Network Working Group
Internet Draft: SASL C API
Document: draft-newman-sasl-c-api-03.txt

C. Newman
Sun Microsystems
A. Melnikov
Isode Ltd.

April 2004
Expires in six months

Simple Authentication and Security Layer C API

Status of this memo

0. Meta-information on this draft

This information is intended to facilitate discussion. It will be removed when this document leaves the Internet-Draft stage.

Editorial comments are marked with << and >>.

0.1. Changes since 02

1. Replaced sasl_version() with sasl_version_info().

2. Added missing reference for GSSAPI.

3. Moved some functions between different sections.

4. Added description of sasl_encodev, sasl_utf8verify.

0.1. Changes since 01

1. Clarified the purpose of sasl_errdetail.
2. Clarified what happens when sasl_client_start called second time.
3. Clarified the meaning of sasl_getsecret_t returning SASL_OK with psecret set to NULL.
4. Added more text clarifying callback/interactions relationship.
5. Fixed the description of SASL flags to sasl_client_start/ sasl_server_start.
6. Clarified IP address syntax and added proper references.
7. Replaced references to RFC 1766 with RFC 3066.
8. Clarified how an application can control the order of SASL mechanism selection on the client side.
9. Clarified relationship between serverFQDN and user_realm in sasl_server_new.
10. Split References to Normative and Informative.
11. Added sasl_version function description.
12. Clarified usage of callbacks parameter in sasl_client_init/ sasl_server_init.
13. Clarified that appname parameter to sasl_server_init must be NULL for a library. Added a new property for setting/querying this value.
14. Clarified that sasl_done must be called for each call to sasl_server_init/sasl_client_init and that reference counters should be used internally by SASL C API implementations.
15. Clarified which strings are NUL terminated. This is an ongoing effort.
16. Updated description of different properties, the list should be complete now.
17. Clarified how sasl_server_init should derive user_realm/ serverFQDN if one is not provided.

18. Added a draft version of text that explains how sasl_encode should be called properly.

19. Clarified that sasl_decode must concatenate data if multiple SASL encoded blocks are provided in the same input buffer.

20. Added an additional error code to sasl_getprop/sasl_setprop.

0.2. ToDo

The list of major pending changes/additions is listed below:

1. Add sasl_authorize and sasl_log callbacks.

2. Add a new function for server interactions.

3. Enabling SASL EXTERNAL mechanism on the client side.

4. A lot of cleanup work: for each parameter define what to do if it is NULL, empty string, negative, etc. Clarify thread safety issues. Clarify when different fields allocated by the library are called.

5. Add sasl_user_exists and sasl_setpass?

Abstract

Almost every protocol needs authentication. However, there does not exist an authentication mechanism suitable for all organizations, nor is it likely that a small fixed set of authentication mechanisms will remain suitable. SASL [SASL] provides the on-the-wire framework for authentication (and a security layer) which separates the design of authentication mechanisms from the protocols in which they're used.

The SASL protocol model suggests a software architecture where application protocols call a generic API to authenticate which in turn calls a generic plug-in interface for extensible authentication modules. This memo documents the API used in one implementation of this architecture in the hope that it will be useful to others. An associated memo documenting the plug-in interface is forthcoming.

1. Conventions Used in this Memo

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].

This assumes familiarity the SASL [SASL] specification.

When describing function parameters the following conventions are used: IN before the parameter means that the function expects this paramer to be specified; OUT means that the function sets the parameter upon return; INOUT means that the function requires the parameter to be specified and will update its value upon return.

1.1. Concepts

The following concepts are necessary to understand this specification.

realm
: A realm is a name (usually a domain-style name) associated with a set of users on a server. One realm may span multiple servers. Alternatively, a single server may have multiple realms. Thus there may be multiple users with the username "chris" on the same server, each in a different realm. Some authentication mechanisms have a special field for the realm (e.g., DIGEST-MD5). For other mechanisms, a realm can be specified by the client by using the syntax "username@realm" in the username field.

service
: A service is a basic function provided by one or more protocols. The GSSAPI service name [GSSAPI] registry is available at:

  <http://www.iana.org/numbers.html#G>

  This registry is used by SASL and the SASL API. The service name may be used for service-specific passwords for advanced users, or advanced authentication mechanisms may restrict the services a given server may offer.

virtual domain
: When a single server has multiple realms and there is a DNS server entry for each realm pointing to the same server IP address, then those realms are "virtual domains". Virtual domains are extremely popular with web hosting services and are becoming more popular with POP mail services. The key to providing virtual domain support is that the client informs the server of the domain it believes it is speaking to either through a special protocol element or by using a username of the form "user@realm".

2. Overview of the SASL C API

The SASL API is initialized once at process startup. The sasl_server_init() and sasl_client_init() functions provide basic

initialization.

When a network connection occurs where SASL will be used, a connection-specific context is created for authentication with sasl_client_new() or sasl_server_new(). The API implementation must support multi-threaded servers and clients by creating the connection context in a thread-safe fashion permitting multiple contexts in a given process. At this point, the caller may adjust security policy for the context, and the set of mechanisms which are enabled is determined by requirements from the configuration or by the caller.

The server end of the API may request a list of enabled authentication mechanisms either in general or for a specific user. The client may either select a single mechanism or request a list from the server (if the SASL profile for the protocol in question supports that) and pass the list to the API for automated mechanism selection by configured policy.

The SASL exchange begins with sasl_client_start() which determines if one of the desired mechanisms is available on the client and may generate an initial client response. The client then sends the appropriate protocol message to initiate the SASL exchange that the server passes to sasl_server_start().

The SASL exchange continues with calls to sasl_client_step() and sasl_server_step(), until the server indicates completion or the client cancels the exchange.

The server queries the user name and user realm resulting from the exchange with the sasl_getprop() routine.

A connection context is released with sasl_dispose() and process termination is indicated with sasl_done().

There are a number of utility functions and customization functions available in the API for additional services.

Note, that all functions described in this document can be implemented as macroses, so an application using this API MUST NOT assume that they are functions.

An application or library trying to use the SASL API described in this document must include "sasl.h" include file.

3. Basic SASL API Routines

This section describes the types and functions likely to be used by every caller of the SASL API.

3.1. Basic SASL API Data Structures

The following datastructures are basic to the SASL API.

