SASL: Simple Authentication and Security Layer

Version 9.4.0.5

Ryan Culpepper <ryanc@racket-lang.org>

November 15, 2026

This library provides implementations of some SASL (RFC 4422) mechanisms. Currently only client support is implemented; future versions of this library may add server support.

1 SASL Introduction

This library implements non-trivial (multi-message) authentication mechanisms using *SASL protocol contexts*. Since in general SASL is embedded as a sub-protocol of some application protocol, this library does not handle I/O directly. Instead, the user is responsible for transferring messages between the SASL context and the application protocol.

Note that some SASL authentication mechanisms leave the communication of authentication success or failure to the application layer. Even when a mechanism normally communicates the result of authentication, some applications choose to convey failure at the application layer.

The following is a sketch of a typical embedding of SASL in an application protocol:

1. server → client: ... Hello. I understand the following SASL mechanisms: `SCRAM-SHA-1` and `CRAM-MD5`
2. client → server: I choose `SCRAM-SHA-1`. My initial SASL message is "<*initial `SCRAM-SHA-1` message*>".
3. server → client: My SASL response is "<*`SCRAM-SHA-1` response*>".
4. client → server: My final SASL message is "*final* <*`SCRAM-SHA-1` message*>".
5. server → client: My final SASL response is "<*final `SCRAM-SHA-1` response*>".
6. client → server: Great! Let's get to work....

In particular, the application layer advertises and selects SASL mechanisms, embeds SASL messages using some application-specific framing, and resumes after authentication is complete.

2 SASL Protocol Contexts

```
(require sasl)
```
package: sasl-lib

```
(sasl-ctx? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a SASL protocol context, `#f` otherwise.

```
(make-sasl-ctx aux out next) → sasl-ctx?
  aux : any/c
  out : (or/c #f bytes? string?)
  next : sasl-next/c
```

Returns a custom SASL protocol context. Use this procedure when you need to implement a SASL mechanism that is not directly supported by this library.

The *aux* argument is an implementation-dependent value that is passed in calls to the context's transition procedures.

The *out* argument provides the initial outgoing message to be sent. If its value is `#f`, then the state of the context is one in which it does not send an initial message. Otherwise, the state of the context is `'send/receive` or `'send/done`, depending on the value of *next*.

The *next* argument transitions the context into the next state when a message is received. If its value is `'done`, the context is transitioned into a state where it may not receive new messages. If its value is a procedure, then the next time a message is received, that procedure will be called with *aux* and the received message as arguments. Its two return values will be used as the value of the next outgoing message, and the next transition procedure, respectively.

When *next* raises an exception, the context is automatically transitioned into the `'error` state and an `exn:fail:sasl:fatal?` exception is raised.

Added in version 1.3 of package `sasl-lib`.

```
(sasl-next-message ctx) → (or/c string? bytes?)
  ctx : sasl-ctx?
```

Returns the next outgoing message to be sent. Subsequent calls to `sasl-next-message` return the same message until the outgoing message is updated after a call to `sasl-receive-message`.

This function may be called only when `(sasl-state ctx)` is `'send/receive` or `'send/done`; otherwise, an exception is raised.

```
(sasl-receive-message ctx message) → void?
  ctx : sasl-ctx?
  message : (or/c string? bytes?)
```

Update the SASL context with a newly received *message*.

If *message* represents progress or success, then *ctx* is updated and subseqent calls to sasl-next-message return a new message (or fail, if the protocol is done).

If *message* indicates authentication failure or if *message* is ill-formed or invalid, an exception is raised and *ctx* enters a permanent error state (see sasl-state). The user must take appropriate action after either kind of failure. For example, upon authentication failure the client might close the connection and try again with different credentials.

This function may be called only when (sasl-state *ctx*) is 'receive or 'send/receive; otherwise, an exception is raised.

```
(sasl-state ctx)
 → (or/c 'receive 'send/receive 'send/done 'done 'error)
  ctx : sasl-ctx?
```

Returns a symbol indicating the state that *ctx* is in with respect to its protocol. The number of states is due to the following factors: the initial SASL message may be sent from the client or the server (depending on the mechanism); the final SASL message mey be sent from the client or the server (depending on the mechanism); and the SASL context doesn't know whether an outgoing message has been forwarded to the application layer and sent.

The possible states consist of the following:

- 'receive: the protocol starts with *ctx* receiving a message
- 'send/receive: send the current outgoing message (sasl-next-message) if it hasn't already been sent, then receive
- 'send/done: send the current outgoing message (sasl-next-message) if it hasn't already been sent, then the SASL protocol is done
- 'done: the SASL protocol ended with the last received message
- 'error: a fatal error occurred

```
sasl-next/c : contract?
 = (or/c 'done
         (-> any/c
             (or/c bytes? string?)
             (values (or/c #f bytes? string?) sasl-next/c)))
```

The contract for custom SASL mechanism state transition procedures.

