Kerberos Application Developer Guide

Release 1.12.2

MIT

CHAPTER

ONE

DEVELOPING WITH GSSAPI

The GSSAPI (Generic Security Services API) allows applications to communicate securely using Kerberos 5 or other security mechanisms. We recommend using the GSSAPI (or a higher-level framework which encompasses GSSAPI, such as SASL) for secure network communication over using the libkrb5 API directly.

GSSAPIv2 is specified in **RFC 2743** and **RFC 2744**. This documentation will describe how various ways of using GSSAPI will behave with the krb5 mechanism as implemented in MIT krb5, as well as krb5-specific extensions to the GSSAPI.

1.1 Name types

A GSSAPI application can name a local or remote entity by calling gss_import_name, specifying a name type and a value. The following name types are supported by the krb5 mechanism:

- **GSS_C_NT_HOSTBASED_SERVICE**: The value should be a string of the form `service` or `service@hostname`. This is the most common way to name target services when initiating a security context, and is the most likely name type to work across multiple mechanisms.
- **GSS_KRB5_NT_PRINCIPAL_NAME**: The value should be a principal name string. This name type only works with the krb5 mechanism, and is defined in the `<gssapi_krb5.h>` header.
- **GSS_C_NT_USER_NAME** or **GSS_C_NULL_OID**: The value is treated as an unparsed principal name string, as above. These name types may work with mechanisms other than krb5, but will have different interpretations in those mechanisms. **GSS_C_NT_USER_NAME** is intended to be used with a local username, which will parse into a single-component principal in the default realm.
- **GSS_C_NT_ANONYMOUS**: The value is ignored. The anonymous principal is used, allowing a client to authenticate to a server without asserting a particular identity (which may or may not be allowed by a particular server or Kerberos realm).
- **GSS_C_NT_MACHINE_UID_NAME**: The value is uid_t object. On Unix-like systems, the username of the uid is looked up in the system user database and the resulting username is parsed as a principal name.
- **GSS_C_NT_STRING_UID_NAME**: As above, but the value is a decimal string representation of the uid.
- **GSS_C_NT_EXPORT_NAME**: The value must be the result of a gss_export_name call.

1.2 Initiator credentials

A GSSAPI client application uses gss_init_sec_context to establish a security context. The *initiator_cred_handle* parameter determines what tickets are used to establish the connection. An application can either pass

GSS_C_NO_CREDENTIAL to use the default client credential, or it can use gss_acquire_cred beforehand to acquire an initiator credential. The call to gss_acquire_cred may include a *desired_name* parameter, or it may pass **GSS_C_NO_NAME** if it does not have a specific name preference.

If the desired name for a krb5 initiator credential is a host-based name, it is converted to a principal name of the form `service/hostname` in the local realm, where *hostname* is the local hostname if not specified. The hostname will be canonicalized using forward name resolution, and possibly also using reverse name resolution depending on the value of the **rdns** variable in *libdefaults*.

If a desired name is specified in the call to gss_acquire_cred, the krb5 mechanism will attempt to find existing tickets for that client principal name in the default credential cache or collection. If the default cache type does not support a collection, and the default cache contains credentials for a different principal than the desired name, a **GSS_S_CRED_UNAVAIL** error will be returned with a minor code indicating a mismatch.

If no existing tickets are available for the desired name, but the name has an entry in the default client *keytab_definition*, the krb5 mechanism will acquire initial tickets for the name using the default client keytab.

If no desired name is specified, credential acquisition will be deferred until the credential is used in a call to gss_init_sec_context or gss_inquire_cred. If the call is to gss_init_sec_context, the target name will be used to choose a client principal name using the credential cache selection facility. (This facility might, for instance, try to choose existing tickets for a client principal in the same realm as the target service). If there are no existing tickets for the chosen principal, but it is present in the default client keytab, the krb5 mechanism will acquire initial tickets using the keytab.

If the target name cannot be used to select a client principal (because the credentials are used in a call to gss_inquire_cred), or if the credential cache selection facility cannot choose a principal for it, the default credential cache will be selected if it exists and contains tickets.

If the default credential cache does not exist, but the default client keytab does, the krb5 mechanism will try to acquire initial tickets for the first principal in the default client keytab.

If the krb5 mechanism acquires initial tickets using the default client keytab, the resulting tickets will be stored in the default cache or collection, and will be refreshed by future calls to gss_acquire_cred as they approach their expire time.

