Document Status#

Note that this document is mostly a stream-of-consciousness repository for the developers thoughts. If anybody else has to read it... well... my apologies.

Requirements#

POSIX access controls have the notion of groups. A user belongs to at least one group and may belong to multiple supplementary groups. Each resource belongs to one group and has a set of permissions that define the access available to a member of the group. The existing Neutrino implementation restricts the number of groups to which a user may belong to 8. Most other Unix/Linux/POSIX systems allow at least 16 supplementary groups for each process.

We need to relieve this restriction. We want the API to allow for an unlimited number of gids. Internal restrictions on the maximum number are probably acceptable, as long as the internal restrictions are very large.

Development Plan#

To satisfy the requirements, we will make the changes described in the design section below. These changes will be rolled out as follows:

- initial changes will be made to lib/c and services/system to implement the new APIs and change internal implementations to make use of them.
- these initial changes will be code inspected and checked in.
- any code within (or not too far out of) the control of the kernel group will be adapted to use the new APIs. These changes will be inspected and checked in individually or in small groups.

Once the initial changes are made to lib/c and services/system, applications that use the old APIs will continue to function, but they will only function completely correctly when each user has fewer than 8 supplemental groups. Many such applications are outside the control of the kernel group. PRs will be generated for any such code owned by QNX. Release notes will be added to explain the changes necessary for customer code.

Current Implementation#

Public Type Definitions#

Current release has:

```
<limits.h>
#define __NGROUPS_MAX 8
#define NGROUPS_MAX __NGROUPS_MAX

<sys/neutrino.h>
struct _cred_info {
    ...
    _Uint32t    ngroups;
    gid_t       grouplist[__NGROUPS_MAX];
}
struct _client_info {
    struct _cred_info cred;
}
```

Users of _cred_info and _client_info#

There are many users of the _cred_info and _client_info structures. As these types currently depend on the NGROUPS_MAX value, users of these types must be examined. Fortunately many do not pay any attention to the grouplist field within the _cred_info structure; changes to the definition of grouplist will be irrelevant for these.

Changes to the _cred_info structure are important to the implementation and users of ConnectClientInfo() and iofunc_client_info(). These routines will be discussed below.

The following code outside of services/system and lib/c uses the _cred_info structure:

- The _proc_getsetid_reply message reply structure includes a _cred_info structure. However, none of the existing users of this message depend on the grouplist field, so we may change the code that returns this data (the handler for the _PROC_ID_GETID message) as required.
- The kerargs_channel_create structure includes a _cred_info structure. The normal ChannelCreate function does not use this field of the kerargs structure: it is only used by the ChannelCreateExt kernel call. This call is used by asyncmsg_channel_create() which passes a null pointer for the cred field, and it is used for the create_mq() call, where a real _cred_info structure is used. The create_mq() call is only used by resmgr_mq_open(), which gets its _cred_info structure from iofunc_client_info().
- The kerargs_net_signal_kill structure includes a _cred_info structure, however processing of this message type ignores the grouplist field.
- nfs2/vc.c and nfs.c copies _cred_info structures using memcpy(dst,src,sizeof(struct _cred_info)). Also, nfs2/struct.h defines two structures (_nfs_commit and _nfs_vc) that embed _cred_info structures. Same for nfs3 vc.c, nfs.c and struct.h.
- io-pkt embeds _cred_info structures in its _io_net_dcmd_ret_cred structure, which is used in a large variety of places.
- io-net uses _cred_info in a manner similar to io-pkt.
- utils/n/netsniff utility clones the _cred_info structure and embeds the clone in a variety of other structures.
- utils/t/traceprinter regression code defines _cred_info structures.

The _client_info structure (which embeds a _cred_info structure) is used in a very large number of locations.

Fortunately, the very large majority of the users of _cred_info and _client_info structures pay no attention to the grouplist field. As such, if the grouplist is not complete, it is irrelevant.

ConnectClientInfo(scoid, client_info, ngroups)#

ConnectClientInfo() and ConnectClientInfo_r() are used to populate a _client_info structure. These are implemented with a kernel call (which invoked ker_connect_client_info()) that does not assume the size of the grouplist array in the _cred_info structure.

The calling code can pass in an ngroups value of zero, in which case the kernel call returns the actual number of groups in client_info->cred.ngroups but the client_info->cred.grouplist field is not populated. If the calling code passes in a non-zero value for ngroups, then when ConnectClientInfo() returns, client_info->cred.grouplist will be populated with as many groups as will fit (up to ngroups maximum) and client_info->cred.ngroups will equal the number of groups copied in.