3.1.1. sasl_callback_t

The sasl_callback_t structure is used for the caller of the SASL API to provide services to both the core SASL API and SASL mechanisms via callbacks. The most important callback is the "getopt" callback (see section 3.2.3) which is used to retrieve security policy option settings from the caller's preferences.

```
typedef struct sasl_callback {
    unsigned int id;
    int (*proc)();
    void *context;
} sasl_callback_t;
```

id is the label for the callback (XXX IANA registry needed), proc is a function pointer whose exact type is determined by the id, and context is a context variable which will be passed to the callback (usually as the first argument). The last callback in the list of callbacks is indicated with an id of SASL_CB_LIST_END.

If proc is NULL, this means that the application doesn't want to specify a corresponding callback, but would provide the necessary data via interaction. See also section 3.1.4.

A SASL mechanism has a list of required callbacks. If all the required callbacks are not provided by the calling application (or not handled as interactions), the SASL mechanism is not going to be selected. Thus it is necessary to list all callbacks that are provided by the application and list all interactions as callbacks with proc as NULL.

3.1.2. sasl_secret_t

The sasl_secret_t structure is used to hold text or binary passwords for the client API.

```
typedef struct sasl_secret {
    unsigned int len;
    unsigned char data[1];
} sasl_secret_t;
```

The len field holds the length of the password, while the data field holds the actual data. The structure is variable sized: enough space must be reserved after the data field to hold the desired password.

An additional '\0' character (not counted in len) is added at the end. Note, however, that binary passwords are permitted to contain '\0' characters.

3.1.3. sasl_conn_t

The sasl_conn_t data type is an opaque data type which reflects the SASL context for a single server connection. Only one SASL API call using a given sasl_conn_t as an argument may be active at a time. However, each sasl_conn_t is independent and thus the SASL API may be used in a true multi-processor multi-threaded environment.

3.1.4. sasl_interact_t

The sasl_interact_t structure is used by sasl_client_start and sasl_client_step to request certain information from the application, when the application did not provide corresponding callbacks. For example, an application may choose to present a single dialog to the user in order to collect all required information interactively.

```
typedef struct sasl_interact {
    unsigned int id;
    const char *challenge;
    const char *prompt;
    const char *defresult;
    const void *result;
    unsigned len;
} sasl_interact_t;
```

The id field holds the value of the callback ID. The prompt field contains a string that should be presented to the user. If non-NULL, challenge is a NUL-terminated string that will allow the user to present a specific credential when prompted. This is different from the prompt in that the prompt is more like a label for a text box (for example "Response:" while challenge is a string that tells the user what specifically is required by the response (for example, an OTP challenge string). The defresult field contains a default value, if any. Upon return from sasl_client_* the "result" field points to the defresult. The client must present the information in the challenge and the prompt to the user and store the result and its length in the result and the len fields respectively.

For example, SASL_CB_PASS interaction may contain the following information:

- id - SASL_CB_PASS
- challenge - NULL
- prompt - "Password:"
- defresult - NULL (no default).

3.2. Basic SASL API Callback Routines

This section describes the basic callback functions needed for a simple client implementation. See the definition of sasl_callback_t in section 3.1.1 for a description of the basic callback structure.

3.2.1. sasl_getsimple_t

```
Arguments:
            void *context,
            int id,
            const char **result,
            unsigned *len

Results:
            SASL_OK          -- success
            SASL_FAIL        -- error
```

This callback is used by the SASL API to request a simple constant string from the application. This is used with id SASL_CB_USER for the username, SASL_CB_AUTHNAME for the authentication name (if different), and SASL_CB_LANGUAGE for a comma separated list of [LANGTAGS] language tags.

The context is the context variable from the sasl_callback_t structure, the id is the id from the sasl_callback_t structure, and the callback is expected to set the result to a constant string and the len to the length of that string. The result and len parameters are never NULL.

3.2.2. sasl_getsecret_t

```
Arguments:
            IN sasl_conn_t *conn,
            IN void *context,
            IN int id,
            OUT sasl_secret_t **psecret

Results:
            SASL_OK          -- success
            SASL_FAIL        -- error
```

This callback is expected to create, prompt or locate a secret and return the pointer to it in the psecret parameter. The secret MUST persist till next call to this callback/sasl_dispose for the same SASL connection. The conn argument is the connection context, the context and id parameters are from the sasl_callback_t structure. The id SASL_CB_PASS is used to request a clear text password. The

conn and the psecret parameters MUST NOT be NULL.

Returning SASL_OK with psecret set to NULL should indicate a user desire to cancel the authentication sequence (e.g., user pressed Cancel in a password dialog box).

3.2.3. sasl_getopt_t

Arguments:

```
            void *context,
            const char *plugin_name,
            const char *option,
            const char **result,
            unsigned int *len
```

Results:

```
            SASL_OK          -- success
            SASL_FAIL        -- error
```

This callback is used by the SASL API to read options from the application. This allows a SASL configuration to be encapsulated in the caller's configuration system. Configuration items may be mechanism-specific and are arbitrary strings. If the application does not provide a sasl_getopt_t callback, then the API MAY obtain configuration information from other sources, for example from a default config file.

The context is the context variable from the sasl_callback_t structure, the plugin_name is the name of plugin (or NULL), the option is the option name, and the callback is expected to set the result to a string valid till next call to sasl_getopt_t in the same thread and the len to the length of that string. The result and len parameters are never NULL. If the name of plugin is NULL, a general SASL option is requested, otherwise a plugin specific version.

3.3. Basic SASL API Client Routines

This section discusses the functions likely to be used by every client caller of the SASL API.

3.3.1. sasl_client_init function

Arguments:

```
            sasl_callback_t *callbacks
```

Results:

```
            SASL_OK        -- Success
            SASL_NOMEM     -- Not enough memory
            SASL_BADVERS   -- Mechanism version mismatch
            SASL_BADPARAM  -- Error in config file
```

This function initializes the client routines for the SASL API.

The callbacks argument is the default list of callbacks (see section 3.1.1 for definition of sasl_callback_t structure). Libraries that are clients of the API MUST supply NULL For this parameter. Applications that are clients of the API MAY supply a list that includes the sasl_getopt_t callback (see section 3.2.3), but MUST NOT supply any other callbacks through this interface. An application or library that needs to specify other callbacks shall use prompt_supp parameter of the sasl_client_new (see section 3.3.2).

