NAME

nis_objects – NIS+ object formats

SYNOPSIS

cc [*flag* . . .] *file*. . . **-lnsl** [*library*. . .]

/usr/include/rpcsvc/nis_objects.x

DESCRIPTION

Common Attributes

The NIS+ service uses a variant record structure to hold the contents of the objects that are used by the NIS+ service. These objects all share a common structure which defines a set of attributes that all objects possess. The **nis_object** structure contains the following members:

```
typedef char          *nis_name;
struct  nis_object {
        nis_oid       zo_oid;
        nis_name      zo_name;
        nis_name      zo_owner;
        nis_name      zo_group;
        nis_name      zo_domain;
        u_long        zo_access;
        u_long        zo_ttl;
        objdata       zo_data;
};
```

In this structure, the first member **zo_oid**, is a 64 bit number that uniquely identifies this instance of the object on this server. This member is filled in by the server when the object is created and changed by the server when the object is modified. When used in conjunction with the object's name and domain it uniquely identifies the object in the entire NIS+ namespace.

The second member, **zo_name**, contains the leaf name of the object. This name is *never* terminated with a '.' (dot). When an object is created or added to the namespace, the client library will automatically fill in this field and the domain name from the name that was passed to the function.

zo_domain contains the name of the NIS+ domain to which this object belongs. This information is useful when tracking the parentage of an object from a cache. When used in conjunction with the members **zo_name** and **zo_oid**, it uniquely identifies an object. This makes it possible to always reconstruct the name of an object by using the code fragment

```
sprintf(buf,"%s.%s", obj→zo_name, obj→zo_domain);
```

The **zo_owner** and **zo_group** members contain the NIS+ names of the object's principal owner and group owner, respectively. Both names *must be* NIS+ fully qualified names. However, neither name can be used directly to identify the object they represent. This stems from the condition that NIS+ uses itself to store information that it exports.

The **zo_owner** member contains a fully qualified NIS+ name of the form *principal.domain*. This name is called a NIS+ principal name and is used to identify authentication information in a credential table. When the server constructs a search query of the form

[cname=*principal***],cred.org_dir.***domain***.**

The query will return to the server credential information about *principal* for all flavors of RPC authentication that are in use by that principal. When an RPC request is made to the server, the authentication flavor is extracted from the request and is used to find out the NIS+ principal name of the client. For example, if the client is using the AUTH_DES authentication flavor, it will include in the authentication credentials the network name or *netname* of the user making the request. This netname will be of the form

unix.*UID@domain*

The NIS+ server will then construct a query on the credential database of the form

[auth_name=*netname***,auth_type=AUTH_DES],cred.org_dir.***domain***.**

This query will return an entry which contains a principal name in the first column. This NIS+ principal name is used to control access to NIS+ objects.

The group owner for the object is treated differently. The group owner member is optional (it should be the null string if not present) but must be fully qualified if present. A group name takes the form

group.domain.

which the server then maps into a name of the form

*group***.groups_dir.***domain.*

The purpose of this mapping is to prevent NIS+ group names from conflicting with user specified domain or table names. For example, if a domain was called *engineering.foo.com.*, then without the mapping a NIS+ group of the same name to represent members of engineering would not be possible. The contents of groups are lists of NIS+ principal names which are used exactly like the **zo_owner** name in the object. See **nis_groups**(3N) for more details.

The **zo_access** member contains the bitmask of access rights assigned to this object. There are four access rights defined, and four are reserved for future use and must be zero. This group of 8 access rights can be granted to four categories of client. These categories are the object's owner, the object's group owner, all authenticated clients (world), and all unauthenticated clients (nobody). Note that access granted to ''nobody'' is really access granted to everyone, authenticated and unauthenticated clients.

The **zo_ttl** member contains the number of seconds that the object can ''live'' in a cache before it is expired. This value is called the time to live for this object. This number is particularly important on group and directory (domain) objects. When an object is cached, the current time is added to the value in **zo_ttl**. Then each time the cached object is used, the time in **zo_ttl** is compared with the current time. If the current time is later than the time in **zo_ttl** the object is said to have expired and the cached copy should not be used.

