Kerberos Plugin Module Developer Guide

Release 1.22.2

MIT

CONTENTS

1 Contents **3**

1.1 General plugin concepts . . . 3
1.2 Client preauthentication interface (clpreauth) . . . 4
1.3 KDC preauthentication interface (kdcpreauth) . . . 5
1.4 Credential cache selection interface (ccselect) . . . 6
1.5 Password quality interface (pwqual) . . . 6
1.6 KADM5 hook interface (kadm5_hook) . . . 6
1.7 kadmin authorization interface (kadm5_auth) . . . 7
1.8 Host-to-realm interface (hostrealm) . . . 7
1.9 Local authorization interface (localauth) . . . 8
1.10 Server location interface (locate) . . . 8
1.11 Configuration interface (profile) . . . 8
1.12 GSSAPI mechanism interface . . . 10
1.13 Internal pluggable interfaces . . . 11
1.14 PKINIT certificate authorization interface (certauth) . . . 12
1.15 KDC policy interface (kdcpolicy) . . . 12

Index **13**

Kerberos plugin modules allow increased control over MIT krb5 library and server behavior. This guide describes how to create dynamic plugin modules and the currently available pluggable interfaces.

See plugin_config for information on how to register dynamic plugin modules and how to enable and disable modules via krb5.conf(5).

CHAPTER

ONE

CONTENTS

1.1 General plugin concepts

A krb5 dynamic plugin module is a Unix shared object or Windows DLL. Typically, the source code for a dynamic plugin module should live in its own project with a build system using automake and libtool, or tools with similar functionality.

A plugin module must define a specific symbol name, which depends on the pluggable interface and module name. For most pluggable interfaces, the exported symbol is a function named `INTERFACE_MODULE_initvt`, where *INTERFACE* is the name of the pluggable interface and *MODULE* is the name of the module. For these interfaces, it is possible for one shared object or DLL to implement multiple plugin modules, either for the same pluggable interface or for different ones. For example, a shared object could implement both KDC and client preauthentication mechanisms, by exporting functions named `kdcpreauth_mymech_initvt` and `clpreauth_mymech_initvt`.

A plugin module implementation should include the header file <`krb5/INTERFACE_plugin.h`>, where *INTERFACE* is the name of the pluggable interface. For instance, a ccselect plugin module implementation should use `#include <krb5/ccselect_plugin.h>`.

initvt functions have the following prototype:

```
krb5_error_code interface_modname_initvt(krb5_context context,
                                         int maj_ver, int min_ver,
                                         krb5_plugin_vtable vtable);
```

and should do the following:

1. Check that the supplied maj_ver argument is supported by the module. If it is not supported, the function should return KRB5_PLUGIN_VER_NOTSUPP.
2. Cast the supplied vtable pointer to the structure type corresponding to the major version, as documented in the pluggable interface header file.
3. Fill in the structure fields with pointers to method functions and static data, stopping at the field indicated by the supplied minor version. Fields for unimplemented optional methods can be left alone; it is not necessary to initialize them to NULL.

In most cases, the context argument will not be used. The initvt function should not allocate memory; think of it as a glorified structure initializer. Each pluggable interface defines methods for allocating and freeing module state if doing so is necessary for the interface.

Pluggable interfaces typically include a **name** field in the vtable structure, which should be filled in with a pointer to a string literal containing the module name.

Here is an example of what an initvt function might look like for a fictional pluggable interface named fences, for a module named “wicker”:

```
krb5_error_code
fences_wicker_initvt(krb5_context context, int maj_ver,
                     int min_ver, krb5_plugin_vtable vtable)
{
    krb5_ccselect_vtable vt;

    if (maj_ver == 1) {
        krb5_fences_vtable vt = (krb5_fences_vtable)vtable;
        vt->name = "wicker";
        vt->slats = wicker_slats;
        vt->braces = wicker_braces;
    } else if (maj_ver == 2) {
        krb5_fences_vtable_v2 vt = (krb5_fences_vtable_v2)vtable;
        vt->name = "wicker";
        vt->material = wicker_material;
        vt->construction = wicker_construction;
        if (min_ver < 2)
            return 0;
        vt->footing = wicker_footing;
        if (min_ver < 3)
            return 0;
        vt->appearance = wicker_appearance;
    } else {
        return KRB5_PLUGIN_VER_NOTSUPP;
    }
    return 0;
}
```