On success, SASL_OK is returned, and on failure a SASL C API error code such as the ones listed above is returned. This function may be called a second time to change the global sasl_getopt_t callback used for new connections, but the first call must be made in a single-threaded environment.

There must be a call to sasl_done for every successful call to sasl_server_init or sasl_client_init made. Each call to sasl_server_init or sasl_client_init increment the internal reference counter. All but the last call to sasl_done descrement the counter, the final sasl_done does the actual cleanup. The data referenced by the sasl_callback_t structure must persist until the very last call to sasl_done().

3.3.2. sasl_client_new function

Arguments:

```
            const char *service,
            const char *server_name,
            const char *iplocalport,
            const char *ipremoteport,
            const sasl_callback_t *prompt_supp,
            unsigned int flags,
            sasl_conn_t **pconn
```

Results:

```
            SASL_OK        -- Success
            SASL_NOTINIT   -- SASL API not initialized
            SASL_NOMECH    -- No mechanisms available
            SASL_NOMEM     -- Not enough memory
```

This function creates a client connection context variable. As long as each thread uses its own connection context, the SASL C API is thread-safe.

The service argument is an IANA registered GSSAPI service element as defined in section 1.1. It MUST NOT be NULL.

The server_name is the host name or IP address of the server to which the client is connecting. NULL may be used for server_name, but may result in advanced mechanisms such as Kerberos being unavailable.

The iplocalport is the string with the client IPv4/IPv6 address, followed by ":" and than by port number. The syntax of IPv4/IPv6 addresses is defined by IPv4address/IPv6address ABNF elements from [RFC 2373]. NULL may be used for iplocalport, but may result in mechanisms requiring IP address being unavailable.

The ipremoteport is the string with the server IPv4/IPv6 address, followed by ":" and than by port number. The syntax of IPv4/IPv6 addresses is defined by IPv4address/IPv6address ABNF elements from [RFC 2373]. NULL may be used for ipremoteport, but may result in mechanisms requiring IP address being unavailable.

User input to the SASL C API may be provided in two ways: either by supplying callbacks (prompt_supp) to this function, or by using an interaction model with the sasl_client_start/sasl_client_step functions. Callbacks are more convenient to obtain information programmatically, such as pulling authentication information directly from a configuration file. Interactions are more convenient if one wants to get all the data in parallel, for example by

displaying a single dialog box instead of a separate popup for authentication name, authorization, password, etc.

The prompt_supp is a list of supported user prompting callbacks discussed in the section 3.1.1. The prompt_supp argument MAY be NULL, which means that interactions (i.e. prompt_need parameter to sasl_client_start (see 3.3.3) and sasl_client_step (see 3.3.4)) are used instead of callbacks. If prompt_supp is NULL, the prompt_need argument to sasl_client_start (see 3.3.3) and sasl_client_step (see 3.3.4) MUST NOT be NULL.

The prompt_supp argument may include a connection-specific sasl_getopt_t callback. If a connection-specific sasl_getopt_t callback is specified in the prompt_supp list, it will take precedence over the global sasl_getopt_t callback specified in the last sasl_client_init() call.

A SASL mechanism has a list of required callbacks. If all the required callbacks are not provided by the calling application (or not handled as interactions), the SASL mechanism is not going to be selected. Thus it is necessary to list all callbacks that are provided by the application and list all interactions as callbacks with proc as NULL.

The flags argument represents client-supported security flags. The value is a bitmask. Currently, only two values are defined: SASL_SUCCESS_DATA and SASL_NEED_PROXY.

The SASL_SUCCESS_DATA flag specifies that the server (and the protocol) supports sending "additional data on success". This flag is used by the SASL library to decide whether it can pass the success data back along with SASL_OK, or if it must return SASL_CONTINUE and then wait for an empty client response before returning SASL_OK.

The SASL_NEED_PROXY flag tells the SASL library that it must only select mechanisms that support proxy authorization.

The pconn argument is set to point to the newly created connection context. The sasl_conn_t type is opaque to the calling application.

3.3.3. sasl_client_start function

Arguments:

```
            sasl_conn_t *conn,
            const char *mechlist,
            sasl_interact_t **prompt_need,
            const char **clientout,
            unsigned int *clientoutlen,
            const char **mech
```

Results:

```
            SASL_NOTINIT   -- SASL API not initialized
            SASL_BADPARAM  -- conn or mechlist is NULL
            SASL_NOMECH    -- No matching mechanisms available
            SASL_NOMEM     -- Not enough memory
            SASL_INTERACT  -- User interaction needed to continue
                              (see prompt_need description below)
            SASL_OK        -- Success
```

This selects an authentication mechanism to use and optionally generates an initial client response.

The conn argument is the connection context from sasl_client_new.

The mechlist argument is a '\0' terminated string containing one or more SASL mechanism names. All characters in the string that are not permitted in a SASL mechanism name [SASL] are ignored except for the purposes of delimiting mechanism names (this permits passing direct results from many protocol capability lists unparsed). Unknown mechanism names are ignored (although SASL_NOMECH is returned if no known mechanisms are found). Mechanisms are tried in an implementation-dependent order. Implementations SHOULD try to use the most secure mechanism possible, within the constraints specified by the application (e.g. SSF value). If the application wants to control which mechanism from the list would be selected, it has to break the list apart and call the function by passing a single SASL mechanism name at a time in the mech_list parameter.

For applications which support interactions, the prompt_need argument should initially point to a NULL pointer. If the selected mechanism needs information from the user (for example, username or password), then prompt_need will be set to point to an array of sasl_interact_t structures (terminated by an entry with id equal to SASL_CB_LIST_END), and sasl_client_start will return SASL_INTERACT. After that the client must fill in the requested information and call this function again with the same parameters.

Applications that do not support interactions MUST pass NULL for prompt_need.

The clientout and clientoutlen parameters are set to the initial client response, if any. If a protocol's SASL profile uses base64 encoding, this represents the data prior to the encoding (see sasl_encode64). If a protocol's SASL profile doesn't include an optional initial client response, then these may be NULL and 0 respectively. The memory used by clientout is interally managed by the SASL API and may be overwritten on the next call to sasl_client_step or a call to sasl_dispose.

The mech argument is set to point to a '\0' terminated string specifying the mechanism actually selected using all uppercase letters. It may be NULL if the client does not care which mechanism was selected from mechlist.