1.3 Acceptor names

A GSSAPI server application uses gss_accept_sec_context to establish a security context based on tokens provided by the client. The *acceptor_cred_handle* parameter determines what *keytab_definition* entries may be authenticated to by the client, if the krb5 mechanism is used.

The simplest choice is to pass **GSS_C_NO_CREDENTIAL** as the acceptor credential. In this case, clients may authenticate to any service principal in the default keytab (typically `FILE:/etc/krb5.keytab`, or the value of the **KRB5_KTNAME** environment variable). This is the recommended approach if the server application has no specific requirements to the contrary.

A server may acquire an acceptor credential with gss_acquire_cred and a *cred_usage* of **GSS_C_ACCEPT** or **GSS_C_BOTH**. If the *desired_name* parameter is **GSS_C_NO_NAME**, then clients will be allowed to authenticate to any service principal in the default keytab, just as if no acceptor credential was supplied.

If a server wishes to specify a *desired_name* to gss_acquire_cred, the most common choice is a host-based name. If the host-based *desired_name* contains just a *service*, then clients will be allowed to authenticate to any host-based service principal (that is, a principal of the form `service/hostname@REALM`) for the named service, regardless of hostname or realm, as long as it is present in the default keytab. If the input name contains both a *service* and a *hostname*, clients will be allowed to authenticate to any host-based principal for the named service and hostname, regardless of realm.

Note: If a *hostname* is specified, it will be canonicalized using forward name resolution, and possibly also using reverse name resolution depending on the value of the **rdns** variable in *libdefaults*.

Note: If the **ignore_acceptor_hostname** variable in *libdefaults* is enabled, then *hostname* will be ignored even if one is specified in the input name.

Note: In MIT krb5 versions prior to 1.10, and in Heimdal's implementation of the krb5 mechanism, an input name with just a *service* is treated like an input name of `service@localhostname`, where *localhostname* is the string returned by gethostname().

If the *desired_name* is a krb5 principal name or a local system name type which is mapped to a krb5 principal name, clients will only be allowed to authenticate to that principal in the default keytab.

1.4 Importing and exporting credentials

The following GSSAPI extensions can be used to import and export credentials (declared in `<gssapi/gssapi_ext.h>`):

```
OM_uint32 gss_export_cred(OM_uint32 *minor_status,
                          gss_cred_id_t cred_handle,
                          gss_buffer_t token);

OM_uint32 gss_import_cred(OM_uint32 *minor_status,
                          gss_buffer_t token,
                          gss_cred_id_t *cred_handle);
```

The first function serializes a GSSAPI credential handle into a buffer; the second unseralizes a buffer into a GSSAPI credential handle. Serializing a credential does not destroy it. If any of the mechanisms used in *cred_handle* do not support serialization, gss_export_cred will return **GSS_S_UNAVAILABLE**. As with other GSSAPI serialization functions, these extensions are only intended to work with a matching implementation on the other side; they do not serialize credentials in a standardized format.

A serialized credential may contain secret information such as ticket session keys. The serialization format does not protect this information from eavesdropping or tampering. The calling application must take care to protect the serialized credential when communicating it over an insecure channel or to an untrusted party.

A krb5 GSSAPI credential may contain references to a credential cache, a client keytab, an acceptor keytab, and a replay cache. These resources are normally serialized as references to their external locations (such as the filename of the credential cache). Because of this, a serialized krb5 credential can only be imported by a process with similar privileges to the exporter. A serialized credential should not be trusted if it originates from a source with lower privileges than the importer, as it may contain references to external credential cache, keytab, or replay cache resources not accessible to the originator.

An exception to the above rule applies when a krb5 GSSAPI credential refers to a memory credential cache, as is normally the case for delegated credentials received by gss_accept_sec_context. In this case, the contents of the credential cache are serialized, so that the resulting token may be imported even if the original memory credential cache no longer exists.