1.1.1 Logging from KDC and kadmind plugin modules

Plugin modules for the KDC or kadmind daemons can write to the configured logging outputs (see logging) by calling the **com_err** function. The first argument (*whoami*) is ignored. If the second argument (*code*) is zero, the formatted message is logged at informational severity; otherwise, the formatted message is logged at error severity and includes the error message for the supplied code. Here are examples:

```
com_err("", 0, "Client message contains %d items", nitems);
com_err("", retval, "while decoding client message");
```

(The behavior described above is new in release 1.17. In prior releases, the *whoami* argument is included for some logging output types, the logged message does not include the usual header for some output types, and the severity for syslog outputs is configured as part of the logging specification, defaulting to error severity.)

1.2 Client preauthentication interface (clpreauth)

During an initial ticket request, a KDC may ask a client to prove its knowledge of the password before issuing an encrypted ticket, or to use credentials other than a password. This process is called preauthentication, and is described in RFC 4120 and RFC 6113. The clpreauth interface allows the addition of client support for preauthentication mechanisms beyond those included in the core MIT krb5 code base. For a detailed description of the clpreauth interface, see the header file <krb5/clpreauth_plugin.h> (or <krb5/preauth_plugin.h> before release 1.12).

A clpreauth module is generally responsible for:

- Supplying a list of preauth type numbers used by the module in the **pa_type_list** field of the vtable structure.

- Indicating what kind of preauthentication mechanism it implements, with the **flags** method. In the most common case, this method just returns `PA_REAL`, indicating that it implements a normal preauthentication type.
- Examining the padata information included in a PREAUTH_REQUIRED or MORE_PREAUTH_DATA_REQUIRED error and producing padata values for the next AS request. This is done with the **process** method.
- Examining the padata information included in a successful ticket reply, possibly verifying the KDC identity and computing a reply key. This is also done with the **process** method.
- For preauthentication types which support it, recovering from errors by examining the error data from the KDC and producing a padata value for another AS request. This is done with the **tryagain** method.
- Receiving option information (supplied by `kinit -X` or by an application), with the **gic_opts** method.

A clpreauth module can create and destroy per-library-context and per-request state objects by implementing the **init**, **fini**, **request_init**, and **request_fini** methods. Per-context state objects have the type krb5_clpreauth_moddata, and per-request state objects have the type krb5_clpreauth_modreq. These are abstract pointer types; a module should typically cast these to internal types for the state objects.

The **process** and **tryagain** methods have access to a callback function and handle (called a "rock") which can be used to get additional information about the current request, including the expected enctype of the AS reply, the FAST armor key, and the client long-term key (prompting for the user password if necessary). A callback can also be used to replace the AS reply key if the preauthentication mechanism computes one.

1.3 KDC preauthentication interface (kdcpreauth)

The kdcpreauth interface allows the addition of KDC support for preauthentication mechanisms beyond those included in the core MIT krb5 code base. For a detailed description of the kdcpreauth interface, see the header file <`krb5/kdcpreauth_plugin.h`> (or <`krb5/preauth_plugin.h`> before release 1.12).

A kdcpreauth module is generally responsible for:

- Supplying a list of preauth type numbers used by the module in the **pa_type_list** field of the vtable structure.
- Indicating what kind of preauthentication mechanism it implements, with the **flags** method. If the mechanism computes a new reply key, it must specify the `PA_REPLACES_KEY` flag. If the mechanism is generally only used with hardware tokens, the `PA_HARDWARE` flag allows the mechanism to work with principals which have the **requires_hwauth** flag set.
- Producing a padata value to be sent with a preauth_required error, with the **edata** method.
- Examining a padata value sent by a client and verifying that it proves knowledge of the appropriate client credential information. This is done with the **verify** method.
- Producing a padata response value for the client, and possibly computing a reply key. This is done with the **return_padata** method.

A module can create and destroy per-KDC state objects by implementing the **init** and **fini** methods. Per-KDC state objects have the type krb5_kdcpreauth_moddata, which is an abstract pointer types. A module should typically cast this to an internal type for the state object.