In most locations where ConnectClientInfo() is used, the ngroups parameter is zero, meaning that the calling code is not retrieving the group list. However there are some places where ConnectClientInfo() is called with a non-zero ngroups parameter. Even in those places, the grouplist field might be ignored.

We'll need to touch (or at least look more closely at) the following users of the grouplist field and ConnectClientInfo()/ConnectClientInfo_r():

- lib/c/1/getgroups.c

- lib/c/1/access.c
- lib/c/iofunc/iofunc_client_info.c
- services/system/procmgr/procmgr_spawn.c
- services/system/pathmgr/devmem.c
- services/system/pathmgr/procfs.c
- lib/io-pkt/sys/kern/uipc_usrreq.c
- services/net/npm/tcpip-1-5/kern/uipc_usrreq.c
- lib/io-pkt/sys/msg.c
- utils/t/traceprinter/regress/bk[2]_kercalls.c

Note that the following invoke ConnectClientInfo() with a non-zero ngroups parameter, but then ignore the group list:

- bcm5690_res.c (two separate instances)
- services/gns/gns_msg.c
- services/gns/gsn_res.c
- hardware/support/IXF1104/ptlone/ixf1104ce_qnx_res.c

iofunc_client_info()#

Many resmgrs call iofunc_client_info() and then use the resulting _client_info structure to invoke other routines. These functions use a fixed-size _client_info structure to hold the set of groups, and don't provide a mechanism for expanding the structure size.

Problem routines (those that depend on the group information) are iofunc_chown(), iofunc_check_access(), create_mq(). The _client_info structure is passed on to many other routines, including (for example) iofunc_open() and iofunc_create which store use the provided _client_info structure for calls to iofunc_check_access(). Because of the intertwined nature of these calls, it is best to assume that any code which invokes iofunc_client_info depends on having an accurate grouplist field.

Note that some code invokes iofunc_check_access() or iofunc_chown() with _client_info data obtained from ConnectClientInfo() directly rather than from iofunc_client_info().

There are a large number of calls to these routines in our own code, and many more will exist in customer-written resource managers. As much of the code is in customer hands, it is not possible to change the users of these functions. We must maintain a large degree of backwards compatibility.

iofunc_attr_init()#

The iofunc_attr_init() routine takes a struct _client_info parameter. Some locations (e.g. services/net/npm/tcpip-1-5/kern/uipc_usrreq.c) invoke ConnectClientInfo() or iofunc_client_info() to retrieve the _client_info data using a non-zero ngroups parameter. However, iofunc_attr_init() does not use the _cred_info field of the _client_info structure, so having incorrect/imcomplete gid information will not cause an immediate problem. It is a bit fragile, however.

getgroups(), getgrouplist()#

The getgroups() function is a POSIX function used to retrieve the list of groups associated with the current process. It accepts as parameters a chunk of memory into which the group ids will be written, and the size of this chunk of memory. If the size parameter is zero, the function will return the number of groups. If the size parameter is non-zero and greater than or equal to the actual number of groups, the function will write the groups to the buffer and return the number of groups. If the size parameter is non-zero but less than the actual number of groups, the function will return an error.

Correct usage of getgroups() is to invoke it first with a zero size, allocate a sufficient buffer, then allocate it again with the size of the buffer. However, since Neutrino has always been restricted to NGROUPS_MAX groups, some code only invokes getgroups() once with a table of size NGROUPS_MAX. This code will break in any system that supports more than NGROUPS_MAX supplementary groups.

The getgrouplist() function is similar. Given a user name, it returns the list of that user's supplementary groups. It is more friendly, however, in that it always indicates the actual number of groups to which the user belongs, even if the provided table isn't large enough to hold them all (in this case it updates the list size parameter and returns an error). However, this leads to a potential programming error – if the calling code always assumes that the provided table is large enough and doesn't check the return code, it will not work correctly.