1.5 AEAD message wrapping

The following GSSAPI extensions (declared in `<gssapi/gssapi_ext.h>`) can be used to wrap and unwrap messages with additional "associated data" which is integrity-checked but is not included in the output buffer:

```
OM_uint32 gss_wrap_aead(OM_uint32 *minor_status,
                        gss_ctx_id_t context_handle,
                        int conf_req_flag, gss_qop_t qop_req,
                        gss_buffer_t input_assoc_buffer,
                        gss_buffer_t input_payload_buffer,
                        int *conf_state,
                        gss_buffer_t output_message_buffer);

OM_uint32 gss_unwrap_aead(OM_uint32 *minor_status,
                          gss_ctx_id_t context_handle,
                          gss_buffer_t input_message_buffer,
                          gss_buffer_t input_assoc_buffer,
                          gss_buffer_t output_payload_buffer,
                          int *conf_state,
                          gss_qop_t *qop_state);
```

Wrap tokens created with gss_wrap_aead will successfully unwrap only if the same *input_assoc_buffer* contents are presented to gss_unwrap_aead.

1.6 IOV message wrapping

The following extensions (declared in <gssapi/gssapi_ext.h>) can be used for in-place encryption, fine-grained control over wrap token layout, and for constructing wrap tokens compatible with Microsoft DCE RPC:

```
typedef struct gss_iov_buffer_desc_struct {
    OM_uint32 type;
    gss_buffer_desc buffer;
} gss_iov_buffer_desc, *gss_iov_buffer_t;

OM_uint32 gss_wrap_iov(OM_uint32 *minor_status,
                       gss_ctx_id_t context_handle,
                       int conf_req_flag, gss_qop_t qop_req,
                       int *conf_state,
                       gss_iov_buffer_desc *iov, int iov_count);

OM_uint32 gss_unwrap_iov(OM_uint32 *minor_status,
                         gss_ctx_id_t context_handle,
                         int *conf_state, gss_qop_t *qop_state,
                         gss_iov_buffer_desc *iov, int iov_count);

OM_uint32 gss_wrap_iov_length(OM_uint32 *minor_status,
                              gss_ctx_id_t context_handle,
                              int conf_req_flag,
                              gss_qop_t qop_req, int *conf_state,
                              gss_iov_buffer_desc *iov,
                              int iov_count);

OM_uint32 gss_release_iov_buffer(OM_uint32 *minor_status,
                                 gss_iov_buffer_desc *iov,
                                 int iov_count);
```

The caller of gss_wrap_iov provides an array of gss_iov_buffer_desc structures, each containing a type and a gss_buffer_desc structure. Valid types include:

- **GSS_C_BUFFER_TYPE_DATA**: A data buffer to be included in the token, and to be encrypted or decrypted in-place if the token is confidentiality-protected.

- **GSS_C_BUFFER_TYPE_HEADER**: The GSSAPI wrap token header and underlying cryptographic header.
- **GSS_C_BUFFER_TYPE_TRAILER**: The cryptographic trailer, if one is required.
- **GSS_C_BUFFER_TYPE_PADDING**: Padding to be combined with the data during encryption and decryption. (The implementation may choose to place padding in the trailer buffer, in which case it will set the padding buffer length to 0.)
- **GSS_C_BUFFER_TYPE_STREAM**: For unwrapping only, a buffer containing a complete wrap token in standard format to be unwrapped.
- **GSS_C_BUFFER_TYPE_SIGN_ONLY**: A buffer to be included in the token's integrity protection checksum, but not to be encrypted or included in the token itself.

For gss_wrap_iov, the IOV list should contain one HEADER buffer, followed by zero or more SIGN_ONLY buffers, followed by one or more DATA buffers, followed by a TRAILER buffer. The memory pointed to by the buffers is not required to be contiguous or in any particular order. If *conf_req_flag* is true, DATA buffers will be encrypted in-place, while SIGN_ONLY buffers will not be modified.

The type of an output buffer may be combined with **GSS_C_BUFFER_FLAG_ALLOCATE** to request that gss_wrap_iov allocate the buffer contents. If gss_wrap_iov allocates a buffer, it sets the **GSS_C_BUFFER_FLAG_ALLOCATED** flag on the buffer type. gss_release_iov_buffer can be used to release all allocated buffers within an iov list and unset their allocated flags. Here is an example of how gss_wrap_iov can be used with allocation requested (*ctx* is assumed to be a previously established gss_ctx_id_t):

```
OM_uint32 major, minor;
gss_iov_buffer_desc iov[4];
char str[] = "message";

iov[0].type = GSS_IOV_BUFFER_TYPE_HEADER | GSS_IOV_BUFFER_FLAG_ALLOCATE;
iov[1].type = GSS_IOV_BUFFER_TYPE_DATA;
iov[1].buffer.value = str;
iov[1].buffer.length = strlen(str);
iov[2].type = GSS_IOV_BUFFER_TYPE_PADDING | GSS_IOV_BUFFER_FLAG_ALLOCATE;
iov[3].type = GSS_IOV_BUFFER_TYPE_TRAILER | GSS_IOV_BUFFER_FLAG_ALLOCATE;

major = gss_wrap_iov(&minor, ctx, 1, GSS_C_QOP_DEFAULT, NULL,
                     iov, 4);
if (GSS_ERROR(major))
    handle_error(major, minor);

/* Transmit or otherwise use resulting buffers. */

(void)gss_release_iov_buffer(&minor, iov, 4);
```