If sasl_client_start is called a second time using the same connection context, it will discard any cached information (e.g., the username and password) and restart the exchange as if this were the first call to the sasl_client_start.

3.3.4. sasl_client_step function

```
Arguments:
            sasl_conn_t *conn,
            const char *serverin,
            unsigned int serverinlen,
            sasl_interact_t **prompt_need,
            const char **clientout,
            unsigned int *clientoutlen

Results:
            SASL_NOTINIT     -- SASL API not initialized
            SASL_NOMECH      -- sasl_client_start not called
            SASL_BADPROT     -- server protocol incorrect/cancelled
            SASL_BADSERV     -- server failed mutual auth
            SASL_INTERACT    -- user interaction needed
            SASL_OK          -- success
```

This routine performs one step in an authentication sequence.

The conn argument must be a connection context created by sasl_client_new and used in a previous call to sasl_client_start.

The serverin and serverinlen parameters hold the SASL octet string received from the server. Note that for those SASL profiles which base64 encode the exchange, this is the result after the removal of the base64 encoding (see the sasl_decode64 routine below). The serverin may contain arbitrary binary data, in particular it MAY contain one or more NUL characters. The serverin MUST have a

terminating NUL character not counted by serverinlen.

The prompt_need argument is the same as for sasl_client_start.

The clientout and clientoutlen parameters hold the SASL octet string to encode (if necessary) and send to the server.

3.4. Basic SASL C API Server Routines

This section describes the basic routines for a server implementation of a SASL profile.

3.4.1. sasl_server_init function

```
Arguments:
            const sasl_callback_t *callbacks,
            const char *appname

Results:
            SASL_BADPARAM     -- error in config file
            SASL_NOMEM        -- out of memory
            SASL_BADVERS      -- Plug-in version mismatch
            SASL_OK           -- success
```

This function initializes the server routines for the SASL C API.

The callbacks argument is the default list of callbacks (see section 3.1.1 for definition of sasl_callback_t structure). Libraries that are clients of the API MUST supply NULL for this parameter. Applications that are clients of the API MAY supply a list that includes the sasl_getopt_t callback (see section 3.2.3), but MUST NOT supply any other callbacks through this interface. An application or library that needs to specify other callbacks shall use callbacks parameter of the sasl_server_new (see section 3.4.2).

The appname argument is the name of the calling application. SASL API may use it, for example, for logging or to read an application specific configuration. A library must pass NULL as appname. appname can be also be set with sasl_setprop function, and can be queried with sasl_getprop. The corresponding constant for the appname option is SASL_APPNAME.

On success, SASL_OK is returned, and on failure a SASL C API error code is returned. This function may be called a second time to change the global sasl_getopt_t callback used for new connections, but the first call must be made in a single-threaded environment.

There must be a call to sasl_done for every successful call to sasl_server_init or sasl_client_init made. Each call to sasl_server_init or sasl_client_init increment the internal reference counter. All but the last call to sasl_done descrement the counter, the final sasl_done does the actual cleanup. The data referenced by the sasl_callback_t structure must persist until the very last call to sasl_done().

3.4.2. sasl_server_new function

```
Arguments:
            IN const char *service,
            IN const char *serverFQDN,
            IN const char *user_realm,
            IN const char *iplocalport,
            IN const char *ipremoteport,
            IN const sasl_callback_t *callbacks,
            In unsigned int flags,
            OUT sasl_conn_t **pconn

Results:
            SASL_OK        -- success
            SASL_NOTINIT   -- SASL API not initialized
            SASL_BADPARAM  -- Invalid parameter supplied
            SASL_NOMECH    -- No mechanisms available
            SASL_NOMEM     -- Not enough memory
```

This function creates a server connection context variable. As long as each thread uses its own connection context, the SASL C API is thread-safe.

The service argument is an IANA registered GSSAPI service element as defined in section 1.1. It MUST NOT be NULL.

The serverFQDN is the fully qualified name of the server. It SHOULD NOT be NULL. If it is NULL, the SASL library can default the value in the implementation defined manner, e.g. the library MAY use a value stored in a configuration file or the result of gethostname() or a similar call.

The user_realm specifies the default realm. A realm defines a set of users on the system for systems which support multiple user communities ("realms"). If user_realm is NULL, the value of serverFQDN is used as the default realm. (If serverFQDN is also NULL, it is assumed that it has the value defaulted as described above.)

The iplocalport is the string with the server IPv4/IPv6 address,

followed by ":" and than by the port number. The syntax of IPv4/IPv6 addresses is defined by IPv4address/IPv6address ABNF elements from [RFC 2373]. NULL may be used for iplocalport, but may result in mechanisms requiring IP address being unavailable.

The ipremoteport is the string with the client IPv4/IPv6 address, followed by ":" and than by the port number. The syntax of IPv4/IPv6 addresses is defined by IPv4address/IPv6address ABNF elements from [RFC 2373]. NULL may be used for ipremoteport, but may result in mechanisms requiring IP address being unavailable.

The callbacks argument is a set of server callbacks which may include a connection-specific sasl_getopt_t. If a connection-specific sasl_getopt_t callback is specified in callbacks list, it will take precedence over the global sasl_getopt_t callback specified in the last sasl_server_init() call.

The flags argument represents server-supported security flags. The value is a bitmask. Currently, only two values are defined: SASL_SUCCESS_DATA and SASL_NEED_PROXY.

The SASL_SUCCESS_DATA flag specifies that the server (and the protocol) supports sending "additional data on success". This flag is used by the SASL library to decide whether it can pass the success data back along with SASL_OK, or if it must return SASL_CONTINUE and then wait for an empty client response before returning SASL_OK.

Even if SASL_SUCCESS_DATA is specified, the server MUST check the serverout pointer it receives from sasl_server_step() to see if it non-null, because some mechs only have success data in certain cases (ie, SRP with SSF==0).

The SASL_NEED_PROXY flag tells the SASL library that it must only select mechanisms that support proxy authorization.

The pconn argument is set to point to the newly created connection context.

