NAME

pcap − Packet Capture library

SYNOPSIS

#include <pcap/pcap.h>

DESCRIPTION

The Packet Capture library provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism. It also supports saving captured packets to a ‘‘savefile’’, and reading packets from a ‘‘savefile’’.

Initializing

pcap_init() initializes the library. It takes an argument giving options; currently, the options are:

PCAP_CHAR_ENC_LOCAL
Treat all strings supplied as arguments, and return all strings to the caller, as being in the local character encoding.

PCAP_CHAR_ENC_UTF_8
Treat all strings supplied as arguments, and return all strings to the caller, as being in UTF-8.

On UNIX-like systems, the local character encoding is assumed to be UTF-8, so no character encoding transformations are done.

On Windows, the local character encoding is the local ANSI code page.

If **pcap_init**() is called, the deprecated **pcap_lookupdev**() routine always fails, so it should not be used, and, on Windows, **pcap_create**() does not attempt to handle UTF-16LE strings.

If **pcap_init**() is not called, strings are treated as being in the local ANSI code page on Windows, **pcap_lookupdev**() will succeed if there is a device on which to capture, and **pcap_create**() makes an attempt to check whether the string passed as an argument is a UTF-16LE string - note that this attempt is unsafe, as it may run past the end of the string - to handle **pcap_lookupdev**() returning a UTF-16LE string. Programs that don’t call **pcap_init**() should, on Windows, call **pcap_wsockinit**() to initialize Winsock; this is not necessary if **pcap_init**() is called, as **pcap_init**() will initialize Winsock itself on Windows.

Routines

pcap_init(3PCAP)
initialize the library

Opening a capture handle for reading

To open a handle for a live capture, given the name of the network or other interface on which the capture should be done, call **pcap_create**(), set the appropriate options on the handle, and then activate it with **pcap_activate**(). If**pcap_acti vate**() fails, the handle should be closed with **pcap_close**().

To obtain a list of devices that can be opened for a live capture, call **pcap_findalldevs**(); to free the list returned by **pcap_findalldevs**(), call **pcap_freealldevs**(). **pcap_lookupdev**() will return the first device on that list that is not a ‘‘loopback‘‘ network interface.

To open a handle for a ‘‘savefile’’ from which to read packets, given the pathname of the ‘‘savefile’’, call **pcap_open_offline**(); to set up a handle for a ‘‘savefile’’, given a **FILE *** referring to a file already opened for reading, call **pcap_fopen_offline**().

In order to get a ‘‘fake’’ **pcap_t** for use in routines that require a **pcap_t** as an argument, such as routines to open a ‘‘savefile’’ for writing and to compile a filter expression, call **pcap_open_dead**().

pcap_create(), **pcap_open_offline**(), **pcap_fopen_offline**(), and **pcap_open_dead**() return a pointer to a **pcap_t**, which is the handle used for reading packets from the capture stream or the ‘‘savefile’’, and for finding out information about the capture stream or ‘‘savefile’’. To close a handle, use **pcap_close**().

The options that can be set on a capture handle include snapshot length

If, when capturing, you capture the entire contents of the packet, that requires more CPU time to copy the packet to your application, more disk and possibly network bandwidth to write the packet data to a file, and more disk space to save the packet. If you don't need the entire contents of the packet - for example, if you are only interested in the TCP headers of packets - you can set the "snapshot length" for the capture to an appropriate value. If the snapshot length is set to*snaplen*, and *snaplen* is less than the size of a packet that is captured, only the first *snaplen* bytes of that packet will be captured and provided as packet data.

A snapshot length of 65535 should be sufficient, on most if not all networks, to capture all the data available from the packet.

The snapshot length is set with **pcap_set_snaplen**().

promiscuous mode

On broadcast LANs such as Ethernet, if the network isn't switched, or if the adapter is connected to a "mirror port" on a switch to which all packets passing through the switch are sent, a network adapter receives all packets on the LAN, including unicast or multicast packets not sent to a network address that the network adapter isn't configured to recognize.

Normally, the adapter will discard those packets; however, many network adapters support "promiscuous mode", which is a mode in which all packets, even if they are not sent to an address that the adapter recognizes, are provided to the host. This is useful for passively capturing traffic between two or more other hosts for analysis.

Note that even if an application does not set promiscuous mode, the adapter could well be in promiscuous mode for some other reason.

For now, this doesn't work on the "any" device; if an argument of "any" or **NULL** is supplied, the setting of promiscuous mode is ignored.