A module can create a per-request state object by returning one in the **verify** method, receiving it in the **return_padata** method, and destroying it in the **free_modreq** method. Note that these state objects only apply to the processing of a single AS request packet, not to an entire authentication exchange (since an authentication exchange may remain unfinished by the client or may involve multiple different KDC hosts). Per-request state objects have the type krb5_kdcpreauth_modreq, which is an abstract pointer type.

The **edata**, **verify**, and **return_padata** methods have access to a callback function and handle (called a "rock") which can be used to get additional information about the current request, including the maximum allowable clock skew, the client's long-term keys, the DER-encoded request body, the FAST armor key, string attributes on the client's database

entry, and the client's database entry itself. The **verify** method can assert one or more authentication indicators to be included in the issued ticket using the `add_auth_indicator` callback (new in release 1.14).

A module can generate state information to be included with the next client request using the `set_cookie` callback (new in release 1.14). On the next request, the module can read this state information using the `get_cookie` callback. Cookie information is encrypted, timestamped, and transmitted to the client in a `PA-FX-COOKIE` pa-data item. Older clients may not support cookies and therefore may not transmit the cookie in the next request; in this case, `get_cookie` will not yield the saved information.

If a module implements a mechanism which requires multiple round trips, its **verify** method can respond with the code `KRB5KDC_ERR_MORE_PREAUTH_DATA_REQUIRED` and a list of pa-data in the *e_data* parameter to be processed by the client.

The **edata** and **verify** methods can be implemented asynchronously. Because of this, they do not return values directly to the caller, but must instead invoke responder functions with their results. A synchronous implementation can invoke the responder function immediately. An asynchronous implementation can use the callback to get an event context for use with the libverto API.

1.4 Credential cache selection interface (ccselect)

The ccselect interface allows modules to control how credential caches are chosen when a GSSAPI client contacts a service. For a detailed description of the ccselect interface, see the header file <`krb5/ccselect_plugin.h`>.

The primary ccselect method is **choose**, which accepts a server principal as input and returns a ccache and/or principal name as output. A module can use the krb5_cccol APIs to iterate over the cache collection in order to find an appropriate ccache to use.

A module can create and destroy per-library-context state objects by implementing the **init** and **fini** methods. State objects have the type krb5_ccselect_moddata, which is an abstract pointer type. A module should typically cast this to an internal type for the state object.

A module can have one of two priorities, "authoritative" or "heuristic". Results from authoritative modules, if any are available, will take priority over results from heuristic modules. A module communicates its priority as a result of the **init** method.

1.5 Password quality interface (pwqual)

The pwqual interface allows modules to control what passwords are allowed when a user changes passwords. For a detailed description of the pwqual interface, see the header file <`krb5/pwqual_plugin.h`>.

The primary pwqual method is **check**, which receives a password as input and returns success (0) or a `KADM5_PASS_Q_` failure code depending on whether the password is allowed. The **check** method also receives the principal name and the name of the principal's password policy as input; although there is no stable interface for the module to obtain the fields of the password policy, it can define its own configuration or data store based on the policy name.

A module can create and destroy per-process state objects by implementing the **open** and **close** methods. State objects have the type krb5_pwqual_moddata, which is an abstract pointer type. A module should typically cast this to an internal type for the state object. The **open** method also receives the name of the realm's dictionary file (as configured by the **dict_file** variable in the kdc_realms section of kdc.conf(5)) if it wishes to use it.

1.6 KADM5 hook interface (kadm5_hook)

The kadm5_hook interface allows modules to perform actions when changes are made to the Kerberos database through kadmin(1). For a detailed description of the kadm5_hook interface, see the header file <`krb5/kadm5_hook_plugin.h`>.

The kadm5_hook interface has five primary methods: **chpass**, **create**, **modify**, **remove**, and **rename**. (The **rename** method was introduced in release 1.14.) Each of these methods is called twice when the corresponding administrative action takes place, once before the action is committed and once afterwards. A module can prevent the action from taking place by returning an error code during the pre-commit stage.

A module can create and destroy per-process state objects by implementing the **init** and **fini** methods. State objects have the type kadm5_hook_modinfo, which is an abstract pointer type. A module should typically cast this to an internal type for the state object.