If the caller does not choose to request buffer allocation by gss_wrap_iov, it should first call gss_wrap_iov_length to query the lengths of the HEADER, PADDING, and TRAILER buffers. DATA buffers must be provided in the iov list so that padding length can be computed correctly, but the output buffers need not be initialized. Here is an example of using gss_wrap_iov_length and gss_wrap_iov:

```
OM_uint32 major, minor;
gss_iov_buffer_desc iov[4];
char str[1024] = "message", *ptr;

iov[0].type = GSS_IOV_BUFFER_TYPE_HEADER;
iov[1].type = GSS_IOV_BUFFER_TYPE_DATA;
iov[1].buffer.value = str;
iov[1].buffer.length = strlen(str);

iov[2].type = GSS_IOV_BUFFER_TYPE_PADDING;
```

```
iov[3].type = GSS_IOV_BUFFER_TYPE_TRAILER;

major = gss_wrap_iov_length(&minor, ctx, 1, GSS_C_QOP_DEFAULT,
                            NULL, iov, 4);
if (GSS_ERROR(major))
    handle_error(major, minor);
if (strlen(str) + iov[0].buffer.length + iov[2].buffer.length +
    iov[3].buffer.length > sizeof(str))
    handle_out_of_space_error();
ptr = str + strlen(str);
iov[0].buffer.value = ptr;
ptr += iov[0].buffer.length;
iov[2].buffer.value = ptr;
ptr += iov[2].buffer.length;
iov[3].buffer.value = ptr;

major = gss_wrap_iov(&minor, ctx, 1, GSS_C_QOP_DEFAULT, NULL,
                     iov, 4);
if (GSS_ERROR(major))
    handle_error(major, minor);
```

If the context was established using the **GSS_C_DCE_STYLE** flag (described in **RFC 4757**), wrap tokens compatible with Microsoft DCE RPC can be constructed. In this case, the IOV list must include a SIGN_ONLY buffer, a DATA buffer, a second SIGN_ONLY buffer, and a HEADER buffer in that order (the order of the buffer contents remains arbitrary). The application must pad the DATA buffer to a multiple of 16 bytes as no padding or trailer buffer is used.

gss_unwrap_iov may be called with an IOV list just like one which would be provided to gss_wrap_iov. DATA buffers will be decrypted in-place if they were encrypted, and SIGN_ONLY buffers will not be modified.

Alternatively, gss_unwrap_iov may be called with a single STREAM buffer, zero or more SIGN_ONLY buffers, and a single DATA buffer. The STREAM buffer is interpreted as a complete wrap token. The STREAM buffer will be modified in-place to decrypt its contents. The DATA buffer will be initialized to point to the decrypted data within the STREAM buffer, unless it has the **GSS_C_BUFFER_FLAG_ALLOCATE** flag set, in which case it will be initialized with a copy of the decrypted data. Here is an example (*token* and *token_len* are assumed to be a pre-existing pointer and length for a modifiable region of data):

```
OM_uint32 major, minor;
gss_iov_buffer_desc iov[2];

iov[0].type = GSS_IOV_BUFFER_TYPE_STREAM;
iov[0].buffer.value = token;
iov[0].buffer.length = token_len;
iov[1].type = GSS_IOV_BUFFER_TYPE_DATA;
major = gss_unwrap_iov(&minor, ctx, NULL, NULL, iov, 2);
if (GSS_ERROR(major))
    handle_error(major, minor);

/* Decrypted data is in iov[1].buffer, pointing to a subregion of
 * token. */
```