3.4.3. sasl_server_start function

Arguments:

```
            sasl_conn_t *conn,
            const char *mech,
            const char *clientin,
            insigned int clientinlen,
            const char **serverout,
            unsigned int *serveroutlen
```

Results:

```
            SASL_CONTINUE  -- Another authentication step required
            SASL_OK        -- Authentication Complete
            SASL_NOTINIT   -- SASL API not initialized
            SASL_BADPARAM  -- Invalid parameter supplied
            SASL_BADPROT   -- Client protocol error
            SASL_NOMECH    -- Mechanism not supported
            SASL_NOVERIFY  -- User exists, but no verifier exists for
                              the mechanism
            SASL_TRANS     -- A password transition is needed to use mechanism
```

This begins an authentication exchange and is called after the client sends the initial authentication command. The mech argument is the mechanism name the client is requesting. If the client includes an optional initial-response, it is passed in the clientin and clientinlen fields. Otherwise NULL and 0 are passed for those arguments. The serverout and serveroutlen are filled in with the server response, if any. If SASL_CONTINUE is returned, the server will need to wait for another client message and call sasl_server_step. If SASL_OK is returned, the authentication is completed successfully, although serverout may be supplied.

<<...Does this work for the server side?...>> If sasl_server_start is called a second time using the same connection context, it will discard any cached information (e.g., the username and password) and restart the exchange as if this were the first call to the sasl_server_start.

3.4.4. sasl_server_step function

Arguments:

```
            sasl_conn_t *conn,
            const char *clientin,
            insigned int clientinlen,
            const char **serverout,
            unsigned int *serveroutlen
```

Results:

```
            SASL_CONTINUE  -- Another authentication step required
            SASL_OK        -- Authentication Complete
            SASL_NOTINIT   -- SASL API not initialized
            SASL_NOMECH    -- sasl_server_start not called
            SASL_BADPARAM  -- Invalid parameter supplied
            SASL_BADPROT   -- Client protocol error
            SASL_NOVERIFY  -- User exists, but no verifier exists for
                              the mechanism
            SASL_TRANS     -- A password transition is needed to use mechanism
```

This routine performs one step in an authentication sequence.

The conn argument must be a connection context created by sasl_server_new and used in a previous call to sasl_server_start.

The clientin and clientinlen parameters hold the SASL octet string received from the client. Note that for those SASL profiles which base64 encode the exchange, this is the result after the removal of the base64 encoding (see the sasl_decode64 routine). The clientin may contain arbitrary binary data, in particular it MAY contain one or more NUL characters. The clientin MUST have a terminating NUL character not counted by serverinlen.

The serverout and serveroutlen parameters hold the SASL octet string to encode (if necessary) and send to the client. If SASL_CONTINUE is returned, the server will need to wait for another client message and call sasl_server_step. If SASL_OK is returned, the authentication is completed successfully, although server out data may be supplied.

3.5. Common SASL API Routines

This section describes the routines that are common to both clients and servers.

3.5.1. sasl_listmech function

Arguments:

```
            sasl_conn_t *conn,
            const char *user,
            const char *prefix,
            const char *sep,
            const char *suffix,
            char **result,
            unsigned int *plen,
            unsigned *pcount
```

Results:

```
            SASL_OK        -- Success
            SASL_NOMEM     -- Not enough memory
            SASL_NOMECH    -- No enabled mechanisms
```

This returns a list of enabled SASL mechanisms in a NUL-terminated string. The list is constructed by placing the prefix string at the beginning, placing the sep string between any pair of mechanisms and placing the suffix string at the end.

When calling this function plen and pcount MAY be NULL.

This function returns the list of the client side SASL mechanisms, if the conn was created by sasl_client_new and the list of the server side mechanisms, if the conn was created by sasl_server_new. The list returned by this function must persist till a next call to sasl_free_listmech or sasl_listmech.

3.5.2. sasl_free_listmech function

Arguments:

```
            sasl_conn_t *conn,
            char **result
```

Results:

```
            none
```

This disposes of the result string returned by sasl_listmech.

3.5.3. sasl_setprop function

```
Arguments:
            sasl_conn_t *conn,
            int propnum,
            const void *value

Results:
            SASL_OK          -- property set
            SASL_BADPARAM    -- invalid propnum or value or connection is NULL
            SASL_BADPROT     -- the property is not settable for this type of
connection
            SASL_NOMEM       -- not enough memory to perform operation
```

This sets a property in a connection context. Commonly used properties with their descriptions are listed below:

SASL_SSF_EXTERNAL

Security layer strength factor (SSF) -- an unsigned integer usable by the caller to specify approximate security layer strength desired. It roughly corresponds to the effective key length for encryption, e.g.
0 = no protection
1 = integrity protection only >1 = key lenght of the cipher

SASL_SSF_EXTERNAL property denotes SSF of the external security layer (e.g. provided by TLS). The value parameter points to sasl_ssf_t, that is described as follows:

```
typedef unsigned sasl_ssf_t;
```

SASL_SEC_PROPS

The value parameter for SASL_SEC_PROPS points to sasl_security_properties_t structure defined below. A particular implementation may extend it with additional fields.

```
typedef struct sasl_security_properties
{
    sasl_ssf_t min_ssf;
    sasl_ssf_t max_ssf;

    unsigned maxbufsize;

    /* bitfield for attacks to protect against */
    unsigned security_flags;
} sasl_security_properties_t;
```

The min_ssf and the max_ssf define the minimal and the maximal acceptable SSF.

The maxbufsize specifies the biggest buffer size that the client/server is able to decode. 0 means that security layer is not supported.

The security_flags is a bitmask of the various security flags described below:

```
SASL_SEC_NOPLAINTEXT          -- don't permit mechanisms susceptible to
simple
                                 passive attack (e.g., PLAIN, LOGIN)
SASL_SEC_NOACTIVE             -- protection from active (non-dictionary)
attacks
                                 during authentication exchange.
                                 Authenticates server.
SASL_SEC_NODICTIONARY         -- don't permit mechanisms susceptible to
passive
                                 dictionary attack
SASL_SEC_FORWARD_SECRECY      -- require forward secrecy between sessions
                                 (breaking one won't help break next)
SASL_SEC_NOANONYMOUS          -- don't permit mechanisms that allow
anonymous login
SASL_SEC_PASS_CREDENTIALS     -- require mechanisms which pass client
                                 credentials, and allow mechanisms which
can pass
                                 credentials to do so
SASL_SEC_MUTUAL_AUTH          -- require mechanisms which provide mutual
                                 authentication
```

SASL_AUTH_EXTERNAL

The value parameter for SASL_AUTH_EXTERNAL property points to the external authentication ID as provided by external authentication method, e.g. TLS, PPP or IPSec.