Most users of these functions are well behaved. The exceptions are:

- lib/io-pkt/crypto/external/bsd/openssh/dist/uidswap.c – couple of places where it uses getgroups() with fixed-size arrays
- utils/n/newgrp/nto/newgrp.c – invokes getgroups() with a fixed-size group array.
- lib/rpc/qnx4/auth_unix.c and lib/rpc/nto/auth_unix.c – uses their own fixed-size arrays for group list – 16 elements
- utils/i/id/id.c
- services/pppd/main.c
- lib/io-pkt/services/pppd/main.c
- ports/rxtx/src/SerialImp.c
- services/openssh/groupaccess.c & lib/io-pkt/crypto/external/bsd/openssh/dist/groupaccess.c

Most of these call getgroups() and use a fixed-size (NGROUPS_MAX) array of gids. These calls will fail with EINVAL if the process has more than 8 groups.

NGROUPS_MAX#

NGROUPS_MAX is a compile-time constant that is related to the maximum number of supplemental groups that a user can have. However, it is an obsolete concept, replaced in modern environments with the sysconf(_SC_NGROUPS_MAX) value.

According to POSIX, NGROUPS_MAX is the minimum value that might be returned by sysconf(_SC_NGROUPS_MAX). That makes it rather useless... It is never safe to use it as an array size. Safe code must always retrieve either the actual number of groups or the _SC_NGROUPS_MAX number first, and then retrieve the list of groups.

But, existing code (both our own and undoubtedly user code) does use NGROUPS_MAX as an array boundary. This has been safe for users of Neutrino where NGROUPS_MAX and sysconf(_SC_NGROUPS_MAX) were equivalent. However, by modern POSIX standards, any code that depends on NGROUPS_MAX being greater than or equal to the maximum number of supplemental groups is broken.

lib/c Implementation#

getgroups()#

Uses ConnectClientInfo() with hardcoded NGROUPS_MAX to retrieve the list of groups from procnto. This implementation will need to be changed to support a larger number of groups.

getgrouplist()/initgroups()/setgroups()#

The getgrouplist() call uses the lib/c/1/passwd.c mechanisms to read the /etc/groups file. It already handles an arbitrarily large number of groups without change. It is used in process initialization to define the groups that the process belongs to, through the initgroups()/setgroups() functions -- initgroups() is invoked by system utilities such as login when it is necessary to establish the context of a new process, and initgroups() uses getgrouplist() and setgroups() to set up the context.

While getgrouplist() can handle an arbitrarily sized list of groups, initgroups() cannot -- it uses a hard-coded constant of size NGROUPS_MAX to hold the values returned from getgrouplist(). This implementation will need to be changed.

procnto Implementation#

Type Definitions

```
<kernel/objects.h>
struct credential_entry {
    ...
    struct _cred_info   info;
}
typedef struct credential_entry CREDENTIAL;
struct process_entry {
    ...
    CREDENTIAL *cred;
}

struct channel_gbl_entry {
    ...
    struct _cred_info cred;
}
```

Internal Details#

Process group lists are held in prp->cred->info.grouplist.

Note that prp->cred is a separate object from the process entry, allocated from its own souls list. The _cred_info soul allocation and deallocation is done from a single routine: nano_cred.c::cred_set().

Global channel group lists are held in chp->cred.grouplist.

In this case, the _cred_info structure is embedded in the channel_gbl_entry structure.

Both of these definitions are internal to services/system, so may be changed as required.

The following procnto code refers to the grouplist field of a _cred_info structure.

- services/system/ker/nano_cred.c
- services/system/ker/ker_net.c
- services/system/procmgr/procmgr_getsetid.c
- services/system/apmgr/apmgr_support.c
- services/system/ker/ker_connect.c

Design#

Public APIs#

The _cred_info and _client_info structure definitions are part of the public API. The fact that they define fixed-sized arrays of groups ids is a problem.

There are two public functional interfaces that we need to consider: the ConnectClientInfo() and ConnectClientInfo_r routines which returns a struct _client_info (and the related iofunc_client_info and iofunc_check_access calls), the getgroups()/getgrouplist() routines.

Finally, there's the NGROUPS_MAX constant which we must take a look at.

Type Definitions#

For simple backwards compatibility reasons, the _cred_info and _client_info structures will be left unchanged.