```
(struct exn:fail:sasl:fatal exn:fail (msg)
    #:extra-constructor-name make-exn:fail:sasl:fatal)
  msg : string?
```

The exception that is raised by SASL contexts when a fatal error occurs.

3 SASLprep

```
(require sasl/saslprep)
```
package: `sasl-lib`

```
(saslprep s
          [#:allow-unassigned? allow-unassigned?]) → string?
  s : string?
  allow-unassigned? : boolean? = #f
```

Implements the SASLprep (RFC 4013) algorithm for preparing user names and passwords for comparison, hashing, etc.

In general, the mechanism implementations in this library call `saslprep` on their arguments when appropriate.

4 SCRAM Authentication

```
(require sasl/scram)
```
package: sasl-lib

This module implements the SCRAM family of authentication mechanisms, namely SCRAM-SHA-1 and SCRAM-SHA-1-PLUS, SCRAM-SHA-256 and SCRAM-SHA-256-PLUS and SCRAM-SHA-512 and SCRAM-SHA-512-PLUS.

The SCRAM protocol family has the following structure:

1. client → server: initial message with nonce prefix
2. server → client: reply with complete nonce and PBKDF2 salt and iteration count
3. client → server: client signature
4. server → client: authentication result and server signature

In particular: the client sends the first message; authentication success or failure is conveyed at in SASL protocol layer; and the server authenticates itself to the client. Messages are represented as strings.

```
(make-scram-client-ctx digest
                       authentication-id
                       password
                      [#:authorization-id authorization-id
                       #:channel-binding channel-binding])
 → sasl-ctx?
  digest : (or/c 'sha1 'sha256 'sha512)
  authentication-id : string?
  password : string?
  authorization-id : (or/c string? #f) = #f
  channel-binding : (or/c #f #t (list/c symbol? bytes?)) = #f
```

Creates a SCRAM protocol context. The *digest* argument selects between SCRAM-SHA-1, SCRAM-SHA-256 and SCRAM-SHA-512, respectively. The *authentication-id*, *password*, and (if provided) *authorization-id* arguments are automatically processed using saslprep.

The *channel-binding* argument must have the form `(list cb-type cb-data)` if the server offered and the client selected a mechanism with channel binding, indicated with a -PLUS suffix, such as SCRAM-SHA-1-PLUS. The *cb-type* must be a symbol naming a channel binding type, such as 'tls-unique, and *cb-data* must be a byte string containing the corresponding data. The available channel binding types depend on the application and the channel. For example, one common type of channel is TLS; use ssl-channel-binding

to get channel binding data for a TLS connection. The *channel-binding* argument should be #t if the client supports channel binding but the server did not offer a PLUS option. The *channel-binding* argument should be #f if the client does not support channel binding (for example, if the channel is not a TLS connection).

Changed in version 1.1 of package sasl-lib: Added the #:channel-binding argument and support for PLUS mechanism variants.
Changed in version 1.2: Added support for the 'sha512 digest.

5 CRAM-MD5 Authentication

```
(require sasl/cram-md5)
```
package: sasl-lib

This module implements the CRAM-MD5 authentication mechanism.

The CRAM-MD5 protocol has the following structure:

- server → client: challenge
- client → server: response

In particular, the server sends the first message, and the server communicates authentication success or failure at the application protocol layer. Messages are represented as strings.

```
(make-cram-md5-client-ctx authentication-id
                          password)          → sasl-ctx?
  authentication-id : string?
  password : string?
```

Creates a CRAM-MD5 protocol context. The *authentication-id* and *password* arguments are automatically processed using saslprep.

6 PLAIN Authentication

```
(require sasl/plain)
```
package: sasl-lib

This module implements the PLAIN mechanism.

Since the PLAIN mechanism consists of a single message from the client to the server, it is implemented as a simple procedure rather than a SASL protocol context. The authentication outcome is conveyed at the application protocol layer.

```
(plain-client-message  authentication-id
                       password
                      [#:authorization-id authorization-id])
 → string?
  authentication-id : string?
  password : string?
  authorization-id : (or/c string? #f) = #f
```

Constructs a PLAIN client message containig the *authentication-id*, *password*, and (if present) *authorization-id*. The arguments are automatically processed with saslprep.