Because the kadm5_hook interface is tied closely to the kadmin interface (which is explicitly unstable), it may not remain as stable across versions as other public pluggable interfaces.

1.7 kadmin authorization interface (kadm5_auth)

The kadm5_auth interface (new in release 1.16) allows modules to determine whether a client principal is authorized to perform an operation in the kadmin protocol, and to apply restrictions to principal operations. For a detailed description of the kadm5_auth interface, see the header file <krb5/kadm5_auth_plugin.h>.

A module can create and destroy per-process state objects by implementing the **init** and **fini** methods. State objects have the type kadm5_auth_modinfo, which is an abstract pointer type. A module should typically cast this to an internal type for the state object.

The kadm5_auth interface has one method for each kadmin operation, with parameters specific to the operation. Each method can return either 0 to authorize access, KRB5_PLUGIN_NO_HANDLE to defer the decision to other modules, or another error (canonically EPERM) to authoritatively deny access. Access is granted if at least one module grants access and no module authoritatively denies access.

The **addprinc** and **modprinc** methods can also impose restrictions on the principal operation by returning a `struct kadm5_auth_restrictions` object. The module should also implement the **free_restrictions** method if it dynamically allocates restrictions objects for principal operations.

kadm5_auth modules can optionally inspect principal or policy objects. To do this, the module must also include <kadm5/admin.h> to gain access to the structure definitions for those objects. As the kadmin interface is explicitly not as stable as other public interfaces, modules which do this may not retain compatibility across releases.

1.8 Host-to-realm interface (hostrealm)

The host-to-realm interface was first introduced in release 1.12. It allows modules to control the local mapping of hostnames to realm names as well as the default realm. For a detailed description of the hostrealm interface, see the header file <krb5/hostrealm_plugin.h>.

Although the mapping methods in the hostrealm interface return a list of one or more realms, only the first realm in the list is currently used by callers. Callers may begin using later responses in the future.

Any mapping method may return KRB5_PLUGIN_NO_HANDLE to defer processing to a later module.

A module can create and destroy per-library-context state objects using the **init** and **fini** methods. If the module does not need any state, it does not need to implement these methods.

The optional **host_realm** method allows a module to determine authoritative realm mappings for a hostname. The first authoritative mapping is used in preference to KDC referrals when getting service credentials.

The optional **fallback_realm** method allows a module to determine fallback mappings for a hostname. The first fallback mapping is tried if there is no authoritative mapping for a realm, and KDC referrals failed to produce a successful result.

The optional **default_realm** method allows a module to determine the local default realm.

If a module implements any of the above methods, it must also implement **free_list** to ensure that memory is allocated and deallocated consistently.

1.9 Local authorization interface (localauth)

The localauth interface was first introduced in release 1.12. It allows modules to control the relationship between Kerberos principals and local system accounts. When an application calls `krb5_kuserok()` or `krb5_aname_to_localname()`, localauth modules are consulted to determine the result. For a detailed description of the localauth interface, see the header file <`krb5/localauth_plugin.h`>.

A module can create and destroy per-library-context state objects using the **init** and **fini** methods. If the module does not need any state, it does not need to implement these methods.

The optional **userok** method allows a module to control the behavior of `krb5_kuserok()`. The module receives the authenticated name and the local account name as inputs, and can return either 0 to authorize access, KRB5_PLUGIN_NO_HANDLE to defer the decision to other modules, or another error (canonically EPERM) to authoritatively deny access. Access is granted if at least one module grants access and no module authoritatively denies access.

The optional **an2ln** method can work in two different ways. If the module sets an array of uppercase type names in **an2ln_types**, then the module's **an2ln** method will only be invoked by `krb5_aname_to_localname()` if an **auth_to_local** value in krb5.conf(5) refers to one of the module's types. In this case, the *type* and *residual* arguments will give the type name and residual string of the **auth_to_local** value.

If the module does not set **an2ln_types** but does implement **an2ln**, the module's **an2ln** method will be invoked for all `krb5_aname_to_localname()` operations unless an earlier module determines a mapping, with *type* and *residual* set to NULL. The module can return KRB5_LNAME_NO_TRANS to defer mapping to later modules.

