NAME

unix - UNIX-domain protocol family

SYNOPSIS

#include <sys/types.h>
#include <sys/un.h>

DESCRIPTION

The UNIX-domain protocol family is a collection of protocols that provides local (on-machine) interprocess communication through the normal socket(2) mechanisms. The UNIX-domain family supports the SOCK_STREAM, SOCK_SEQPACKET, and SOCK_DGRAM socket types and uses file system pathnames for addressing.

ADDRESSING

UNIX-domain addresses are variable-length file system pathnames of at most 104 characters. The include file <*sys/un.h*> defines this address:

```
struct sockaddr_un {
        u_char  sun_len;
        u_char  sun_family;
        char    sun_path[104];
};
```

Binding a name to a UNIX-domain socket with bind(2) causes a socket file to be created in the file system. This file is *not* removed when the socket is closed -- unlink(2) must be used to remove the file.

The length of UNIX-domain address, required by bind(2) and connect(2), can be calculated by the macro **SUN_LEN**() defined in <*sys/un.h*>. The *sun_path* field must be terminated by a NUL character to be used with **SUN_LEN**(), but the terminating NUL is *not* part of the address.

The UNIX-domain protocol family does not support broadcast addressing or any form of "wildcard" matching on incoming messages. All addresses are absolute- or relative-pathnames of other UNIX-domain sockets. Normal file system access-control mechanisms are also applied when referencing pathnames; e.g., the destination of a connect(2) or sendto(2) must be writable.

CONTROL MESSAGES

The UNIX-domain sockets support the communication of UNIX file descriptors and process credentials through the use of the *msg_control* field in the *msg* argument to sendmsg(2) and recvmsg(2). The items to be passed are described using a *struct cmsghdr* that is defined in the include file <*sys/socket.h*>.

To send file descriptors, the type of the message is SCM_RIGHTS, and the data portion of the messages is an array of integers representing the file descriptors to be passed. The number of descriptors being passed is defined by the length field of the message; the length field is the sum of the size of the header plus the size of the array of file descriptors.

The received descriptor is a *duplicate* of the sender's descriptor, as if it were created via dup(fd) or fcntl(fd, F_DUPFD_CLOEXEC, 0) depending on whether MSG_CMSG_CLOEXEC is passed in the recvmsg(2) call. Descriptors that are awaiting delivery, or that are purposely not received, are automatically closed by the system when the destination socket is closed.

Credentials of the sending process can be transmitted explicitly using a control message of type SCM_CREDS with a data portion of type *struct cmsgcred*, defined in <*sys/socket.h*> as follows:

```
struct cmsgcred {
  pid_t     cmcred_pid;                 /* PID of sending process */
  uid_t     cmcred_uid;                 /* real UID of sending process */
  uid_t     cmcred_euid;                /* effective UID of sending process */
  gid_t     cmcred_gid;                 /* real GID of sending process */
  short     cmcred_ngroups;             /* number of groups */
  gid_t     cmcred_groups[CMGROUP_MAX];    /* groups */
};
```

The sender should pass a zeroed buffer which will be filled in by the system.

The group list is truncated to at most CMGROUP_MAX GIDs.

The process ID *cmcred_pid* should not be looked up (such as via the KERN_PROC_PID sysctl) for making security decisions. The sending process could have exited and its process ID already been reused for a new process.

SOCKET OPTIONS

UNIX domain sockets support a number of socket options for the options level SOL_LOCAL, which can be set with setsockopt(2) and tested with getsockopt(2):

LOCAL_CREDS — This option may be enabled on SOCK_DGRAM, SOCK_SEQPACKET, or a SOCK_STREAM socket. This option provides a mechanism for the receiver to receive the credentials of the process calling write(2), send(2), sendto(2) or sendmsg(2) as a recvmsg(2) control message. The *msg_control* field in the *msghdr* structure points to a buffer that contains a *cmsghdr* structure followed

by a variable length *sockcred* structure, defined in <*sys/socket.h*> as follows:

```
struct sockcred {
  uid_t     sc_uid;             /* real user id */
  uid_t     sc_euid;   /* effective user id */
  gid_t     sc_gid;             /* real group id */
  gid_t     sc_egid;   /* effective group id */
  int       sc_ngroups;         /* number of supplemental groups */
  gid_t     sc_groups[1];       /* variable length */
};
```

The current implementation truncates the group list to at most CMGROUP_MAX groups.

The **SOCKCREDSIZE**() macro computes the size of the *sockcred* structure for a specified number of groups. The *cmsghdr* fields have the following values:

```
cmsg_len = CMSG_LEN(SOCKCREDSIZE(ngroups))
cmsg_level = SOL_SOCKET
cmsg_type = SCM_CREDS
```

On SOCK_STREAM and SOCK_SEQPACKET sockets credentials are passed only on the first read from a socket, then the system clears the option on the socket.