In usage, however, it is expected that the size of these structures will vary with the number of groups held in the _cred_info grouplist field -- even though it is declared to be fixed-size with 8 elements, it is treated as variable-sized. Macros will be introduced to calculate the size of the structures, and to allow local or static variables to be defined to support a given number of elements.

```
#define _CRED_INFO_SIZE(_groupcount)        \
          (offsetof(struct _cred_info, grouplist)     \
        + ((_groupcount)*sizeof(gid_t)))
#define _CLIENT_INFO_SIZE(_groupcount)        \
        (offsetof(struct _client_info, cred.grouplist) \
        + ((_groupcount)*sizeof(gid_t)))
#define _DECL_CRED_INFO(_name,_groupcount)                     \
    unsigned       __buf_##_name[_CRED_INFO_SIZE((_groupcount))/sizeof(unsigned)]; \
    struct _cred_info   *_name = (struct _cred_info*)&__buf_##_name[0];
#define _DECL_CLIENT_INFO(_name,_groupcount)                     \
    unsigned       __buf_##_name[_CLIENT_INFO_SIZE((_groupcount))/sizeof(unsigned)];   \
    struct _client_info *_name = (struct _client_info*)&__buf_##_name[0];
```

...

```
// example code
void foo(struct _cred_info *ci_p) {
    _DECL_CRED_INFO(local_ci_p, ci_p->ngroups);
    memcpy(local_ci_p, ci_p, _CRED_INFO_SIZE(ci_p->ngroups));
    ...
}
```

It is acceptable to declare a variable of type struct _cred_info or type struct _client_info, as long as the code is careful to never attempt to fully populate the groups field. For example, in several places existing code defines a local of type struct _client_info and then populates it with a call to ConnectClientInfo specifying a zero grouplist size (the code is only interested in the uids); such code is acceptable and could be left unchanged. Alternately, to save a small bit of stack space, the code could be changed to use the _DECL_CLIENT_INFO macro.

For example, services/system/proc/support.c includes the following definition of proc_isaccess():

```
int
proc_isaccess(PROCESS *prp, struct _msg_info *rcvinfo) {
    struct _client_info              info;

    return ConnectClientInfo(rcvinfo->scoid, &info, 0) == -1 ? 0 :
        (info.cred.euid == 0 || (prp && prp->cred->info.euid == info.cred.euid));
}
```

It could be left as-is, or could be changed to the version given below. This version uses 28 bytes less stack space.

```
int
```

```
proc_isaccess(PROCESS *prp, struct _msg_info *rcvinfo) {
   _DECL_CLIENT_INFO( info_p, 0);

   return ConnectClientInfo(rcvinfo->scoid, info_p, 0) == -1 ? 0 :
      (info_p->cred.euid == 0 || (prp && prp->cred->info.euid == info_p->cred.euid));
}
```

ConnectClientInfo() and Related#

We must support current applications' usage of ConnectClientInfo()/iofunc_client_info()/iofunc_check_access(). For complete support there does not seem to be an alternative to touching each place that these functions are used.

Investigation of the symbol 'grouplist' indicates that there are very few places that reference it directly. Almost all places that use a _client_info structure ignore the grouplist, excepting that they pass the structure on to iofunc_check_access() or similar functions.

One option would be to modify the semantics of the ConnectClientInfo() kernel call so that on return the client_info->cred.ngroups field is always equal to the number of actual groups, rather than the number stored in the table. Then any function that references the grouplist field would need to check the ngroups field against the bounds of the grouplist field. If (ngroups > bounds of grouplist) the code would need to make another kernel call to get the complete grouplist.

This change might allow us to avoid changing existing applications, however it would break backwards compatibility. An application compiled and run against 6.5.0 libc would expect that ngroups field to be the actual number of groups copied out of the kernel, and iofunc_check_access() (or equivalent) would overrun the array bounds on examining the grouplist.

Instead we'll introduce a new parallel API for ConnectClientInfo() (we'll call it ConnectClientInfoExt()) that will allocate a _client_info structure with a sufficiently large amount of memory to hold the entire group list. We will modify existing code that invokes ConnectClientInfo() with a non-zero ngroups parameter to invoke the new API and to free the allocated resources when they are no longer needed (using a new ClientInfoExtFree() routine). We'll also implement iofunc_client_info_ext() which will make use of ConnectClientInfoExt() (along with the necessary iofunc_client_info_ext_free()).

The new ConnectClientInfoExt() call will use an initial call to ConnectClientInfo() to retrieve a large number of groups. If it turns out that this large number is insufficient, a second call will be made to retrieve the grouplist size, and a third call will be made to retrieve the entire group list. Once the routine knows the size of the group list it will allocate memory from the process heap (using malloc) to hold the entire _client_info structure. This allocated memory will be returned to the calling code, and must be freed later using ClientInfoExtFree() (which does a simple free() call). The ConnectClientInfoExt() will also differ from the ConnectClientInfo() call in that it will set a new flag in the _client_info.flags field to indicate that the group list is complete.