1.7 IOV MIC tokens

The following extensions (declared in <gssapi/gssapi_ext.h>) can be used in release 1.12 or later to construct and verify MIC tokens using an IOV list:

```
OM_uint32 gss_get_mic_iov(OM_uint32 *minor_status,
                          gss_ctx_id_t context_handle,
                          gss_qop_t qop_req,
                          gss_iov_buffer_desc *iov,
                          int iov_count);

OM_uint32 gss_get_mic_iov_length(OM_uint32 *minor_status,
                                 gss_ctx_id_t context_handle,
                                 gss_qop_t qop_req,
                                 gss_iov_buffer_desc *iov,
                                 iov_count);

OM_uint32 gss_verify_mic_iov(OM_uint32 *minor_status,
                             gss_ctx_id_t context_handle,
                             gss_qop_t *qop_state,
                             gss_iov_buffer_desc *iov,
                             int iov_count);
```

The caller of gss_get_mic_iov provides an array of gss_iov_buffer_desc structures, each containing a type and a gss_buffer_desc structure. Valid types include:

- **GSS_C_BUFFER_TYPE_DATA** and **GSS_C_BUFFER_TYPE_SIGN_ONLY**: The corresponding buffer for each of these types will be signed for the MIC token, in the order provided.
- **GSS_C_BUFFER_TYPE_MIC_TOKEN**: The GSSAPI MIC token.

The type of the MIC_TOKEN buffer may be combined with **GSS_C_BUFFER_FLAG_ALLOCATE** to request that gss_get_mic_iov allocate the buffer contents. If gss_get_mic_iov allocates the buffer, it sets the **GSS_C_BUFFER_FLAG_ALLOCATED** flag on the buffer type. gss_release_iov_buffer can be used to release all allocated buffers within an iov list and unset their allocated flags. Here is an example of how gss_get_mic_iov can be used with allocation requested (*ctx* is assumed to be a previously established gss_ctx_id_t):

```
OM_uint32 major, minor;
gss_iov_buffer_desc iov[3];

iov[0].type = GSS_IOV_BUFFER_TYPE_DATA;
iov[0].buffer.value = "sign1";
iov[0].buffer.length = 5;
iov[1].type = GSS_IOV_BUFFER_TYPE_SIGN_ONLY;
iov[1].buffer.value = "sign2";
iov[1].buffer.length = 5;
iov[2].type = GSS_IOV_BUFFER_TYPE_MIC_TOKEN | GSS_IOV_BUFFER_FLAG_ALLOCATE;

major = gss_get_mic_iov(&minor, ctx, GSS_C_QOP_DEFAULT, iov, 3);
if (GSS_ERROR(major))
    handle_error(major, minor);

/* Transmit or otherwise use iov[2].buffer. */

(void)gss_release_iov_buffer(&minor, iov, 3);
```

If the caller does not choose to request buffer allocation by gss_get_mic_iov, it should first call gss_get_mic_iov_length to query the length of the MIC_TOKEN buffer. Here is an example of using gss_get_mic_iov_length and gss_get_mic_iov:

```
OM_uint32 major, minor;
gss_iov_buffer_desc iov[2];
char data[1024];
```

```
iov[0].type = GSS_IOV_BUFFER_TYPE_MIC_TOKEN;
iov[1].type = GSS_IOV_BUFFER_TYPE_DATA;
iov[1].buffer.value = "message";
iov[1].buffer.length = 7;

major = gss_wrap_iov_length(&minor, ctx, 1, GSS_C_QOP_DEFAULT,
                            NULL, iov, 2);
if (GSS_ERROR(major))
    handle_error(major, minor);
if (iov[0].buffer.length > sizeof(data))
    handle_out_of_space_error();
iov[0].buffer.value = data;

major = gss_wrap_iov(&minor, ctx, 1, GSS_C_QOP_DEFAULT, NULL,
                     iov, 2);
if (GSS_ERROR(major))
    handle_error(major, minor);
```

CHAPTER

TWO

DIFFERENCES BETWEEN HEIMDAL AND MIT KERBEROS API

`krb5_auth_con_getaddrs()`	H5l: If either of the pointers to local_addr and remote_addr is not NULL, it is freed firs
`krb5_auth_con_setaddrs()`	H5l: If either address is NULL, the previous address remains in place
`krb5_auth_con_setports()`	H5l: Not implemented as of version 1.3.3
`krb5_auth_con_setrecvsubkey()`	H5l: If either port is NULL, the previous port remains in place
`krb5_auth_con_setsendsubkey()`	H5l: Not implemented as of version 1.3.3
`krb5_cc_set_config()`	MIT: Before version 1.10 it was assumed that the last argument *data* is ALWAYS non-z
`krb5_cccol_last_change_time()`	H5l takes 3 arguments: krb5_context context, const char *type, krb5_timestamp *chang
`krb5_set_default_realm()`	H5l: Caches the computed default realm context field. If the second argument is NULL