Setting the TTL is somewhat of an art. You can think of it as the ''half life'' of the object, or half the amount of time you believe will pass before the object changes. The benefit of setting the ttl to a large number is that the object will stay in a cache for long periods of time. The problem with setting it to a large value is that when the object changes it will take a long time for the caches to flush out old copies of that object. The problems and benefits are reversed for setting the time to a small value. Generally setting the value to 43200 (12 hrs) is reasonable for things that change day to day, and 3024000 is good for things that change week to week. Setting the value to 0 will prevent the object from ever being cached since it would expire immediately.

The **zo_data** member is a discriminated union with the following members:

```
zotypes zo_type;
union {
        struct directory_obj    di_data;
        struct group_obj        gr_data;
        struct table_obj        ta_data;
        struct entry_obj        en_data;
        struct link_obj         li_data;
        struct {
                u_int           po_data_len;
                char            *po_data_val;
        } po_data;
} objdata_u;
```

The union is discriminated based on the type value contained in **zo_type**. There six types of objects currently defined in the NIS+ service. These types are the directory, link, group, table, entry, and private types.

```
enum zotypes {
        BOGUS_OBJ       = 0,
        NO_OBJ          = 1,
        DIRECTORY_OBJ   = 2,
        GROUP_OBJ       = 3,
        TABLE_OBJ       = 4,
        ENTRY_OBJ       = 5,
        LINK_OBJ        = 6,
        PRIVATE_OBJ     = 7
};
typedef enum zotypes zotypes;
```

All object types define a structure that contains data specific to that type of object. The simplest are private objects which are defined to contain a variable length array of octets. Only the owner of the object is expected to understand the contents of a private object. The following section describe the other five object types in more significant detail.

Directory Objects

The first type of object is the *directory* object. This object's variant part is defined as follows:

```
enum nstype {
        UNKNOWN     = 0,
        NIS         = 1,
        SUNYP       = 2,
        DNS         = 4,
        X500        = 5,
        DNANS       = 6,
        XCHS        = 7,
}
typedef enum nstype nstype;

struct oar_mask {
                u_long      oa_rights;
                zotypes     oa_otype;
}
typedef struct oar_mask oar_mask;

struct endpoint {
                char        *uaddr;
                char        *family;
                char        *proto;
}
typedef struct endpoint endpoint;

struct nis_server {
                nis_name    name;
                struct {
                            u_int           ep_len;
                            endpoint        *ep_val;
                } ep;
                u_long      key_type;
                netobj      pkey;
}
typedef struct nis_server nis_server;

struct directory_obj {
                nis_name    do_name;
                nstype      do_type;
                struct {
                            u_int           do_servers_len;
                            nis_server      *do_servers_val;
                } do_servers;
                u_long      do_ttl;
                struct {
```

```
                u_int       do_armask_len;
                oar_mask    *do_armask_val;
        } do_armask;
}
typedef struct directory_obj directory_obj;
```

The main structure contains five primary members: **do_name**, **do_type**, **do_servers**, **do_ttl**, and **do_armask**. The information in the **do_servers** structure is sufficient for the client library to create a network connection with the named server for the directory.

The **do_name** member contains the name of the directory or domain represented in a format that is understandable by the type of nameservice serving that domain. In the case of NIS+ domains, this is the same as the name that can be composed using the **zo_name** and **zo_domain** members. For other name services, this name will be a name that they understand. For example, if this were a directory object describing an X.500 namespace that is ''under'' the NIS+ directory *eng.sun.com.*, this name might contain ''/C=US, /O=Sun Microsystems, /OU=Engineering/''. The type of nameservice that is being described is determined by the value of the member **do_type**.