If a module implements **an2ln**, it must also implement **free_string** to ensure that memory is allocated and deallocated consistently.

1.10 Server location interface (locate)

The locate interface allows modules to control how KDCs and similar services are located by clients. For a detailed description of the ccselect interface, see the header file <`krb5/locate_plugin.h`>.

A locate module exports a structure object of type krb5plugin_service_locate_ftable, with the name `service_locator`. The structure contains a minor version and pointers to the module's methods.

The primary locate method is **lookup**, which accepts a service type, realm name, desired socket type, and desired address family (which will be AF_UNSPEC if no specific address family is desired). The method should invoke the callback function once for each server address it wants to return, passing a socket type (SOCK_STREAM for TCP or SOCK_DGRAM for UDP) and socket address. The **lookup** method should return 0 if it has authoritatively determined the server addresses for the realm, KRB5_PLUGIN_NO_HANDLE if it wants to let other location mechanisms determine the server addresses, or another code if it experienced a failure which should abort the location process.

A module can create and destroy per-library-context state objects by implementing the **init** and **fini** methods. State objects have the type void *, and should be cast to an internal type for the state object.

1.11 Configuration interface (profile)

The profile interface allows a module to control how krb5 configuration information is obtained by the Kerberos library and applications. For a detailed description of the profile interface, see the header file <`profile.h`>.

> **Note**
>
> The profile interface does not follow the normal conventions for MIT krb5 pluggable interfaces, because it is part of a lower-level component of the krb5 library.

As with other types of plugin modules, a profile module is a Unix shared object or Windows DLL, built separately from the krb5 tree. The krb5 library will dynamically load and use a profile plugin module if it reads a `module` directive at the beginning of krb5.conf, as described in profile_plugin_config.

A profile module exports a function named `profile_module_init` matching the signature of the profile_module_init_fn type. This function accepts a residual string, which may be used to help locate the configuration source. The function fills in a vtable and may also create a per-profile state object. If the module uses state objects, it should implement the **copy** and **cleanup** methods to manage them.

A basic read-only profile module need only implement the **get_values** and **free_values** methods. The **get_values** method accepts a null-terminated list of C string names (e.g., an array containing "libdefaults", "clockskew", and NULL for the **clockskew** variable in the libdefaults section) and returns a null-terminated list of values, which will be cleaned up with the **free_values** method when the caller is done with them.

Iterable profile modules must also define the **iterator_create**, **iterator**, **iterator_free**, and **free_string** methods. The core krb5 code does not require profiles to be iterable, but some applications may iterate over the krb5 profile object in order to present configuration interfaces.

Writable profile modules must also define the **writable**, **modified**, **update_relation**, **rename_section**, **add_relation**, and **flush** methods. The core krb5 code does not require profiles to be writable, but some applications may write to the krb5 profile in order to present configuration interfaces.

The following is an example of a very basic read-only profile module which returns a hardcoded value for the **default_realm** variable in libdefaults, and provides no other configuration information. (For conciseness, the example omits code for checking the return values of malloc and strdup.)

```
#include <stdlib.h>
#include <string.h>
#include <profile.h>

static long
get_values(void *cbdata, const char *const *names, char ***values)
{
    if (names[0] != NULL && strcmp(names[0], "libdefaults") == 0 &&
        names[1] != NULL && strcmp(names[1], "default_realm") == 0) {
        *values = malloc(2 * sizeof(char *));
        (*values)[0] = strdup("ATHENA.MIT.EDU");
        (*values)[1] = NULL;
        return 0;
    }
    return PROF_NO_RELATION;
}

static void
free_values(void *cbdata, char **values)
{
    char **v;

    for (v = values; *v; v++)
        free(*v);
    free(values);
}

long
profile_module_init(const char *residual, struct profile_vtable *vtable,
                    void **cb_ret);
```

(continues on next page)

(continued from previous page)

```
long
profile_module_init(const char *residual, struct profile_vtable *vtable,
                    void **cb_ret)
{
    *cb_ret = NULL;
    vtable->get_values = get_values;
    vtable->free_values = free_values;
    return 0;
}
```

1.12 GSSAPI mechanism interface

The GSSAPI library in MIT krb5 can load mechanism modules to augment the set of built-in mechanisms.