CHAPTER

THREE

INITIAL CREDENTIALS

Software that performs tasks such as logging users into a computer when they type their Kerberos password needs to get initial credentials (usually ticket granting tickets) from Kerberos. Such software shares some behavior with the *kinit(1)* program.

Whenever a program grants access to a resource (such as a local login session on a desktop computer) based on a user successfully getting initial Kerberos credentials, it must verify those credentials against a secure shared secret (e.g., a host keytab) to ensure that the user credentials actually originate from a legitimate KDC. Failure to perform this verification is a critical vulnerability, because a malicious user can execute the "Zanarotti attack": the user constructs a fake response that appears to come from the legitimate KDC, but whose contents come from an attacker-controlled KDC.

Some applications read a Kerberos password over the network (ideally over a secure channel), which they then verify against the KDC. While this technique may be the only practical way to integrate Kerberos into some existing legacy systems, its use is contrary to the original design goals of Kerberos.

The function `krb5_get_init_creds_password()` will get initial credentials for a client using a password. An application that needs to verify the credentials can call `krb5_verify_init_creds()`. Here is an example of code to obtain and verify TGT credentials, given strings *princname* and *password* for the client principal name and password:

```
krb5_error_code ret;
krb5_creds creds;
krb5_principal client_princ = NULL;

memset(&creds, 0, sizeof(creds));
ret = krb5_parse_name(context, princname, &client_princ);
if (ret)
    goto cleanup;
ret = krb5_get_init_creds_password(context, &creds, client_princ,
                                   password, NULL, NULL, 0, NULL, NULL);
if (ret)
    goto cleanup;
ret = krb5_verify_init_creds(context, &creds, NULL, NULL, NULL, NULL);

cleanup:
krb5_free_principal(context, client_princ);
krb5_free_cred_contents(context, &creds);
return ret;
```

3.1 Options for get_init_creds

The function `krb5_get_init_creds_password()` takes an options parameter (which can be a null pointer). Use the function `krb5_get_init_creds_opt_alloc()` to allocate an options structure, and `krb5_get_init_creds_opt_free()` to free it. For example:

```
krb5_error_code ret;
krb5_get_init_creds_opt *opt = NULL;
krb5_creds creds;

memset(&creds, 0, sizeof(creds));
ret = krb5_get_init_creds_opt_alloc(context, &opt);
if (ret)
    goto cleanup;
krb5_get_init_creds_opt_set_tkt_life(opt, 24 * 60 * 60);
ret = krb5_get_init_creds_password(context, &creds, client_princ,
                                   password, NULL, NULL, 0, NULL, opt);
if (ret)
    goto cleanup;

cleanup:
krb5_get_init_creds_opt_free(context, opt);
krb5_free_cred_contents(context, &creds);
return ret;
```

3.2 Getting anonymous credentials

As of release 1.8, it is possible to obtain fully anonymous or partially anonymous (realm-exposed) credentials, if the KDC supports it. The MIT KDC supports issuing fully anonymous credentials as of release 1.8 if configured appropriately (see *anonymous_pkinit*), but does not support issuing realm-exposed anonymous credentials at this time.

To obtain fully anonymous credentials, call `krb5_get_init_creds_opt_set_anonymous()` on the options structure to set the anonymous flag, and specify a client principal with the KDC's realm and a single empty data component (the principal obtained by parsing `@`*realmname*). Authentication will take place using anonymous PKINIT; if successful, the client principal of the resulting tickets will be `WELLKNOWN/ANONYMOUS@WELLKNOWN:ANONYMOUS`. Here is an example:

```
krb5_get_init_creds_opt_set_anonymous(opt, 1);
ret = krb5_build_principal(context, &client_princ, strlen(myrealm),
                           myrealm, "", (char *)NULL);
if (ret)
    goto cleanup;
ret = krb5_get_init_creds_password(context, &creds, client_princ,
                                   password, NULL, NULL, 0, NULL, opt);
if (ret)
    goto cleanup;
```

To obtain realm-exposed anonymous credentials, set the anonymous flag on the options structure as above, but specify a normal client principal in order to prove membership in the realm. Authentication will take place as it normally does; if successful, the client principal of the resulting tickets will be `WELLKNOWN/ANONYMOUS@`*realmname*.

