textual Names" registry (see http://www.iana.org).

For interoperability, all SCRAM clients and servers MUST implement the SCRAM-HMAC-SHA-1 authentication mechanism, i.e. an authentication mechanism from the SCRAM family that uses the SHA-1 hash function as defined in [RFC3174].

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 SCRAM-HMAC-SHA-1 authentication exchange:

    C: n=Chris Newman,r=ClientNonce
    S: r=ClientNonceServerNonce,s=PxR/wv+epq,i=128
    C: r=ClientNonceServerNonce,p=WxPv/siO5l+qxN4
    S: v=WxPv/siO5l+qxN4

First, the client sends a message containing the username, an
optional authorization identity 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. The server verifies the nonce and the proof, verifies that the authorization identity (if supplied by the client in the initial 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 first 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 according to the used SASL protocol profile. 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 ‘=’ 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 ',' or '=' in usernames are sent as '=2C' and '=3D' respectively. If the server receives a username which contains '=' not followed by either '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 may be sent by either the client or the server. If specified, the authentication MUST fail unless the value is successfully verified. 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.

For SCRAM-HMAC-SHA-1 SASL mechanism 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].

```
generic-message = attr-val *("," attr-val)

attr-val = ALPHA "=" value

value = *(value-char)

value-safe-char = %01-2B / %2D-3C / %3E-7F / UTF8-2 / UTF-3 / UTF8-4

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]

saslname = 1*(value-safe-char / ";=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=" (%x31-39) *DIGIT
;; A positive number

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

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

client-final-message-without-proof = nonce "," channel-binding ["," extensions]
client-final-message =
    client-final-message-without-proof "," proof

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

extensions =
    <One or more attribute=value separated by commas.
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

IANA is requested to add the following entry to the SASL Mechanism registry established by [RFC4422]:

To: iana@iana.org
Subject: Registration of a new SASL mechanism SCRAM-HMAC-SHA-1

SASL mechanism name (or prefix for the family): SCRAM-HMAC-SHA-1
Security considerations: Section 7 of [RFCXXXX]
Published specification (optional, recommended): [RFCXXXX]
Person & email address to contact for further information:
IETF SASL WG <ietf-sasl@imc.org>
Intended usage: COMMON
Owner/Change controller: IESG <iesg@ietf.org>

Note that even though this document defines a family of SCRAM-HMAC mechanisms, it doesn’t register a family of SCRAM-HMAC mechanisms in the SASL Mechanisms registry. IANA is requested to prevent future registrations of SASL mechanisms starting with SCRAM-HMAC- without consulting the SASL mailing list <ietf-sasl@imc.org> first.

Note to future SCRAM-HMAC mechanism designers: each new SCRAM-HMAC SASL mechanism MUST be explicitly registered with IANA and MUST comply with SCRAM-HMAC mechanism naming convention defined in Section 4 of this document.

9. Acknowledgements

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


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


11. Informative References


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
Appendix A: Other Authentication Mechanisms

The 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 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 can be used with Unix hashed password databases.

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

- Easier to implement than DIGEST-MD5 in both clients and servers.
Appendix C: SCRAM Examples

<<To be written.>>
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?

Changes since -07

Updated References.

Clarified purpose of the m= attribute.

Fixed a problem with authentication/authorization identity’s ABNF not allowing for some characters.

Updated ABNF for nonce to show client-generated and server-generated parts.

Only register SCRAM-HMAC-SHA-1 with IANA and require explicit registrations of all other SCRAM-HMAC- mechanisms.

Changes since -06

Removed hash negotiation from SCRAM and turned it into a family of SASL mechanisms.

Start using "Hash Function Textual Names" IANA registry for SCRAM mechanism naming.

Fixed definition of Hi(str, salt) to be consistent with [RFC2898].

Clarified extensibility of SCRAM: added m= attribute (for future mandatory extensions) and specified that all unrecognized attributes must be ignored.

Changes since -05

Changed the mandatory to implement hash algorithm to SHA-1 (as per
WG consensus).

Added text about use of SASLPrep for username canonicalization/validation.

Clarified that authorization identity is canonicalized/verified according to SASL protocol profile.

Clarified that iteration count is per-user.

Clarified how clients select the authentication function.

Added IANA registration for the new mechanism.

Added missing normative references (UTF-8, SASLPrep).

Various editorial changes based on comments from Hallvard B Furuseth, Nico William and Simon Josefsson.

Changes since -04

- Update Base64 and Security Glossary references.
- Add Formal Syntax section.
- Don’t bother with "v=".
- Make MD5 mandatory to implement. Suggest i=128.

Changes since -03

- Seven years have passed, in which it became clear that DIGEST-MD5 suffered from unacceptably bad interoperability, so SCRAM-MD5 is now back from the dead.
- Be hash agnostic, so MD5 can be replaced more easily.
- General simplification.