Promiscuous mode is set with **pcap_set_promisc**().

monitor mode

On IEEE 802.11 wireless LANs, even if an adapter is in promiscuous mode, it will supply to the host only frames for the network with which it's associated. It might also supply only data frames, not management or control frames, and might not provide the 802.11 header or radio information pseudo-header for those frames.

In "monitor mode", sometimes also called "rfmon mode" (for "Radio Frequency MONitor"), the adapter will supply all frames that it receives, with 802.11 headers, and might supply a pseudo-header with radio information about the frame as well.

Note that in monitor mode the adapter might disassociate from the network with which it's associated, so that you will not be able to use any wireless networks with that adapter. This could prevent accessing files on a network server, or resolving host names or network addresses, if you are capturing in monitor mode and are not connected to another network with another adapter.

Monitor mode is set with **pcap_set_rfmon**(), and **pcap_can_set_rfmon**() can be used to determine whether an adapter can be put into monitor mode.

packet buffer timeout

If, when capturing, packets are delivered as soon as they arrive, the application capturing the packets will be woken up for each packet as it arrives, and might have to make one or more calls to the operating system to fetch each packet.

If, instead, packets are not delivered as soon as they arrive, but are delivered after a short delay (called a "packet buffer timeout"), more than one packet can be accumulated before the packets are delivered, so that a single wakeup would be done for multiple packets, and each set of calls made to the operating system would supply multiple packets, rather than a single packet. This reduces the per-packet CPU overhead if packets are arriving at a high rate, increasing the number of packets per second that can be captured.

The packet buffer timeout is required so that an application won't wait for the operating system's capture buffer to fill up before packets are delivered; if packets are arriving slowly, that wait could take an arbitrarily long period of time.

Not all platforms support a packet buffer timeout; on platforms that don't, the packet buffer timeout is ignored. A zero value for the timeout, on platforms that support a packet buffer timeout, will cause a read to wait forever to allow enough packets to arrive, with no timeout. A negative value is invalid; the result of setting the timeout to a negative value is unpredictable.

NOTE: the packet buffer timeout cannot be used to cause calls that read packets to return within a limited period of time, because, on some platforms, the packet buffer timeout isn't supported, and, on other platforms, the timer doesn't start until at least one packet arrives. This means that the packet buffer timeout should **NOT** be used, for example, in an interactive application to allow the packet capture loop to ''poll'' for user input periodically, as there's no guarantee that a call reading packets will return after the timeout expires even if no packets have arrived.

The packet buffer timeout is set with **pcap_set_timeout**().

immediate mode

In immediate mode, packets are always delivered as soon as they arrive, with no buffering. Immediate mode is set with **pcap_set_immediate_mode**().

buffer size

Packets that arrive for a capture are stored in a buffer, so that they do not have to be read by the application as soon as they arrive. On some platforms, the buffer's size can be set; a size that's too small could mean that, if too many packets are being captured and the snapshot length doesn't limit the amount of data that's buffered, packets could be dropped if the buffer fills up before the application can read packets from it, while a size that's too large could use more non-pageable operating system memory than is necessary to prevent packets from being dropped.

The buffer size is set with **pcap_set_buffer_size**().

timestamp type

On some platforms, the time stamp given to packets on live captures can come from different sources that can have different resolutions or that can have different relationships to the time values for the current time supplied by routines on the native operating system. See **pcap-tstamp**(7) for a list of time stamp types.

The time stamp type is set with **pcap_set_tstamp_type**().

Reading packets from a network interface may require that you have special privileges:

Under SunOS 3.x or 4.x with NIT or BPF:

You must have read access to */dev/nit* or */dev/bpf**.

Under Solaris with DLPI:

You must have read/write access to the network pseudo device, e.g. */dev/le*. On at least some versions of Solaris, however, this is not sufficient to allow *tcpdump* to capture in promiscuous mode; on those versions of Solaris, you must be root, or the application capturing packets must be installed setuid to root, in order to capture in promiscuous mode. Note that, on many (perhaps all) interfaces, if you don't capture in promiscuous mode, you will not see any outgoing packets, so a capture not done in promiscuous mode may not be very useful.

In newer versions of Solaris, you must have been given the **net_rawaccess** privilege; this is both necessary and sufficient to give you access to the network pseudo-device - there is no need to change the privileges on that device. A user can be given that privilege by, for example, adding that privilege to the user's **defaultpriv** key with the **usermod**(8) command.