The **do_servers** structure contains two members. **do_servers_val** is an array of *nis_server* structures; **do_servers_len** is the number of cells in the array. The *nis_server* structure is designed to contain enough information such that machines on the network providing name services can be contacted without having to use a name service. In the case of NIS+ servers, this information is the name of the machine in *name*, its public key for authentication in *pkey*, and a variable length array of endpoints, each of which describes the network endpoint for the **rpcbind** daemon on the named machine. The client library uses the addresses to contact the server using a transport that both the client and server can communicate on and then queries the **rpcbind** daemon to get the actual transport address that the server is using.

Note that the first server in the *do_servers* list is always the master server for the directory.

The *key_type* field describes the type of key stored in the *pkey* netobj (see **/usr/include/rpc/xdr.h** for a definition of the network object structure). Currently supported types are **NIS_PK_NONE** for no public key and **NIS_PK_DH** for a Diffie-Hellman type public key.

The **do_ttl** member contains a copy of the **zo_ttl** member from the common attributes. This is the duplicated because the cache manager only caches the variant part of the directory object.

The **do_armask** structure contains two members. **do_armask_val** is an array of **oar_mask** structures; **do_armask_len** is the number of cells in the array. The **oar_mask** structure contains two members: **oa_rights** specifies the access rights allowed for objects of type **oa_otype**. These access rights are used for objects of the given type in the directory when they are present in this array.

The granting of access rights for objects contained within a directory is actually two-tiered. If the directory object itself grants a given access right (using the **zo_access** member in the **nis_object** structure representing the directory), then all objects within the directory are allowed that access. Otherwise, the **do_armask** structure is examined to see if the access is allowed specifically for that type of structure. This allows the administrator of a namespace to set separate policies for different object types, for example, one policy for the creation of tables and another policy for the creation of other directories. See **nis+**(1) for more details.

Link Objects

Link objects provide a means of providing *aliases* or symbolic links within the namespace. Their variant part is defined as follows.

```
struct link_obj {
        zotypes     li_rtype;
        struct {
                u_int       li_attrs_len;
                nis_attr    *li_attrs_val;
        } li_attrs;
        nis_name li_name;
}
```

The **li_rtype** member contains the object type of the object pointed to by the link. This is only a hint, since the object which the link points to may have changed or been removed. The fully qualified name of the object (table or otherwise) is specified in the member **li_name**.

NIS+ links can point to either other objects within the NIS+ namespace, or to entries within a NIS+ table. If the object pointed to by the link is a table and the member **li_attrs** has a nonzero number of attributes (index name/value pairs) specified, the table is searched when this link is followed. All entries which match the specified search pattern are returned. Note, that unless the flag **FOLLOW_LINKS** is specified, the **nis_lookup**(3N) function will always return non-entry objects.

Group Objects

Group objects contain a membership list of NIS+ principals. The group objects' variant part is defined as follows.

```
struct group_obj {
        u_long      gr_flags;
        struct {
                u_int       gr_members_len;
                nis_name    *gr_members_val;
        } gr_members;
}
```

The **gr_flags** member contains flags that are currently unused. The **gr_members** structure contains the list of principals. For a complete description of how group objects are manipulated see **nis_groups**(3N).

Table Objects

The NIS+ table object is analogous to a YP map. The differences stem from the access controls, and the variable schemas that NIS+ allows. The table objects data structure is defined as follows:

```
#define TA_BINARY        1
#define TA_CRYPT         2
#define TA_XDR           4
#define TA_SEARCHABLE    8
#define TA_CASE          16

struct table_col {
        char      *tc_name;
        u_long    tc_flags;
        u_long    tc_rights;
}
typedef struct table_col table_col;

struct table_obj {
        char      *ta_type;
        u_int     ta_maxcol;
        u_char    ta_sep;
        struct {
                  u_int       ta_cols_len;
                  table_col   *ta_cols_val;
        } ta_cols;
        char      *ta_path;
}
```

The **ta_type** member contains a string that identifies the type of entries in this table. NIS+ does not enforce any policies as to the contents of this string. However, when entries are added to the table, the NIS+ service will check to see that they have the same ''type'' as the table as specified by this member.