3.3 User interaction

Authenticating a user usually requires the entry of secret information, such as a password. A password can be supplied directly to `krb5_get_init_creds_password()` via the *password* parameter, or the application can supply prompter and/or responder callbacks instead. If callbacks are used, the user can also be queried for other secret information such as a PIN, informed of impending password expiration, or prompted to change a password which has expired.

3.3.1 Prompter callback

A prompter callback can be specified via the *prompter* and *data* parameters to `krb5_get_init_creds_password()`. The prompter will be invoked each time the krb5 library has a question to ask or information to present. When the prompter callback is invoked, the *banner* argument (if not null) is intended to be displayed to the user, and the questions to be answered are specified in the *prompts* array. Each prompt contains a text question in the *prompt* field, a *hidden* bit to indicate whether the answer should be hidden from display, and a storage area for the answer in the *reply* field. The callback should fill in each question's `reply->data` with the answer, up to a maximum number of `reply->length` bytes, and then reset `reply->length` to the length of the answer.

A prompter callback can call `krb5_get_prompt_types()` to get an array of type constants corresponding to the prompts, to get programmatic information about the semantic meaning of the questions. `krb5_get_prompt_types()` may return a null pointer if no prompt type information is available.

Text-based applications can use a built-in text prompter implementation by supplying `krb5_prompter_posix()` as the *prompter* parameter and a null pointer as the *data* parameter. For example:

```
ret = krb5_get_init_creds_password(context, &creds, client_princ,
                                   NULL, krb5_prompter_posix, NULL, 0,
                                   NULL, NULL);
```

3.3.2 Responder callback

A responder callback can be specified through the init_creds options using the `krb5_get_init_creds_opt_set_responder()` function. Responder callbacks can present a more sophisticated user interface for authentication secrets. The responder callback is usually invoked only once per authentication, with a list of questions produced by all of the allowed preauthentication mechanisms.

When the responder callback is invoked, the *rctx* argument can be accessed to obtain the list of questions and to answer them. The `krb5_responder_list_questions()` function retrieves an array of question types. For each question type, the `krb5_responder_get_challenge()` function retrieves additional information about the question, if applicable, and the `krb5_responder_set_answer()` function sets the answer.

Responder question types, challenges, and answers are UTF-8 strings. The question type is a well-known string; the meaning of the challenge and answer depend on the question type. If an application does not understand a question type, it cannot interpret the challenge or provide an answer. Failing to answer a question typically results in the prompter callback being used as a fallback.

Password question

The `KRB5_RESPONDER_QUESTION_PASSWORD` (or `"password"`) question type requests the user's password. This question does not have a challenge, and the response is simply the password string.

One-time password question

The KRB5_RESPONDER_QUESTION_OTP (or "otp") question type requests a choice among one-time password tokens and the PIN and value for the chosen token. The challenge and answer are JSON-encoded strings, but an application can use convenience functions to avoid doing any JSON processing itself.

The krb5_responder_otp_get_challenge() function decodes the challenge into a krb5_responder_otp_challenge structure. The krb5_responder_otp_set_answer() function selects one of the token information elements from the challenge and supplies the value and pin for that token.

PKINIT password or PIN question

The KRB5_RESPONDER_QUESTION_PKINIT (or "pkinit") question type requests PINs for hardware devices and/or passwords for encrypted credentials which are stored on disk, potentially also supplying information about the state of the hardware devices. The challenge and answer are JSON-encoded strings, but an application can use convenience functions to avoid doing any JSON processing itself.

The krb5_responder_pkinit_get_challenge() function decodes the challenges into a krb5_responder_pkinit_challenge structure. The krb5_responder_pkinit_set_answer() function can be used to supply the PIN or password for a particular client credential, and can be called multiple times.