Under HP-UX with DLPI:

You must be root or the application capturing packets must be installed setuid to root.

Under IRIX with snoop:
You must be root or the application capturing packets must be installed setuid to root.

Under Linux:
You must be root or the application capturing packets must be installed setuid to root, unless your distribution has a kernel that supports capability bits such as CAP_NET_RAW and code to allow those capability bits to be given to particular accounts and to cause those bits to be set on a user's initial processes when they log in, in which case you must have CAP_NET_RAW in order to capture.

Under ULTRIX and Digital UNIX/Tru64 UNIX:
Any user may capture network traffic. However, no user (not even the super-user) can capture in promiscuous mode on an interface unless the super-user has enabled promiscuous-mode operation on that interface using *pfconfig*(8), and no user (not even the super-user) can capture unicast traffic received by or sent by the machine on an interface unless the super-user has enabled copy-all-mode operation on that interface using *pfconfig*, so *useful* packet capture on an interface probably requires that either promiscuous-mode or copy-all-mode operation, or both modes of operation, be enabled on that interface.

Under BSD (this includes macOS):
You must have read access to */dev/bpf** on systems that don't have a cloning BPF device, or to */dev/bpf* on systems that do. On BSDs with a devfs (this includes macOS), this might involve more than just having somebody with super-user access setting the ownership or permissions on the BPF devices - it might involve configuring devfs to set the ownership or permissions every time the system is booted, if the system even supports that; if it doesn't support that, you might have to find some other way to make that happen at boot time.

Reading a saved packet file doesn't require special privileges.

The packets read from the handle may include a ‘‘pseudo-header’’ containing various forms of packet meta-data, and probably includes a link-layer header whose contents can differ for different network interfaces. To determine the format of the packets supplied by the handle, call **pcap_datalink**(); *https://www.tcp-dump.org/linktypes.html* lists the values it returns and describes the packet formats that correspond to those values.

Do **NOT** assume that the packets for a given capture or ‘‘savefile‘‘ will have any given link-layer header type, such as **DLT_EN10MB** for Ethernet. For example, the "any" device on Linux will have a link-layer header type of **DLT_LINUX_SLL** or **DLT_LINUX_SLL2** even if all devices on the system at the time the "any" device is opened have some other data link type, such as **DLT_EN10MB** for Ethernet.

To obtain the **FILE *** corresponding to a **pcap_t** opened for a ‘‘savefile’’, call **pcap_file**().

Routines

pcap_create(3PCAP)
get a **pcap_t** for live capture

pcap_activate(3PCAP)
activate a **pcap_t** for live capture

pcap_findalldevs(3PCAP)
get a list of devices that can be opened for a live capture

pcap_freealldevs(3PCAP)
free list of devices

pcap_lookupdev(3PCAP)
get first non-loopback device on that list

pcap_open_offline(3PCAP)
open a **pcap_t** for a ‘‘savefile’’, given a pathname

pcap_open_offline_with_tstamp_precision(3PCAP)
: open a **pcap_t** for a ‘‘savefile’’, given a pathname, and specify the precision to provide for packet time stamps

pcap_fopen_offline(3PCAP)
: open a **pcap_t** for a ‘‘savefile’’, given a **FILE ***

pcap_fopen_offline_with_tstamp_precision(3PCAP)
: open a **pcap_t** for a ‘‘savefile’’, given a **FILE ***, and specify the precision to provide for packet time stamps

pcap_open_dead(3PCAP)
: create a ‘‘fake’’ **pcap_t**

pcap_close(3PCAP)
: close a **pcap_t**

pcap_set_snaplen(3PCAP)
: set the snapshot length for a not-yet-activated **pcap_t** for live capture

pcap_snapshot(3PCAP)
: get the snapshot length for a **pcap_t**

pcap_set_promisc(3PCAP)
: set promiscuous mode for a not-yet-activated **pcap_t** for live capture

pcap_set_protocol_linux(3PCAP)
: set capture protocol for a not-yet-activated **pcap_t** for live capture (Linux only)

pcap_set_rfmon(3PCAP)
: set monitor mode for a not-yet-activated **pcap_t** for live capture

pcap_can_set_rfmon(3PCAP)
: determine whether monitor mode can be set for a **pcap_t** for live capture

pcap_set_timeout(3PCAP)
: set packet buffer timeout for a not-yet-activated **pcap_t** for live capture

pcap_set_immediate_mode(3PCAP)
: set immediate mode for a not-yet-activated **pcap_t** for live capture