There are a large number of users of iofunc_client_access(). To assist with backwards compatibility, we will modify that routine so that if necessary it can retrieve additional group information from the kernel. In this manner, existing applications will not need to be changed to work with more than 8 groups. However, this will come at a slight cost in efficiency. The iofunc_client_access() routine will be modified so that if the _client_info structure passed in contains exactly 8 groups and the _client_info.flags don't indicate that the group list is complete, the group list will not be trusted and an additional call will be made to iofunc_client_info_ext() to retrieve the full group list -- this requires an additional kernel call (or, in extreme cases, an additional 3 kernel calls). Further, iofunc_client_access() cannot differentiate between a client with exactly 8 groups, and one that has more than 8 groups. In the case of a client with exactly 8 groups, iofunc_client_access() cannot trust that the group list is accurate, so the extra overhead will be imposed unnecessarily.

Regardless, these changes will allow most existing applications to work in the new environment without modification. This backwards compatibility is worth a little extra overhead in the applications that have not been updated.

Most existing customer applications will work seamlessly if they make use of the new libc routines. Applications that refer directly to the group list in the client information structure will need to be modified to work correctly in environments with more than 8 groups. This design results in the following compatibility matrix.

environment	vanilla application	application refers to _client_info.cred.grouplist
application not recompiled, statically linked or using pre-6.6 libc	8	8
application using 6.6 libc	X	8
application updated, using 6.6 libc	X	X

where an entry of '8' indicates that the application will work correctly if all users have 8 or fewer groups, and an extry of 'X' indicates that the application will work correctly with any number of groups.

This behaviour will have to be well documented, and a release note raised to get customer applications modified to use the new ConnectClientInfoExt()/iofunc_client_info_ext() routines.

getgroups()/getgrouplist()#

While the implementations of these routines are correct, there are a number of places where these routines are used with a fixed-size (NGROUPS_MAX) array. Each of these places will need to be visited and modified.

In addition, we must create a release note to the effect that existing customer code that uses getgroups() or getgrouplist() with a fixed-size array might need to be revisited.

NGROUPS_MAX#

We cannot simply change the value of NGROUPS_MAX without care, as the value of NGROUPS_MAX defines the size of the _client_info structure, and changing the size of the _client_info structure would break backwards compatibility.

We can change NGROUPS_MAX if we also modify the definition of _cred_info to hard-code the size of the grouplist array at 8 rather than as NGROUPS_MAX. However, there is a large quantity of code that uses the NGROUPS_MAX constant. Much of this code concerns networking, and changing NGROUPS_MAX has great potential to cause breakage.

We will regardless proceed with a change to NGROUPS_MAX. We will modify the definition of _cred_info to remove the dependency on NGROUPS_MAX; we will modify the value of NGROUPS_MAX; and we will do some preliminary testing to determine if there are any obvious sanity issues. As this will be done early in the 6.6 release cycle, there will be plenty of time to deal with any issues that become apparent.

If we are going to change the value of NGROUPS_MAX, the question becomes: what value should it be given? The true value is MAX_INT (the gid_t type is aliased with _GID_T, which is defined as _INT32). However, if we define it to be such a large value, any application that currently uses NGROUPS_MAX to define an array boundary will break ungracefully if it is recompiled. Instead, we will use a value of 65536 -- this is the maximum value of a 16-bit unsigned integer, which is the common type for a gid_t in *NIX systems, is large enough to be effectively infinite and yet small enough that a declaration of an array of gid_t with NGROUPS_MAX elements won't seriously break most code (though it will be wasteful).

We will document the fact that NGROUPS_MAX should not be used, and we will create a release note to ensure that customers are directed to revisit any code that uses NGROUPS_MAX.

Internal kernel/procnto Implementation#

Credentials Storage#

The existing implementation embeds a _cred_info structure in a CREDENTIAL object. Such objects must be fixed size (this is a characteristic of the object allocator). Using a fixed size will impose a limit on the number of groups that can be held in the CREDENTIAL object. Any reasonably large upper limit will result in excessive wasted store for the majority of CREDENTIAL objects.