The structure **ta_cols** contains two members. **ta_cols_val** is an array of **table_col** structures. The length of the array depends on the number of columns in the table; it is defined when the table is created and is stored in **ta_cols_len**. **ta_maxcol** also contains the number of columns in the table and always has the same value as **ta_cols_len**. Once the table is created, this length field cannot be changed.

The **ta_sep** character is used by client applications that wish to print out an entry from the table. Typically this is either space ('' '') or colon ('':'').

The **ta_path** string defines a concatenation path for tables. This string contains an ordered list of fully qualified table names, separated by colons, that are to be searched if a search on this table fails to match any entries. This path is only used with the flag **FOLLOW_PATH** with a **nis_list( )** call. See **nis_tables**(3N) for information on these flags.

In addition to checking the type, the service will check that the number of columns in an entry is the same as those in the table before allowing that entry to be added.

Each column has associated with it a name in **tc_name**, a set of flags in **tc_flags**, and a set of access rights in **tc_rights**. The name should be indicative of the contents of that column.

The **TA_BINARY** flag indicates that data in the column is binary (rather than text). Columns that are searchable cannot contain binary data. The **TA_CRYPT** flag specifies that the information in this column should be encrypted prior to sending it over the network. This flag has no effect in the export version of NIS+. The **TA_XDR** flag is used to tell the client application that the data in this column is encoded using the XDR protocol. The **TA_BINARY** flag must be specified with the XDR flag. Further, by convention, the name of a column that has the **TA_XDR** flag set is the name of the XDR function that will decode the data in that column.

The **TA_SEARCHABLE** flag specifies that values in this column can be searched. Searchable columns must contain textual data and must have a name associated with them. The flag **TA_CASE** specifies that searches involving this column ignore the case of the value in the column. At least one of the columns in the table should be searchable. Also, the combination of all searchable column values should uniquely select an entry within the table.

Entry Objects

Entry objects are stored in tables. The structure used to define the entry data is as follows.

```
#define EN_BINARY       1
#define EN_CRYPT        2
#define EN_XDR          4
#define EN_MODIFIED     8

struct entry_col {
        u_long    ec_flags;
        struct {
                  u_int       ec_value_len;
                  char        *ec_value_val;
        } ec_value;
}
typedef struct entry_col entry_col;

struct entry_obj {
        char      *en_type;
        struct {
                  u_int       en_cols_len;
                  entry_col   *en_cols_val;
        } en_cols;
}
```

The **en_type** member contains a string that specifies the type of data this entry represents. The NIS+ server will compare this string to the type string specified in the table object and disallow any updates or modifications if they differ.

The **en_cols** structure contains two members: **en_cols_len** and **en_cols_val**. **en_cols_val** is an array of **entry_col** structures. **en_cols_len** contains a count of the number of cells in the **en_cols_val** array and reflects the number of columns in the table -- it always contains the same value as the **table_obj.ta_cols.ta_cols_len** member from the table which contains the entry.

The **entry_col** structure contains information about the entry's per-column values. **ec_value** contains information about a particular value. It has two members: **ec_value_val**, which is the value itself, and **ec_value_len**, which is the length (in bytes) of the value. **entry_col** also contains the member **ec_flags**, which contains a set of flags for the entry.

The flags in **ec_flags** are primarily used when adding or modifying entries in a table. All columns that have the flag **EN_CRYPT** set will be encrypted prior to sending them over the network. Columns with **EN_BINARY** set are presumed to contain binary data. The server will ensure that the column in the table object specifies binary data prior to allowing the entry to be added. When modifying entries in a table, only those columns that have changed need be sent to the server. Those columns should each have the **EN_MODIFIED** flag set to indicate this to the server.

SEE ALSO

nis+(1), **nis_groups**(3N), **nis_names**(3N), **nis_server**(3N), **nis_subr**(3N), **nis_tables**(3N)