A mechanism module is a Unix shared object or Windows DLL, built separately from the krb5 tree. Modules are loaded according to the GSS mechanism config files described in gssapi_plugin_config.

For the most part, a GSSAPI mechanism module exports the same functions as would a GSSAPI implementation itself, with the same function signatures. The mechanism selection layer within the GSSAPI library (called the "mechglue") will dispatch calls from the application to the module if the module's mechanism is requested. If a module does not wish to implement a GSSAPI extension, it can simply refrain from exporting it, and the mechglue will fail gracefully if the application calls that function.

The mechglue does not invoke a module's **gss_add_cred**, **gss_add_cred_from**, **gss_add_cred_impersonate_name**, or **gss_add_cred_with_password** function. A mechanism only needs to implement the "acquire" variants of those functions.

A module does not need to coordinate its minor status codes with those of other mechanisms. If the mechglue detects conflicts, it will map the mechanism's status codes onto unique values, and then map them back again when **gss_display_status** is called.

1.12.1 NegoEx modules

Some Windows GSSAPI mechanisms can only be negotiated via a Microsoft extension to SPNEGO called NegoEx. Beginning with release 1.18, mechanism modules can support NegoEx as follows:

- Implement the gssspi_query_meta_data(), gssspi_exchange_meta_data(), and gssspi_query_mechanism_info() SPIs declared in <gssapi/gssapi_ext.h>.
- Implement gss_inquire_sec_context_by_oid() and answer the **GSS_C_INQ_NEGOEX_KEY** and **GSS_C_INQ_NEGOEX_VERIFY_KEY** OIDs to provide the checksum keys for outgoing and incoming checksums, respectively. The answer must be in two buffers: the first buffer contains the key contents, and the second buffer contains the key encryption type as a four-byte little-endian integer.

By default, NegoEx mechanisms will not be directly negotiated via SPNEGO. If direct SPNEGO negotiation is required for interoperability, implement gss_inquire_attrs_for_mech() and assert the GSS_C_MA_NEGOEX_AND_SPNEGO attribute (along with any applicable RFC 5587 attributes).

1.12.2 Interposer modules

The mechglue also supports a kind of loadable module, called an interposer module, which intercepts calls to existing mechanisms rather than implementing a new mechanism.

An interposer module must export the symbol **gss_mech_interposer** with the following signature:

```
gss_OID_set gss_mech_interposer(gss_OID mech_type);
```

This function is invoked with the OID of the interposer mechanism as specified in the mechanism config file, and returns a set of mechanism OIDs to be interposed. The returned OID set must have been created using the mechglue's gss_create_empty_oid_set and gss_add_oid_set_member functions.

An interposer module must use the prefix `gssi_` for the GSSAPI functions it exports, instead of the prefix `gss_`. In most cases, unexported `gssi_` functions will result in failure from their corresponding `gss_` calls.

An interposer module can link against the GSSAPI library in order to make calls to the original mechanism. To do so, it must specify a special mechanism OID which is the concatention of the interposer's own OID byte string and the original mechanism's OID byte string.

Functions that do not accept a mechanism argument directly require no special handling, with the following exceptions:

Since **gss_accept_sec_context** does not accept a mechanism argument, an interposer mechanism must, in order to invoke the original mechanism's function, acquire a credential for the concatenated OID and pass that as the *verifier_cred_handle* parameter.

Since **gss_import_name**, **gss_import_cred**, and **gss_import_sec_context** do not accept mechanism parameters, the SPI has been extended to include variants which do. This allows the interposer module to know which mechanism should be used to interpret the token. These functions have the following signatures:

```
OM_uint32 gssi_import_sec_context_by_mech(OM_uint32 *minor_status,
    gss_OID desired_mech, gss_buffer_t interprocess_token,
    gss_ctx_id_t *context_handle);

OM_uint32 gssi_import_name_by_mech(OM_uint32 *minor_status,
    gss_OID mech_type, gss_buffer_t input_name_buffer,
    gss_OID input_name_type, gss_name_t output_name);

OM_uint32 gssi_import_cred_by_mech(OM_uint32 *minor_status,
    gss_OID mech_type, gss_buffer_t token,
    gss_cred_id_t *cred_handle);
```