Full list of properties is provided in section 6.

3.5.4. sasl_getprop function

```
Arguments:
            sasl_conn_t *conn,
            int propnum,
            const void **pvalue

Results:
            SASL_OK        -- Success
            SASL_NOTDONE   -- Authentication exchange must complete prior to
                              retrieving this attribute
            SASL_BADPARAM  -- bad property number or invalid connection type
            SASL_BADPROT   -- the property is not readable for this type of
connection
```

This requests a pointer to a constant property available through the SASL API. The most common use by servers is to get the SASL_USERNAME property which returns the authorization identity (user to login as) from the SASL mechanism as a UTF-8 string in the pvalue parameter. Full list of properties is provided in section 6.

3.5.5. sasl_dispose function

```
Arguments:
            sasl_conn_t **pconn

Results:
            none
```

This function disposes of the connection state created with sasl_client_new or sasl_server_new, and sets the pointer to NULL. If the pconn is already NULL the function does nothing.

3.5.6. sasl_done function

```
Arguments:
            none

Results:
            <<SASL_OK        -- Success
            SASL_CONTINUE  -- Internal reference counter is not 0, need to call
sasl_done until it returns SASL_OK
            SASL_FAIL      -- Reference counter is already 0>>
```

A SASL application that is finished with the SASL API must call this function. This function frees any memory allocated by the SASL library or any other library state. After this call most of the SASL API function will again return the SASL_NOTINIT error code.

There must be a call to sasl_done for every successful call to

sasl_server_init or sasl_client_init made. Only the final sasl_done does the actual cleanup; the preceding calls simply decrement an internal reference count.

Connection states MUST be disposed of with sasl_dispose before calling this function.

3.5.7. sasl_version_info function

Arguments:

```
const char **implementation,
const char **version_string,
int *version_major,
int *version_minor,
int *version_step,
int *version_patch
```

Results:

none

This function returns a string identifying a particular implementation of the SASL C API (implementation parameter) and a vendor specific version number. This function can be used for library version display and logging. It may be also used for bug workarounds in old library versions. This funtion should not to be used for API feature detection.

The vendor specific version number is returned in a string form in the version_string parameter. Specific components of the version number are returned in version_major, version_minor, version_step and version_patch. Note, that a particular SASL implementation may choose not to use a version component, for example the version_patch. In this case the function should always return 0 in such parameter.

All parameters of this function are optional. If NULL is specified, the value is not returned.

The implementation and the version_string strings are not allocated and must not be freed.

3.5.8. sasl_errstring function

```
Arguments:
            int saslerr,
            const char *langlist,
            const char **outlang

Results:
            const char *
```

This converts a SASL error number into a constant string. The second argument MAY be NULL for the default language, or a comma-separated list of [LANGTAGS] language tags. The final parameter is set (if not NULL) to the [LANGTAGS] language tag of the string returned which will be "i-default" if no matching language is found. The strings are UTF-8. This requires no context so it may be used for the result of an sasl_*_init or sasl_*_new result code.

3.5.9. sasl_errdetail function

```
Arguments:
            sasl_conn_t *conn

Results:
            const char *
```

This function returns a human readable string that corresponds to the last SASL error that occured on the connection. The string MUST be in UTF8. This string is suitable to be passed back over the protocol and presented to an end-user. Thus, it must not leak a security sensitive information, e.g. it must not show the distinction between "user not found" and "bad password".

The function must use the SASL_CB_LANGUAGE callback (see section 3.2.1) to determine the language to use. It may return more detailed information than sasl_errstring does.

3.5.10. sasl_seterror function

```
Arguments:
            sasl_conn_t *conn
            unsigned flags,
            const char *fmt,
             ...

Results:
            none
```

This function sets the error string which will be returned by

sasl_errdetail. It uses syslog()-style formatting (i.e. printf-style with %m as the string form of the errno error).

Messages should be sensitive to the current language setting. If there is no SASL_CB_LANGUAGE callback for the connection, text MUST be in US-ASCII. Otherwise UTF-8 is used and use of RFC 2482 for mixed-language text is encouraged.

The resulting formatted string should be stored in connection context until connection context is destroyed or a next call to sasl_seterror() <<and can be retrieved by calling sasl_getprop with SASL_PLUGERR property>>.

The flags parameter can be either 0 or SASL_NOLOG. If the flags parameter is 0, upon formatting the error message the sasl_seterror will call the SASL logging callback (if any) with a level of SASL_LOG_FAIL. <<Is this callback defined anywhere?>>

This function may be used by server callbacks.

If conn is NULL, the function does nothing.

3.6. Basic SASL C API Utility Routines

This section describes utility functions provided as part of the SASL API which may be used both by clients and servers.

3.6.1. sasl_decode64 function

```
Arguments:
            const char *in,
            unsigned int inlen,
            char *out,
            unsigned int outmax,
            unsigned int *outlen

Results:
            SASL_BUFOVER    -- output buffer too small
            SASL_BADPROT    -- invalid base64 string
            SASL_OK         -- successful decode
```

This utility routine converts a base64 string of length inlen pointed by in into an octet string. It is useful for SASL profiles which use base64 such as the IMAP [IMAP4] and POP [POP-AUTH] profiles. The output is copied to the buffer specified by the out parameter. It is NUL terminated and the length of the output is placed in the outlen parameter if outlen is non-NULL. The lenght doesn't include the terminating NUL character.

When the size of the output buffer, as specified by outmax, is too small, the function returns SASL_BUFOVER error code and the required length is stored in the outlen parameter if it is not NULL.

The function may also return SASL_BADPROT error code when it encounters an invalid base64 character.

3.6.2. sasl_encode64 function

Arguments:

```
            const char *in,
            unsigned int inlen,
            char *out,
            unsigned int outmax,
            unsigned int *outlen
```

Results:

```
            SASL_BUFOVER    -- output buffer too small
            SASL_OK         -- successful decode
```

This utility routine converts an octet string of length inlen pointed by in into a base64 string. It is useful for SASL profiles which use base64 such as the IMAP [IMAP4] and POP [POP-AUTH] profiles.