pcap_set_buffer_size(3PCAP)
: set buffer size for a not-yet-activated **pcap_t** for live capture

pcap_set_tstamp_type(3PCAP)
: set time stamp type for a not-yet-activated **pcap_t** for live capture

pcap_list_tstamp_types(3PCAP)
: get list of available time stamp types for a not-yet-activated **pcap_t** for live capture

pcap_free_tstamp_types(3PCAP)
: free list of available time stamp types

pcap_tstamp_type_val_to_name(3PCAP)
: get name for a time stamp type

pcap_tstamp_type_val_to_description(3PCAP)
: get description for a time stamp type

pcap_tstamp_type_name_to_val(3PCAP)
: get time stamp type corresponding to a name

pcap_set_tstamp_precision(3PCAP)
: set time stamp precision for a not-yet-activated **pcap_t** for live capture

pcap_get_tstamp_precision(3PCAP)
: get the time stamp precision of a **pcap_t** for live capture

pcap_datalink(3PCAP)
: get link-layer header type for a **pcap_t**

pcap_file(3PCAP)
: get the **FILE *** for a **pcap_t** opened for a ‘‘savefile’’

pcap_is_swapped(3PCAP)
: determine whether a ‘‘savefile’’ being read came from a machine with the opposite byte order

pcap_major_version(3PCAP)
pcap_minor_version(3PCAP)
: get the major and minor version of the file format version for a ‘‘savefile’’

Selecting a link-layer header type for a live capture

Some devices may provide more than one link-layer header type. To obtain a list of all link-layer header types provided by a device, call **pcap_list_datalinks**() on an activated **pcap_t** for the device. To free a list of link-layer header types, call **pcap_free_datalinks**(). To set the link-layer header type for a device, call **pcap_set_datalink**(). This should be done after the device has been activated but before any packets are read and before any filters are compiled or installed.

Routines

pcap_list_datalinks(3PCAP)
: get a list of link-layer header types for a device

pcap_free_datalinks(3PCAP)
: free list of link-layer header types

pcap_set_datalink(3PCAP)
: set link-layer header type for a device

pcap_datalink_val_to_name(3PCAP)
: get name for a link-layer header type

pcap_datalink_val_to_description(3PCAP)
pcap_datalink_val_to_description_or_dlt(3PCAP)
: get description for a link-layer header type

pcap_datalink_name_to_val(3PCAP)
: get link-layer header type corresponding to a name

Reading packets

Packets are read with **pcap_dispatch**() or **pcap_loop**(), which process one or more packets, calling a callback routine for each packet, or with **pcap_next**() or **pcap_next_ex**(), which return the next packet. The callback for **pcap_dispatch**() and **pcap_loop**() is supplied a pointer to a *struct pcap_pkthdr*, which includes the following members:

ts	a *struct timeval* containing the time when the packet was captured
caplen	a *bpf_u_int32* giving the number of bytes of the packet that are available from the capture
len	a *bpf_u_int32* giving the length of the packet, in bytes (which might be more than the number of bytes available from the capture, if the length of the packet is larger than the maximum number of bytes to capture).

The callback is also supplied a *const u_char* pointer to the first **caplen** (as given in the *struct pcap_pkthdr* mentioned above) bytes of data from the packet. This won’t necessarily be the entire packet; to capture the entire packet, you will have to provide a value for *snaplen* in your call to **pcap_set_snaplen**() that is sufficiently large to get all of the packet’s data - a value of 65535 should be sufficient on most if not all networks). When reading from a ‘‘savefile’’, the snapshot length specified when the capture was performed will limit the amount of packet data available.

pcap_next() is passed an argument that points to a *struct pcap_pkthdr* structure, and fills it in with the time stamp and length values for the packet. It returns a*const u_c har* to the first **caplen** bytes of the packet on success, and **NULL** on error.

pcap_next_ex() is passed two pointer arguments, one of which points to a *struct*pcap_pkthdr* and one of which points to a *const u_char**. It sets the first pointer to point to a*struct pcap_pkthdr* structure with the time stamp and length values for the packet, and sets the second pointer to point to the first **caplen** bytes of the packet.

To force the loop in **pcap_dispatch**() or **pcap_loop**() to terminate, call **pcap_breakloop**().