To re-enter the original mechanism when importing tokens for the above functions, the interposer module must wrap the mechanism token in the mechglue's format, using the concatenated OID (except in **gss_import_name**). The mechglue token formats are:

- For **gss_import_sec_context**, a four-byte OID length in big-endian order, followed by the concatenated OID, followed by the mechanism token.
- For **gss_import_name**, the bytes 04 01, followed by a two-byte OID length in big-endian order, followed by the mechanism OID, followed by a four-byte token length in big-endian order, followed by the mechanism token. Unlike most uses of OIDs in the API, the mechanism OID encoding must include the DER tag and length for an object identifier (06 followed by the DER length of the OID byte string), and this prefix must be included in the two-byte OID length. input_name_type must also be set to GSS_C_NT_EXPORT_NAME.
- For **gss_import_cred**, a four-byte OID length in big-endian order, followed by the concatenated OID, followed by a four-byte token length in big-endian order, followed by the mechanism token. This sequence may be repeated multiple times.

1.13 Internal pluggable interfaces

Following are brief discussions of pluggable interfaces which have not yet been made public. These interfaces are functional, but the interfaces are likely to change in incompatible ways from release to release. In some cases, it may

be necessary to copy header files from the krb5 source tree to use an internal interface. Use these with care, and expect to need to update your modules for each new release of MIT krb5.

1.13.1 Kerberos database interface (KDB)

A KDB module implements a database back end for KDC principal and policy information, and can also control many aspects of KDC behavior. For a full description of the interface, see the header file <kdb.h>.

The KDB pluggable interface is often referred to as the DAL (Database Access Layer).

1.13.2 Authorization data interface (authdata)

The authdata interface allows a module to provide (from the KDC) or consume (in application servers) authorization data of types beyond those handled by the core MIT krb5 code base. The interface is defined in the header file <krb5/authdata_plugin.h>, which is not installed by the build.

1.14 PKINIT certificate authorization interface (certauth)

The certauth interface was first introduced in release 1.16. It allows customization of the X.509 certificate attribute requirements placed on certificates used by PKINIT enabled clients. For a detailed description of the certauth interface, see the header file <krb5/certauth_plugin.h>

A certauth module implements the **authorize** method to determine whether a client's certificate is authorized to authenticate a client principal. **authorize** receives the DER-encoded certificate, the requested client principal, and a pointer to the client's krb5_db_entry (for modules that link against libkdb5). The method must decode the certificate and inspect its attributes to determine if it should authorize PKINIT authentication. It returns the authorization status and optionally outputs a list of authentication indicator strings to be added to the ticket.

Beginning in release 1.19, the authorize method can request that the hardware authentication bit be set in the ticket by returning **KRB5_CERTAUTH_HWAUTH**. Beginning in release 1.20, the authorize method can return **KRB5_CERTAUTH_HWAUTH_PASS** to request that the hardware authentication bit be set in the ticket but otherwise defer authorization to another certauth module. A module must use its own internal or library-provided ASN.1 certificate decoder.

A module can optionally create and destroy module data with the **init** and **fini** methods. Module data objects last for the lifetime of the KDC process.

If a module allocates and returns a list of authentication indicators from **authorize**, it must also implement the **free_ind** method to free the list.

1.15 KDC policy interface (kdcpolicy)

The kdcpolicy interface was first introduced in release 1.16. It allows modules to veto otherwise valid AS and TGS requests or restrict the lifetime and renew time of the resulting ticket. For a detailed description of the kdcpolicy interface, see the header file <krb5/kdcpolicy_plugin.h>.

The optional **check_as** and **check_tgs** functions allow the module to perform access control. Additionally, a module can create and destroy module data with the **init** and **fini** methods. Module data objects last for the lifetime of the KDC process, and are provided to all other methods. The data has the type krb5_kdcpolicy_moddata, which should be cast to the appropriate internal type.

kdcpolicy modules can optionally inspect principal entries. To do this, the module must also include <kdb.h> to gain access to the principal entry structure definition. As the KDB interface is explicitly not as stable as other public interfaces, modules which do this may not retain compatibility across releases.

INDEX

R

RFC
    RFC 4120, 4
    RFC 6113, 4