We could change the definition of the CREDENTIAL object so that instead of embedding the group list it contained a pointer to it. Then when a CREDENTIAL object is allocated, we would need to allocate a corresponding variable-sized piece of data from the heap. If we're going to allocate from the heap in any case, we might as well allocate the whole CREDENTIAL structure from the heap. As such, we'll modify procnto to eliminate the CREDENTIAL souls list, and modify the locations where CREDENTIAL objects are allocated and freed. As all allocations and deallocations are done through a single routine (nano_cred.c::cred_set()), this is a simple change.

When a CREDENTIAL structure is allocated, we always have the complete group list available. We will use _smalloc() and allocate sufficient memory to hold the CREDENTIAL structure with the complete group list. When a CREDENTIAL structure is freed, we will free it with _sfree() (the size for _smalloc/_sfree are calculated based on the ngroups field, which is stored in the structure).

Once the CREDENTIAL structure accounts for more than 8 groups, all of the code which references the group list will work correctly without modification. Code locations where CREDENTIAL structures are copied or passed around must be visited to ensure that entire structure is copied. These locations are described in the next section.

Credentials Interface#

The CREDENTIAL structures stored with each process serve to hold the _cred_info structure. Those places that refer directly to the _cred_info structure through the pointer to the prp->cred point are limited to looking at the various uid fields, and so are of no concern. Some locations set new credentials using the cred_set() function -- these locations are simply modified to ensure the CREDENTIAL structures are large enough to hold the required number of groups.

All code that refers to the ngroups or grouplist field access the _cred_info structure through the CredGet() and CredSet() routines. These routines copy the credentials in and out of the kernel through kerext functions. Their implementation and code that uses them must be visited to ensure they properly support an arbitrary number of groups.

In particular, the CredGet() call will be modified so that it accepts an ngroups parameter that indicates how much room for the grouplist is available in the return structure. Like other credential-returning routines, if provided a zero ngroups input, the CredGet() call will return the actual number of groups in the CREDENTIAL structure but won't try to copy any of them out. Thus users of CredGet() can get the entire group list with two calls.

A number of locations modify credentials by reading the existing credentials with CredGet(), modifying them, and then writing back the results with CredSet(). These locations will have extra overhead with the new CredGet() implementation, as they will need to make two kernel calls to retrieve the entire group list. In these cases the group list is not being modified, but it is required as an input to CredSet(). Instead of incurring the extra kernel call overhead of retrieving the full group list with CredGet(), we will modify CredSet() and cred_set() to accept a new parameter that indicates whether or not the exinsting group list should be preserved.

This makes the implementation of cred_set much more complicated, but the extra efficiency is worth the complication.

The CredGet() and CredSet() routines are used internally to implement external interfaces, through the procnto _PROC_GETSETID message. This message type is used in a number of places, but very few are concerned with the group list. The message is handled by the procmgr_getsetid() routine. The message data structure is a clone of the _cred_info data structure excepting that it does not include the grouplist field. The reply data structure embeds a _cred_info structure.

- Only the _PROC_ID_SETGROUPS message subtype (used by lib/c/1/setgroups.c) passes a group list to procnto. The implementation of the setgroups() function is already correct, but the implementation of the message handling by procmgr_getsetid() needs some minor modifications to handle more than 8 groups.
- A number of places retrieve the credential information using the _PROC_ID_GETID message sub-type. None of these pay any attention to the group list, and so if they fail to retrieve the whole group list from the message reply it is irrelevant. Nothing need be done to support this message subtype. Though the message handler will return the complete group list correctly, the calling code will receive the results using a structure big enough to hold 8 groups. Since the calling code will ignore the returned groups, this is correct. (One might think that lib/c/1/getgroups.c would use _PROC_ID_GETID to retrieve the group list, but that routine is implemented using ConnectClientInfo() instead.).
- A number of places set individual uid fields in the credentials structure using different _PROC_ID_SET* message subtypes. In these cases, procmgr_getsetid() reads the existing credentials with CredGet(), replaces the one field being set, and then writes the credentials back with CredSet(). Only the id value being set is read from the message. These code locations will take advantage of the new CredSet() behaviour and will refrain from reading and rewriting the unchanged group information.

Customer Documentation#

We are making two changes to the code that require changes to the documentation.