The output is copied to the buffer specified by the out parameter. It is NUL terminated and the length of the output is placed in the outlen parameter if outlen is non-NULL. The lenght doesn't include the terminating NUL character.

When the size of the output buffer, as specified by outmax, is too small, the function returns SASL_BUFOVER error code and the required length is stored in the outlen parameter if it is not NULL.

3.6.3. sasl_erasebuffer function

Arguments:

```
            char *buf,
            unsigned len
```

Results:

```
            none
```

This function fills the buffer buf of the length len with '\0' characters. The function may be used to clear from memory sensitive informations, like passwords.

3.6.4. sasl_utf8verify function

```
Arguments:
            const char *str,
            unsigned len

Results:
            SASL_BADPROT    -- the input string is not a valid UTF-8 string
            SASL_OK         -- the input string is a valid UTF-8 string
```

This function verifies if the str is a valid UTF-8 string. If the len is 0, the length of the string is determined by calling strlen(str).

4. SASL Security Layer Routines

This section describes the routines need to support a security layer.

4.1. sasl_encode function

```
Arguments:
            sasl_conn_t *conn,
            const char *input,
            unsigned int inputlen,
            const char **output,
            unsigned int *outputlen

Results:
            SASL_OK        -- Success (returns input if no layer was negotiated)
            SASL_NOTDONE   -- Security layer negotiation is not yet finished
            SASL_BADPARAM  -- inputlen is greater than the SASL_MAXOUTBUF
property,
                              or a parameter is invalid (i.e. conn is NULL,
input
                              or output is NULL).
            <<SASL_TOOWEAK  -- security layer is not supported (maxbuf == 0)>>
```

This function encodes a block of data for transmission using security layer (if any). The output and outputlen are filled in with the encoded data and its length respectively. If there is no security layer the input buffer is returned in the output. Otherwise, the output is only valid until a next call to sasl_encode, sasl_encodev or sasl_dispose.

Both sides of the connection should follow the following guidelines

* Before starting authentication, the application should set

the maxbufsize field in the sasl_security_properties_t

to be the buffer size that the application passes to
the read() system call, i.e. the amount of data that the
application is prepared to read at any one time.
After that the application should use sasl_setprop() with
the property index of SASL_SEC_PROPS to notify SASL API
about the choice.

* After authentication finishes, the application should use
sasl_getprop() to retrieve the SASL_MAXOUTBUF value, and call
sasl_encode() with chunks of data of that size or less.
sasl_encode() will return SASL_BADPARAM if it is called
with a larger chunk of data.

4.2. sasl_encodev function

```
Arguments:
            sasl_conn_t *conn,
            const struct iovec *invec,
            unsigned int numiov,
            const char **output,
            unsigned int *outputlen

Results:
            SASL_OK        -- Success (returns input if no layer was negotiated)
            SASL_NOTDONE   -- Security layer negotiation is not yet finished
            SASL_BADPARAM  -- inputlen is greater than the SASL_MAXOUTBUF
property,
                              or a parameter is invalid (i.e. conn is NULL,
input
                              or output is NULL).
            <<SASL_TOOWEAK  -- security layer is not supported (maxbuf == 0)>>
```

This function encodes one or more block of data for transmission using security layer (if any). The output and outputlen are filled in with the encoded data and its length respectively. <<If there is no security layer the input buffer is returned in the output.>> Otherwise, the output is only valid until a next call to sasl_encode, sasl_encodev or sasl_dispose.

The invec parameter is an array of iovec structures defined as follows:

```
struct iovec {
    long iov_len;
    char *iov_base;
};
```

where iov_base is a pointer to a block of data to be encoded and iov_len is the length of the block. This is the same structure

that is used by writev() function on many Unix systems.

The number of data blocks in invec is specified by the numiov parameter.

See also note in section 4.1 about use of this function.

4.3. sasl_decode function

```
Arguments:
            sasl_conn_t *conn,
            const char *input,
            unsigned int inputlen,
            const char **output,
            unsigned int *outputlen

Results:
            SASL_OK        -- Success (returns input if no layer was negotiated)
            SASL_NOTDONE   -- Security layer negotiation is not yet finished
            SASL_BADMAC    -- Bad message integrity check
            SASL_BADPARAM  -- A parameter is invalid (i.e. conn is NULL, input
                              or output is NULL).
```

This function decodes a block of data received using security layer (if any). The output and outputlen are filled in with the decoded data and its length respectively. If there is no security layer the input buffer is returned in the output. Otherwise, the output is only valid until a next call to sasl_decode or sasl_dispose.

If the input contains more than one SASL token, sasl_decode MUST concatenate the decoded data together. If the input contains one or more SASL tokens and a few bytes of a subsequent SASL token, sasl_decode MUST save the beginning of the uncomplete token in the conn and use the saved data when the sasl_decode is called the next time. Because all SASL mechanisms MUST negotiate or declare the maximal token size, preallocated buffer inside sasl_conn_t can be used for this.

5. Advanced SASL API Routines

This section describes the less frequently used functions available in the SASL API.

5.1. Additional Initialization Routines

5.1.1. sasl_set_mutex function

Arguments:

```
            sasl_mutex_alloc_t  *mutex_alloc,
            sasl_mutex_lock_t   *mutex_lock,
            sasl_mutex_unlock_t *mutex_unlock,
            sasl_mutex_free_t   *mutex_free
```

Results:

```
            None
```

The sasl_set_mutex call sets the callbacks which the SASL API and plug-ins will use whenever exclusive access to a process shared resource is needed. A single-threaded client or server need not call this. The types are designed to be compatible with the LDAP API [LDAP-API]:

```
typedef void *sasl_mutex_alloc_t(void);
```

On success, this returns a pointer to an allocated and initialized mutex structure. On failure, it returns NULL.

```
typedef int sasl_mutex_lock_t(void *mutex);
```

This will block the current thread until it is possible to get an exclusive lock on a mutex allocated by the mutex_alloc callback. On success it returns 0, on failure due to deadlock or bad parameter, it returns -1.

```
typedef int sasl_mutex_unlock_t(void *mutex);
```

This releases a lock on a mutex allocated by the mutex_alloc callback. On success it returns 0, on failure due to an already unlocked mutex, or bad parameter, it returns -1.

```
typedef void sasl_mutex_free_t(void *mutex);
```

This disposes of a mutex allocated by mutex_alloc.