Example

Here is an example of using a responder callback:

```
static krb5_error_code
my_responder(krb5_context context, void *data,
             krb5_responder_context rctx)
{
    krb5_error_code ret;
    krb5_responder_otp_challenge *chl;

    if (krb5_responder_get_challenge(context, rctx,
                                     KRB5_RESPONDER_QUESTION_PASSWORD)) {
        ret = krb5_responder_set_answer(context, rctx,
                                        KRB5_RESPONDER_QUESTION_PASSWORD,
                                        "open sesame");
        if (ret)
            return ret;
    }
    ret = krb5_responder_otp_get_challenge(context, rctx, &chl);
    if (ret == 0 && chl != NULL) {
        ret = krb5_responder_otp_set_answer(context, rctx, 0, "1234",
                                            NULL);
        krb5_responder_otp_challenge_free(context, rctx, chl);
        if (ret)
            return ret;
    }
    return 0;
}

static krb5_error_code
get_creds(krb5_context context, krb5_principal client_princ)
{
    krb5_error_code ret;
    krb5_get_init_creds_opt *opt = NULL;
```

```
    krb5_creds creds;

    memset(&creds, 0, sizeof(creds));
    ret = krb5_get_init_creds_opt_alloc(context, &opt);
    if (ret)
        goto cleanup;
    ret = krb5_get_init_creds_opt_set_responder(context, opt, my_responder,
                                                NULL);
    if (ret)
        goto cleanup;
    ret = krb5_get_init_creds_password(context, &creds, client_princ,
                                       NULL, NULL, NULL, 0, NULL, opt);

cleanup:
    krb5_get_init_creds_opt_free(context, opt);
    krb5_free_cred_contents(context, &creds);
    return ret;
}
```

3.4 Verifying initial credentials

Use the function `krb5_verify_init_creds()` to verify initial credentials. It takes an options structure (which can be a null pointer). Use `krb5_verify_init_creds_opt_init()` to initialize the caller-allocated options structure, and `krb5_verify_init_creds_opt_set_ap_req_nofail()` to set the "nofail" option. For example:

```
krb5_verify_init_creds_opt vopt;

krb5_verify_init_creds_opt_init(&vopt);
krb5_verify_init_creds_opt_set_ap_req_nofail(&vopt, 1);
ret = krb5_verify_init_creds(context, &creds, NULL, NULL, NULL, &vopt);
```

The confusingly named "nofail" option, when set, means that the verification must actually succeed in order for `krb5_verify_init_creds()` to indicate success. The default state of this option (cleared) means that if there is no key material available to verify the user credentials, the verification will succeed anyway. (The default can be changed by a configuration file setting.)

This accommodates a use case where a large number of unkeyed shared desktop workstations need to allow users to log in using Kerberos. The security risks from this practice are mitigated by the absence of valuable state on the shared workstations—any valuable resources that the users would access reside on networked servers.

CHAPTER

FOUR

PRINCIPAL MANIPULATION AND PARSING

Kerberos principal structure

`krb5_principal_data`

`krb5_principal`

Create and free principal

`krb5_build_principal()`

`krb5_build_principal_alloc_va()`

`krb5_build_principal_ext()`

`krb5_copy_principal()`

`krb5_free_principal()`

`krb5_cc_get_principal()`

Comparing

`krb5_principal_compare()`

`krb5_principal_compare_flags()`

`krb5_principal_compare_any_realm()`

`krb5_sname_match()`

`krb5_sname_to_principal()`

Parsing:

`krb5_parse_name()`

`krb5_parse_name_flags()`

`krb5_unparse_name()`

`krb5_unparse_name_flags()`

Utilities:

`krb5_is_config_principal()`

`krb5_kuserok()`

`krb5_set_password()`

`krb5_set_password_using_ccache()`

`krb5_set_principal_realm()`

`krb5_realm_compare()`

CHAPTER

FIVE

COMPLETE REFERENCE - API AND DATATYPES

5.1 krb5 API

5.1.1 Frequently used public interfaces

5.1.2 Rarely used public interfaces

5.1.3 Public interfaces that should not be called directly

5.1.4 Legacy convenience interfaces

5.1.5 Deprecated public interfaces

5.2 krb5 types and structures

5.2.1 Public

krb5_int32

krb5_int32

krb5_int32 is a signed 32-bit integer type

krb5_ui_4

krb5_ui_4

krb5_ui_4 is an unsigned 32-bit integer type.

5.2.2 Internal

5.3 krb5 simple macros

5.3.1 Public

5.3.2 Deprecated macros

K

R