The first is the change to the NGROUPS_MAX semantics: NGROUPS_MAX can no longer be taken as the maximum number of supplemental groups to which a user can belong. Instead, applications must use sysconf(_SC_NGROUPS_MAX) where the absolute maximum is required. Any existing code that currently depends on NGROUPS_MAX must be revisited. Any documentation that mentions NGROUPS_MAX must be changed. We must both change our product documentation and create a release note for this issue.

The second change that must be documented is that applications and resource managers should move away from the ConnectClientInfo()/iofunc_client_info() routines. Also note that the ConnectClientInfo() documentation makes reference to the NGROUPS_MAX constant -- this reference should be corrected.

If existing code continues to be used without a recompile, it will continue to work as long as the maximum number of supplemental groups does not exceed 8. With a recompile but no changes, it will continue to work as long as the maximum number of supplemental groups does not exceed the new NGROUPS_MAX value, but the new large NGROUPS_MAX value will result in excessive wasteful store requirements.

Testing#

Testing will consider a number of different functional areas. Each of these is approached differently.

Two primary mechanisms will be used for testing of functional areas:

- debug/testing code added in a "#if 0" block to procnto to test proc/kernel internals
- a test application to exercise libc and interfaces.

In addition to unit testing of the functional areas, the following system tests will be run:

- boot and exercise an x86 with full file system and full user/group id setup
- run regression on simple board
- run regression on board with full file system and full user/group id setup
- run regression on board with full file system a full user/group id setup, using 6.5 lib/c

Kernel Credentials Implementation#

Large Credential Lists and APIs: nano_cred.c, kerext_cred.c#

These must be tested from within the kernel or from a procnto thread. Write some test code that can be compiled into procnto and exercised with with a debug message to proc.

Things to test:

- exercise various paths through cred_set():
  - cip==NULL -- i.e. deleting a CREDENTIAL structure
  - crp==NULL -- i.e. creating a CREDENTIAL structure where none existed before
  - preserve_groups==0, various numbers of new groups: 0, 1, 8, 1000
  - preserve_groups==1, various numbers of existing groups: 0, 1, 8, 1000
  - preserve_groups==0, various numbers of new groups, existing CREDENTIAL matches new
  - preserve_groups==1, various numbers of existing groups, existing CREDENTIAL matches new
- CredGet()/CredSet():
  - test with various ngroups values: 0, 1, 8, 1000

procnto API: procmgr_getsetid.c#

Exercise the various message subtypes. These will be tested as part of testing libc interfaces.

Users of Credentials: kerext_process.c, procmgr_fork/spawn/wait#

Need to visit each of these individually and figure out how to exercise them.

- kerext_process: ensure that across process creation the child inherits the parents credentials. A fork() test will be added to the test application to validate the child's credentials.
- procmgr_fork: ensure that the child has access to the scheduler and memory partitions on a fork. The fork() test will be executed in an APS environment.
- procmgr_spawn and procmgr_posix_spawn: same as procmgr_fork. We will add a spawn() test to the test application that validates the child's credentials across both spawn and posix_spawn. This test will be run in an APS environment.

libc#

Direct Grouplist Interfaces: access, getgroups#

Exercise these functions with a test program.

ConnectClientInfoExt#

Write a test program that acts as a resource manager and uses these functions.

iofunc_client_info/iofunc_check_access#

Write a test program that acts as a resource manager and uses these functions.

Users of iofunc_client_info#

Find an existing resource manager that uses the various functions (link, unlink, mknod, chmod, chown, etc). Verify that with the 6.5 libc but a 6.6 procnto it starts to fail with more than 8 groups, but that it works correctly in this case if it uses a 6.6 libc.

Procnto Resource Managers#

devmem.c, namedsem.c, procfs.c

Utilitites#

Need to test utilities that use groups: id

Progress#

Implementation in services/system and lib/c missing:

- apmgr usage of iofunc_client_info
- various qnet embedding credentials in messages
- various async and global messages w/ embedded credentials
- _cred_info embedded in channel_gbl_entry structure
- investigate usage of CREDENTIAL structure in CONNECT.un.net.cred to ensure we account for variable-sized nature

At this point I believe I have sufficient functionality for the first check-in. The remaining work items can be implemented later. Until then, certain functional areas (e.g. async messaging, qnet, apm) will not work correctly if there are more than 8 groups. That's fine -- these areas are not initially critical.

I have proceeded with testing, starting with the kernel implementation and usage of CREDENTIAL structures. Testing is complete (though a little weak with iofunc_ functions -- I should add more test cases) and code is up for code inspection.