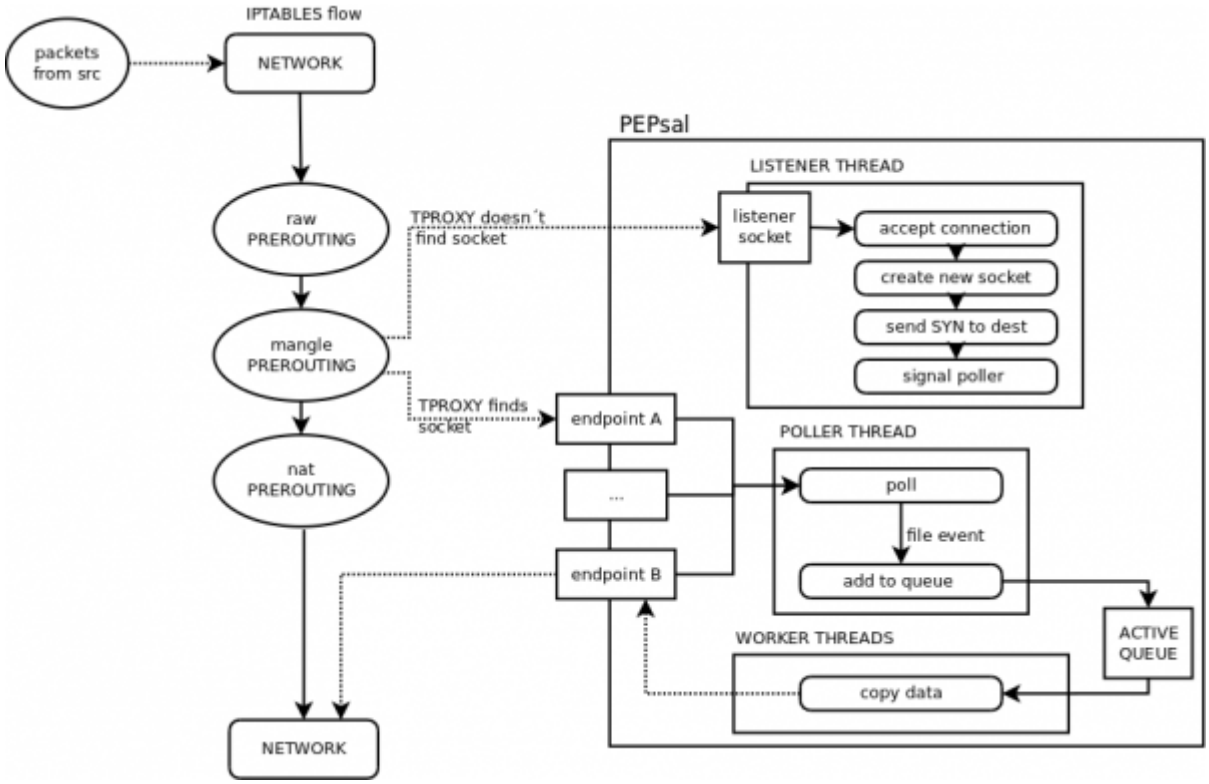
PEPsal design

Network level



Incoming packets are usually handled by the kernel, and are not accessible to the user to modify. In Linux, it is possible to configure the behaviour of the packets (e.g. routing) using the framework **Netfilter**. Netfilter offers various tools for packet filtering, network address translation (NAT), and port translation. It also offers mechanisms for passing packets to a queue accessible from the userspace, and then returning these packets back to the kernel.

PEPsal uses netfilter in order to capture SYN segments traversing the system (e.g. being routed, but not to the host) that are establishing a new connection. The rules to perform this action are set using `iptables`, and are placed in the `mangle` table, PREROUTING chain (before the packets are routed), and the target used is a TPROXY (Transparent Proxy). The target TPROXY works as the following: the system searchs for a socket with the same destination pair (IP address and port) on the local machine. If a socket is found, that means that a transparent proxy is already established, and thus, redirects this packet to said socket. In the other hand, if no socket is found (meaning no connection was established, and that, probably, the segment is a SYN), `netfilter` redirects the packet to a local port, where a socket is listening for incoming connections. In this case, this port is that of PEPsal, which is listening for new connections. At such event, PEPsal 1) accepts the connection, “impersonating” the remote host, and 2), creates a new socket, bound to the original source address and port, and establishes a connection with the original destination host, splitting the TCP connection in two. All subsequent traffic corresponding to any of these two connections is affected by the same rule in iptables, and will be redirected to the corresponding sockets (since now, `netfilter` will be able to find one with the same destination address and port).

Besides the `iptables` rule, in order to locally route the packets to the PEP socket, a mark is used on all packets that match the rules added. Since the packets must not exit the machine, but be redirected to the local sockets, they must be routed to the loopback interface. In order to not interfere

with the routing of all the other packets not affected by PEPsal, a different routing table than the default one is used for marked packets. Thus, the mark added by the TPROXY target (with the `--tproxy-mark` option) is assigned to an specific routing table. The default route for this table routes all packets to the loopback interface (refer to this section for the commands).

Application level

At application level, an application (called **pepsal**), is in charge of accepting the connections on behalf of the original destination, of creating new sockets and establishing a connection with the destination, and of redirecting the traffic between the two endpoints (TCP Splitting).

PEPsal contains several threads to perform the different tasks in parallel. These threads are: the **listener** thread, the **poller** thread, and the **worker** threads.

Listener thread

The **listener** thread is in charge of intercepting the SYN segments trying to start new connection that arrive to the listener socket (by default, bound to the port 5000 on all interfaces). First, the listener socket is opened, and then an infinite loop is entered, waiting for incoming connection requests on this socket. Once a connection request is intercepted (redirected by the TPROXY target that was unable to find a matching socket), the connection is accepted on behalf of the remote host, and a new socket is created, bound to the original source address. Immediately after, a new socket with the original source address is created, and a connection is established with the original destination. A signal is sent to the poller thread, for it to watch the events that are triggered on them. The work of the listener ends here, since all subsequent packets pertaining to these connections will not be redirected the listener port.

Poller thread

The **poller** thread works with the sockets that the **listener** has created (both endpoints). This thread contains a list of threads that are periodically polled, looking for events that may be triggered. Each time the listener thread accepts a new connection, and creates new sockets, these are signaled to the poller thread, that adds them to the local list that is being polled. When the connections are effectively established, an event of type CONNECTED is detected, and the connections are considered as open (moment at which, buffers are created in order to temporarily store the relayed data).

Once opened, the events that can be detected are the reception of data segments, or the end of a connection. In the first case, the sockets are queued to be treated by the worker threads. In the latter case, the connection on both endpoints will be closed.

Worker threads

The **worker** threads are in charge of forwarding the traffic from one endpoint of the proxy to the other. The threads are signaled by the poller when there is data ready to be relayed in at least one proxy. These proxies are placed in a queue by the poller, from which a pool of worker threads will extract them, and copy the information received on one endpoint to the other. The workers will

continue to work until there are no more items in the active queue, and then will wait for a new signal by the poller.

Timer scheduler

There exists another thread, called the **timer scheduler** thread, that runs periodically, checking for pending logs to be displayed on screen (or to the log file), and checking if it is time to call the garbage connections collector.

By default, the garbage connections collector is called every 15 hours, and its purpose is to search for proxies with state “PENDING”, with a syn_time greater than a specified value, i.e. connections that weren't successfully established.