This option and the above explicit *struct cmsgcred* both use the same value SCM_CREDS but incompatible control messages. If this option is enabled and the sender attached a SCM_CREDS control message with a *struct cmsgcred*, it will be discarded and a *struct sockcred* will be included.

Many setuid programs will write(2) data at least partially controlled by the invoker, such as error messages. Therefore, a message accompanied by a particular *sc_euid* value should not be trusted as being from that user.

LOCAL_CREDS_PERSISTENT This option is similar to LOCAL_CREDS, except that socket credentials are passed on every read from a SOCK_STREAM or

SOCK_SEQPACKET socket, instead of just the first read. Additionally, the *msg_control* field in the *msghdr* structure points to a buffer that contains a *cmsghdr* structure followed by a variable length *sockcred2* structure, defined in <*sys/socket.h*> as follows:

```
struct sockcred2 {
  int      sc_version;      /* version of this structure */
  pid_t    sc_pid;          /* PID of sending process */
  uid_t    sc_uid;          /* real user id */
  uid_t    sc_euid;   /* effective user id */
  gid_t    sc_gid;          /* real group id */
  gid_t    sc_egid;   /* effective group id */
  int      sc_ngroups;      /* number of supplemental groups */
  gid_t    sc_groups[1];    /* variable length */
};
```

The current version is zero.

The *cmsghdr* fields have the following values:

```
cmsg_len = CMSG_LEN(SOCKCRED2SIZE(ngroups))
cmsg_level = SOL_SOCKET
cmsg_type = SCM_CREDS2
```

The LOCAL_CREDS and LOCAL_CREDS_PERSISTENT options are mutually exclusive.

LOCAL_CONNWAIT

Used with SOCK_STREAM sockets, this option causes the connect(2) function to block until accept(2) has been called on the listening socket.

LOCAL_PEERCRED

Requested via getsockopt(2) on a SOCK_STREAM or SOCK_SEQPACKET socket returns credentials of the remote side. These will arrive in the form of a filled in *xucred* structure, defined in <*sys/ucred.h*> as follows:

```
struct xucred {
  u_int    cr_version;              /* structure layout version */
  uid_t    cr_uid;                  /* effective user id */
  short    cr_ngroups;              /* number of groups */
```

```
  gid_t     cr_groups[XU_NGROUPS];  /* groups */
  pid_t     cr_pid;                 /* process id of the sending process */
};
```

The *cr_version* fields should be checked against XUCRED_VERSION define.

The credentials presented to the server (the listen(2) caller) are those of the client when it called connect(2); the credentials presented to the client (the connect(2) caller) are those of the server when it called listen(2). This mechanism is reliable; there is no way for either party to influence the credentials presented to its peer except by calling the appropriate system call (e.g., connect(2) or listen(2)) under different effective credentials.

To reliably obtain peer credentials on a SOCK_DGRAM socket refer to the LOCAL_CREDS socket option.

BUFFERING

Due to the local nature of the UNIX-domain sockets, they do not implement send buffers. The send(2) and write(2) families of system calls attempt to write data to the receive buffer of the destination socket.

The default buffer sizes for SOCK_STREAM and SOCK_SEQPACKET UNIX-domain sockets can be configured with *net.local.stream* and *net.local.seqpacket* branches of sysctl(3) MIB respectively. Note that setting the send buffer size (sendspace) affects only the maximum write size.

The UNIX-domain sockets of type SOCK_DGRAM are unreliable and always non-blocking for write operations. The default receive buffer can be configured with *net.local.dgram.recvspace*. The maximum allowed datagram size is limited by *net.local.dgram.maxdgram*. A SOCK_DGRAM socket that has been bound with bind(2) can have multiple peers connected at the same time. The modern FreeBSD implementation will allocate *net.local.dgram.recvspace* sized private buffers in the receive buffer of the bound socket for every connected socket, preventing a situation when a single writer can exhaust all of buffer space. Messages coming from unconnected sends using sendto(2) land on the shared buffer of the receiving socket, which has the same size limit. A side effect of the implementation is that it doesn't guarantee that writes from different senders will arrive at the receiver in the same chronological order they were sent. The order is preserved for writes coming through a particular connection.

SEE ALSO

connect(2), dup(2), fcntl(2), getsockopt(2), listen(2), recvmsg(2), sendto(2), setsockopt(2), socket(2), CMSG_DATA(3), intro(4), sysctl(8)

"An Introductory 4.3 BSD Interprocess Communication Tutorial", *PS1*, 7.

"An Advanced 4.3 BSD Interprocess Communication Tutorial", *PS1*, 8.