By default, when reading packets from an interface opened for a live capture, **pcap_dispatch**(), **pcap_next**(), and **pcap_next_ex**() will, if no packets are currently available to be read, block waiting for packets to become available. On some, but *not* all, platforms, if a packet buffer timeout was specified, the wait will terminate after the packet buffer timeout expires; applications should be prepared for this, as it happens on some platforms, but should not rely on it, as it does not happen on other platforms. Note that the wait might, or might not, terminate even if no packets are available; applications should be prepared for this to happen, but must not rely on it happening.

A handle can be put into ‘‘non-blocking mode’’, so that those routines will, rather than blocking, return an indication that no packets are available to read. Call **pcap_setnonblock**() to put a handle into non-blocking mode or to take it out of non-blocking mode; call **pcap_getnonblock**() to determine whether a handle is in non-blocking mode. Note that non-blocking mode does not work correctly in Mac OS X 10.6.

Non-blocking mode is often combined with routines such as **select**(2) or **poll**(2) or other routines a platform offers to wait for any of a set of descriptors to be ready to read. To obtain, for a handle, a descriptor that can be used in those routines, call **pcap_get_selectable_fd**(). If the routine indicates that data is available to read on the descriptor, an attempt should be made to read from the device.

Not all handles have such a descriptor available; **pcap_get_selectable_fd**() will return **−1** if no such descriptor is available. If no such descriptor is available, this may be because the device must be polled periodically for packets; in that case, **pcap_get_required_select_timeout**() will return a pointer to a **struct timeval** whose value can be used as a timeout in those routines. When the routine returns, an attmept should be made to read packets from the device. If**pcap_get_r equired_select_timeout**() returns **NULL**, no such timeout is available, and those routines cannot be used with the device.

In addition, for various reasons, one or more of those routines will not work properly with the descriptor; the documentation for **pcap_get_selectable_fd**() gives details. Note that, just as an attempt to read packets from a **pcap_t** may not return any packets if the packet buffer timeout expires, a **select**(), **poll**(), or other such call may, if the packet buffer timeout expires, indicate that a descriptor is ready to read even if there are no packets available to read.

Routines

pcap_dispatch(3PCAP)
: read a bufferful of packets from a **pcap_t** open for a live capture or the full set of packets from a **pcap_t** open for a ‘‘savefile’’

pcap_loop(3PCAP)
: read packets from a **pcap_t** until an interrupt or error occurs

pcap_next(3PCAP)
: read the next packet from a **pcap_t** without an indication whether an error occurred

pcap_next_ex(3PCAP)
: read the next packet from a **pcap_t** with an error indication on an error

pcap_breakloop(3PCAP)
: prematurely terminate the loop in **pcap_dispatch**() or **pcap_loop**()

pcap_setnonblock(3PCAP)
: set or clear non-blocking mode on a **pcap_t**

pcap_getnonblock(3PCAP)
get the state of non-blocking mode for a **pcap_t**

pcap_get_selectable_fd(3PCAP)
attempt to get a descriptor for a **pcap_t** that can be used in calls such as **select**(2) and **poll**(2)

pcap_get_required_select_timeout(3PCAP)
attempt to get a timeout required for using a **pcap_t** in calls such as **select**(2) and **poll**(2)

Filters

In order to cause only certain packets to be returned when reading packets, a filter can be set on a handle. For a live capture, the filtering will be performed in kernel mode, if possible, to avoid copying ‘‘uninteresting’’ packets from the kernel to user mode.

A filter can be specified as a text string; the syntax and semantics of the string are as described by **pcap-filter**(7). A filter string is compiled into a program in a pseudo-machine-language by**pcap_compile**() and the resulting program can be made a filter for a handle with **pcap_setfilter**(). The result of**pcap_compile**() can be freed with a call to **pcap_freecode**(). **pcap_compile**() may require a network mask for certain expressions in the filter string; **pcap_lookupnet**() can be used to find the network address and network mask for a given capture device.

A compiled filter can also be applied directly to a packet that has been read using **pcap_offline_filter**().

Routines

pcap_compile(3PCAP)
compile filter expression to a pseudo-machine-language code program

pcap_freecode(3PCAP)
free a filter program

pcap_setfilter(3PCAP)
set filter for a **pcap_t**

pcap_lookupnet(3PCAP)
get network address and network mask for a capture device

pcap_offline_filter(3PCAP)
apply a filter program to a packet

Incoming and outgoing packets

By default, libpcap will attempt to capture both packets sent by the machine and packets received by the machine. To limit it to capturing only packets received by the machine or, if possible, only packets sent by the machine, call **pcap_setdirection**().

