NAME

knc - kerberized netcat

SYNOPSIS

knc -l [**-n**] [**-d**] [**-a** *bind_address*] [**-f**] [**-c** *num*] *port prog* [args]
knc -il [**-n**] [**-d**] *prog* [args]
knc -lS *path* [**-n**] [**-d**] [**-a** *bind_address*] [**-f**] [**-c** *num*] *port*
knc -ilS *path* [**-n**] [**-d**]

knc [**-d**] [**-n**] *service@host:port*
knc [**-d**] [**-n**] -*N fd service@host*

DESCRIPTION

knc provides an 8-bit clean, mutually authenticated, integrity protected, private (encrypted) tunnel between two endpoints. The same executable provides both client and server functionality.

The server can operate in either "inetd" or standalone mode. In server mode, **knc** either launches *prog* with arguments *args* or connects to a UNIX domain socket (depending on the presence of the **-S** flag).

The options are as follows:

-l listener (server) mode.

-i set "inetd" mode.

-n do not use the resolver for any name look ups (no DNS mode).

-d increment debug level (specify multiple times for increased debugging).

-a *bind_address*
bind to address *bind_address* when in server mode (default is INADDR_ANY).

-S *path* connect to the named Unix domain socket upon accepting a connection rather than launching a program.

-f don’t fork when in server mode (useful for debugging).

-c in server mode, limit the maximum number of child processes to *num*.

-o *opt* allows for the setting of options. Currently, the following options are implemented:

keepalive	enables TCP keepalives.
no-half-close	disable the half close functionality.
noprivacy	disable encryption (but leave integrity protection).
syslog-ident	set the ident of syslog messages instead of the default of argv[0].

-w in server mode, start as an inetd wait service. That is, expect stdin to be a listening socket and process requests on it.

-M *max* in server mode, the maximum number of connexions to process before exiting.

-N *fd* in client mode, do not attempt to connect to a remote host, but instead use the supplied, pre-connected file descriptor *fd*. The usual knc handshake will be performed over this file descriptor.

-P *sprinc* in client mode, specify the Kerberos principal that we will use for the server.

-S *sun_path* in server mode, connect to the UNIX domain socket specified by *sun_path* rather than run a program.

-T *max_time* in server mode, the maximum time to process requests.

When **knc** launches a program, it inserts the principal of the counter-party into the environment variable KNC_CREDS as well as populating other environment variables. (See *ENVIRONMENT AND UNIX DOMAIN SOCKET PROTOCOL*)

The server connects its network side to the stdin and stdout file descriptors of the launched program. Any reads or writes by the launched program are translated into reads and writes to the network side. Likewise, reads and writes on the network side are translated to the local side. End of file conditions (EOF) are similarly translated.

Similarly, the client connects its stdin and stdout file descriptors to its network side, translating reads and writes as above.

ENVIRONMENT AND UNIX DOMAIN SOCKET PROTOCOL

knc has two distinct ways of communicating information to the server-side process. If **knc** is launching an executable, it communicates by populating the environment of the launched program. However, if **knc** is instead connecting to a Unix domain socket, it must transmit the same information over the socket to the server process.

For launched executables, the current environment variables are defined:

KNC_MECH	Either "krb5" if the mechanism used was Kerberos or the hex-encoded form of the mechanism OID. Note: this can be blank if the GSS library doesn't supply the appropriate functions.
KNC_CREDS	The display name of the remote counterparty. This is only set if the mechanism is Kerberos.
KNC_EXPORT_NAME	The hex encoded export name of the remote counterparty.
KNC_REMOTE_IP	The IP address of the **knc** client program. N.B. *NO ENTITLEMENT DECISIONS* should be based on the contents of this variable. Further, it is only the "nearest" client to the server. Remember that various other tunnels (including **knc**) may be between you and the actual user.
KNC_REMOTE_PORT	The source port of the client.
KNC_VERSION	The version of the server. This is not the version of the client as the server does not know this.

When **knc** instead connects to a UNIX domain socket, it uses the following protocol to transmit the information contained in the environment variables:

```
Key_1:Value_1
Key_2:Value_2
...
END
```

These *KEY:VALUE* pairs will be the very first data transmitted across the newly accepted Unix domain socket. Currently defined *KEY*s are precisely the same as the environment variables detailed above, without the KNC_ prefix. (e.g. *CREDS*, *REMOTE_IP*, etc.)

The server application must parse this protocol until the *END\n* indicator is seen. The application is free to ignore any of the *KEY:VALUE* pairs it sees.

Once these have been transmitted, **knc** begins relaying data as normal. No acknowledgement on the part of the server application is required, and further, it is prohibited, as this will be counted as part of the normal data stream.

SECURITY CONSIDERATIONS

Use of **knc** must be carefully considered in order to bring security benefits to your application. In particular, applications launched by **knc** which wish to trust the contents of KNC_CREDS must not allow themselves to be executed by any means other than **knc**. One method of ensuring this is to cause the launched program to be owned and executable only by a special-purpose uid which issues the **knc** command.

DETAILS AND APPLICATION CONSIDERATIONS

A typical **knc** deployment looks like the diagram below:

```
   A      B                    C      D
  -->    -->                  -->    -->
client   knc   ... network ...   knc   server
  <--    <--                  <--    <--
   E      F                    G      H
```

knc makes no assumptions about the protocol running over its connection. In order to appeal to the widest application and protocol audience, **knc** will attempt to mimick the behavior of TCP sockets insofar as it is possible.

Sockets have a property that most other types of file descriptors do not: they can be *half closed* -- meaning closed in only one direction. This is accomplished in the BSD sockets API by calling shutdown(2). **knc** passes EOF indications on to the "opposite" side by way of this call. For example, if the server exits, or closes the socket *[D,H]*, this produces and EOF condition on *G* (but not *C* -- writes to *C* will get EPIPE). This causes the server side **knc** to pass this EOF condition on to *F* by way of shutdown(2). The EOF condition on *F* is now passed to *E* by way of the client **knc** calling shutdown(2). This produces an EOF condition on *E*, which the client application should see and respond to appropriately (perhaps by calling close(2) on *[A,E]*) This close of *[A,E]* produces an EOF in the client side knc on *B*, which in turn calls shutdown(2), producing an EOF on the server side **knc** on *C*. At this point, the server side **knc** knows communication is not possible in either direction and exits. Similarly for the client side **knc**

The astute reader will point out that *[A,E]* is not a socket in the general case, and that shutdown(2) fails on non-sockets. This is why **knc** *actually* invokes an internal routine **shutdown_or_close**() which handles the non-socket case appropriately.

EXAMPLE

A simple loopback test can be performed by invoking the server as:

```
$ KRB5_KTNAME=/etc/krb5.keytab knc -l 12345 /bin/cat
```

Next, invoke the client as:

```
$ knc host@host_on_which_server_is_running 12345
```

SEE ALSO

nc(1), gssapi(3), kerberos(8).