5.1.2. sasl_set_alloc function

Arguments:

```
            sasl_malloc_t  *malloc,
            sasl_calloc_t  *calloc,
            sasl_realloc_t *realloc,
            sasl_free_t    *free
```

Results:

```
            None
```

This sets the memory allocation functions which the SASL API will use. The SASL API will use its own routines (usually the standard C library) if these are not set.

```
typedef void *sasl_malloc_t(unsigned long mem_size);
```

This allocates memory mem_size bytes of memory. The memory is not initialized to any particular value. It returns NULL on a failure, or when mem_size is 0.

```
typedef void *sasl_calloc_t(unsigned long elem_size,
                unsigned long num_elem);
```

This allocates elem_size * num_elem bytes of memory. The memory is initialized to 0. It returns NULL on a failure or when either elem_size and/or num_elem is 0.

```
typedef void *sasl_realloc_t(void *mem_ptr, unsigned long
new_size);
```

This changes the size of a memory block previously allocated by malloc or calloc, and returns a pointer to the new location (which may be different from mem_ptr). If mem_ptr is NULL, it is identical to the malloc function.

It returns NULL on a failure or when new_size is 0. On failure the original block is unchanged. When new_size is 0 the function works as the free function.

```
typedef void sasl_free_t(void *mem_ptr);
```

This releases the memory in mem_ptr that was allocated by the malloc or the calloc or resized by the realloc. If mem_ptr is NULL, the function does nothing and returns immediately. The contents of the memory may be altered by this call.

6. Standard Properties

r/o - read only

```
      r/w - read-write
      server - server side only

      SASL_USERNAME     -- (r/o) pointer to NUL terminated user name
(authorization id)
      SASL_SSF          -- (r/o) security layer security strength factor,
                           if 0, call to sasl_encode, sasl_decode unnecessary
      SASL_MAXOUTBUF    -- (r/o) security layer maximal output buffer size,
unsigned.
                           Use maxbufsize field in sasl_security_properties_t
structure
                           (settable as SASL_SEC_PROPS property) to set
required maximal
                           output buffer size before starting a SASL
authentication.
                           The value returned by SASL_MAXOUTBUF may differ (be
less) from
                           the value specified in maxbufsize field.
      SASL_DEFUSERREALM -- (r/w, server) default realm passed to
sasl_server_new or set with
                           sasl_setprop
      SASL_GETOPTCTX    -- (r/o) context for getopt callback
      SASL_CALLBACK     -- (r/o) current callback function list
      SASL_IPLOCALPORT  -- (r/w) iplocalport string passed to sasl_server_new/
                           sasl_client_new or set with sasl_setprop.
      SASL_IPREMOTEPORT -- (r/w) ipremoteport string passed to sasl_server_new/
                           sasl_client_new or set with sasl_setprop.
      SASL_PLUGERR      -- (r/o) a string which is either empty or has an error
message
                           from the sasl_seterror (e.g., from a plug-in or
callback).
                           It differs from the result of sasl_errdetail() which
also takes
                           into account the last return status code.
      SASL_SERVICE      -- (r/o) service passed to sasl_*_new
      SASL_SERVERFQDN   -- (r/o) serverFQDN passed to sasl_*_new
      SASL_AUTHSOURCE   -- (r/o) name of the active plugin, if any.
                           If the implementation of SASL API doesn't support
plugins,
                           it SHOULD return the name of the active SASL
mechanism (if any).
                           If no mechanism name is available, sasl_getprop
should fail
                           with SASL_NOTDONE error code.
      SASL_MECHNAME     -- (r/o) active SASL mechanism name, if any
      SASL_AUTHUSER     -- (r/o) pointer to a NUL terminated authentication/
admin user
                           (authentication id).
```

SASL_APPNAME -- (r/w) name of the calling application.
This name can be used for logging purposes and/or to construct
the name of the configuration file. This property is available
to both server and client applications.
When a server application specifies an non NULL appname parameter
in a call to sasl_server_init function, a SASL API implementation
MUST call sasl_setprop internally with the appname as the value
for the SASL_APPNAME property.
SASL_SSF_EXTERNAL -- (r/w) pointer to an unsigned integer that denotes SSF provided
by an external security layer (e.g. TLS). See section 3.5.3 for
more details.
SASL_SEC_PROPS -- (r/w) pointer to sasl_security_properties_t. See section 3.5.3
for detailed description.
SASL_AUTH_EXTERNAL-- (r/w) pointer to a NUL terminated authentication id provided by

an external security layer (e.g. TLS). See section 3.5.3 for more details.

7. References

7.1. Normative References

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

[SASL] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, Netscape Communications, October 1997. <<Update the reference>>

[RFC 2373] Hinden, R., Deering, S., "IP Version 6 Addressing Architecture", RFC 2373, July 1998.

[LANGTAGS] Alvestrand, H., "Tags for the Identification of Languages", RFC 3066, Cisco Systems, January 2001.

7.2. Informative References

[IMAP4] Crispin, M., "Internet Message Access Protocol - Version 4rev1", RFC 3501, University of Washington, March 2003.

[POP-AUTH] Myers, "POP3 AUTHentication command", RFC 1734, Carnegie Mellon, December 1994.

[GSSAPI] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[LDAP] <<>>

8. Acknowledgements

The editor would like to thank Rob Siemborski, Ken Murchison, Philip Guenther, Randy Presuhn, Simon Josefsson, Lawrence Greenfield and Greg Hudson for providing useful feedback and suggestions.

9. Author's and Editor's Addresses

Author:

Chris Newman
Sun Microsystems
1050 Lakes Drive

West Covina, CA 91790 USA

Email: Chris.Newman@Sun.COM

Editor:

Alexey Melnikov
Isode Ltd.
5 Castle Business Village,
36 Station Road,
Hampton, Middlesex,
United Kingdom, TW12 2BX

Email: alexey.melnikov@isode.com

10. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

A. Appendix A -- Design Goals

The design goals of the SASL C API are as follows:

- To be simple and practical to use.
- To provide related utility services in addition to core SASL functionality.
- To be reasonably extensible.
- To be suitable for use in a multi-threaded server or client.
- To avoid dependancies on a specific memory allocation system, thread package or network model.
- To be an independent service rather than a new layer.

B. SASL API Index

<<To be completed>>