Routines

pcap_setdirection(3PCAP)
specify whether to capture incoming packets, outgoing packets, or both

Capture statistics

To get statistics about packets received and dropped in a live capture, call **pcap_stats**().

Routines

pcap_stats(3PCAP)
get capture statistics

Opening a handle for writing captured packets

To open a ‘‘savefile‘‘ to which to write packets, given the pathname the ‘‘savefile’’ should have, call **pcap_dump_open**(). To open a ‘‘savefile‘‘ to which to write packets, given the pathname the ‘‘savefile’’ should have, call **pcap_dump_open**(); to set up a handle for a ‘‘savefile’’, given a **FILE *** referring to a file already opened for writing, call **pcap_dump_fopen**(). They each return pointers to a **pcap_dumper_t**, which is the handle used for writing packets to the ‘‘savefile’’. If it succeeds, it will have created the file if it doesn’t exist and truncated the file if it does exist. To close a **pcap_dumper_t**, call **pcap_dump_close**().

Routines

pcap_dump_open(3PCAP)
open a **pcap_dumper_t** for a ‘‘savefile‘‘, given a pathname

pcap_dump_fopen(3PCAP)
open a **pcap_dumper_t** for a ‘‘savefile‘‘, given a **FILE ***

pcap_dump_close(3PCAP)
close a **pcap_dumper_t**

pcap_dump_file(3PCAP)
get the **FILE *** for a **pcap_dumper_t** opened for a ‘‘savefile’’

Writing packets

To write a packet to a **pcap_dumper_t**, call **pcap_dump**(). Packets written with **pcap_dump**() may be buffered, rather than being immediately written to the ‘‘savefile’’. Closing the**pcap_dumper_t** will cause all buffered-but-not-yet-written packets to be written to the ‘‘savefile’’. To force all packets written to the **pcap_dumper_t**, and not yet written to the ‘‘savefile’’ because they’re buffered by the **pcap_dumper_t**, to be written to the ‘‘savefile’’, without closing the **pcap_dumper_t**, call **pcap_dump_flush**().

Routines

pcap_dump(3PCAP)
write packet to a **pcap_dumper_t**

pcap_dump_flush(3PCAP)
flush buffered packets written to a **pcap_dumper_t** to the ‘‘savefile’’

pcap_dump_ftell(3PCAP)
get current file position for a **pcap_dumper_t**

Injecting packets

If you have the required privileges, you can inject packets onto a network with a **pcap_t** for a live capture, using **pcap_inject**() or **pcap_sendpacket**(). (The two routines exist for compatibility with both OpenBSD and WinPcap/Npcap; they perform the same function, but have different return values.)

Routines

pcap_inject(3PCAP)
pcap_sendpacket(3PCAP)
transmit a packet

Reporting errors

Some routines return error or warning status codes; to convert them to a string, use **pcap_statustostr**().

Routines

pcap_statustostr(3PCAP)
get a string for an error or warning status code

Getting library version information

To get a string giving version information about libpcap, call **pcap_lib_version**().

Routines

pcap_lib_version(3PCAP)
get library version string

BACKWARD COMPATIBILITY

In versions of libpcap prior to 1.0, the **pcap.h** header file was not in a **pcap** directory on most platforms; if you are writing an application that must work on versions of libpcap prior to 1.0, include **<pcap.h>**, which will include **<pcap/pcap.h>** for you, rather than including **<pcap/pcap.h>**.

pcap_create() and **pcap_activate**() were not available in versions of libpcap prior to 1.0; if you are writing an application that must work on versions of libpcap prior to 1.0, either use **pcap_open_live**() to get a handle for a live capture or, if you want to be able to use the additional capabilities offered by using **pcap_create**() and **pcap_activate**(), use an **autoconf**(1) script or some other configuration script to check whether the libpcap 1.0 APIs are available and use them only if they are.

SEE ALSO

autoconf(1), **tcpdump**(8), **tcpslice**(1), **pcap-filter**(7), **pfconfig**(8), **usermod**(8)

AUTHORS

The original authors of libpcap are:

Van Jacobson, Craig Leres and Steven McCanne, all of the Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

The current version is available from "The Tcpdump Group"'s Web site at

https://www.tcpdump.org/

BUGS

To report a security issue please send an e-mail to security@tcpdump.org.

To report bugs and other problems, contribute patches, request a feature, provide generic feedback etc please see the file*CONTRIB UTING.md* in the libpcap source tree root.