CS60, Lab 3: An ICMP echo responder with IP Raw Sockets

Sergey Bratus, Spring 2017

Date due: Solutions to this lab will be due on Sunday April 30 at 9pm.

The Task: In this lab, you will use the special Linux kernel interface, *raw sockets*, that allows a user program running with root privileges to receive full IP packets and to send full IP packets. With normal sockets, the kernel creates the Ethernet, IP, and TCP/UDP layers on the data you write()/send(), and removes these layers from incoming packets before it gives you their payloads via read()/recv(). With raw sockets, you get the IP full packet as it comes in; and you must create the full packet as it goes out.[1]

The purpose of this lab is to get used to raw sockets. You will use a raw socket to receive ICMP Echo Request packets. You will then manipulate these packets as byte buffers, to make ICMP Echo Reply packets out of them. Then you will use the raw socket to send your newly made reply packets out. To make sure that the kernel itself does not respond to these pings, do

```
   echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

Pinging the host without your user program running will then fail; while your program is running, pinging will succeed. In effect, your program will be doing what the kernel normally does!

Platform: Your solutions must compile and run in the provided Linux virtual machine. You will *not* be able to develop or run them on CS Unix systems (because you don't have root privileges there). In theory, you could make it work on your MacOS, but the code developed on Darwin/MacOS would need to be substantially changed because the raw socket interfaces are different between the OSes, and behave differently. Ask me for hints on how to make it work on MacOS if you are interested.[2]

Submission: Your solutions must be submitted by checking them into the CS department LabGit system at `https://gitlab.cs.dartmouth.edu/`. In your `cs60` project, create a directory called `lab3` and work there. Don't forget to give access to your project to your section's grader and TAs (they must be able to see your code to grade it!)

Raw sockets: Raw sockets for IP ("Internet") packets are created with `AF_INET` and `SOCK_RAW` (instead of the more familiar `SOCK_STREAM` or `SOCK_DGRAM`). There is an additional argument you can use here to limit the packets you get from the socket: the protocol from */usr/include/netinet/in.h.* For this lab, we are interested in `IPPROTO_ICMP`, hence our socket will be

```
sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

After this, we can read raw IP packets from the kernel by calling `recvfrom()`[3] on this socket. See *icmp4-rawrecv.c* for example code.

We must tell the raw socket that we intend to write fully constructed IP packets into it. We do so by setting the socket option `IP_HDRINCL`:

```
const int on = 1;
setsockopt (sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof (on));
```

Now we can send packets with `sendto()`. Note that `sendto()` needs to be passed a pointer to a `struct sockaddr` with the IP address of the correct interface—otherwise, the packet may be sent on `lo`, or may refuse to go out. Raw sending often requires specifying the interface with either its IP directly or with the interface number, because the kernel's normal routing and connection tracking doesn't apply to it.

[1]There are other methods of working with raw sockets, but we will stick to full IP packets; the other possibilities are full Ethernet packets or just the TCP/UDP/ICMP parts.

[2]Essentially, you will need to get raw packets via *libpcap* rather than raw sockets, and then turn on MacOS firewall "stealth mode" to tell the kernel to not respond to pings. You will still use raw sockets to send your crafted replies, of course.

[3]Or `recv()`, but in that case we won't be told which interface the packet came in on—whereas `recvfrom()` fill that information into the `struct sockaddr_in` passed to it.

See *icmp4-rawsend.c* for details (adapted from *icmp4.c* example at `http://www.pdbuchan.com/rawsock/rawsock.html`).

Hints: Unlike the *icmp4-rawsend.c* example, where an echo request packet is built up from scratch, you will be holding a valid received packet. If you copy it into a separate buffer, your echo reply packet is almost ready; you just need to swap a couple of fields, set the correct value for another, and then fix the checksums in IP and ICMP. The latter are important: the raw socket would happily send a packet with incorrect checksums, but the destination system will reject it as damaged on the way.[4]

Submit: Your `icmp4-responder.c`.

Preparing your Virtual Machine: See instructions in `http://www.cs.dartmouth.edu/~sergey/cs60/lab3/vm-config/` and *vm-networking.pdf*.

Extra credit: (inspired by a student who asked, "Where's the catch?")

Remember that IP packets can be fragmented (cf. `https://en.wikipedia.org/wiki/IP_fragmentation_attack` or read about IP fragmentation in the textbook's chapter 10.7). The tool `hping3` can be used to make a fragmented ICMP echo request packet, e.g.,

```
hping3 --icmp -C 8 -K 0 --frag -d 100 192.168.56.100
```

(this command is non-optimal; there are shorter ways to express the same with hping3).

For extra credit, your implementation must respond to fragmented pings as your VM system does (find out how).

Raw sockets not so raw. Note that you will need a different configuration for a socket that can receive raw *fragmented* packets. The standard `socket(AF_INET, SOCK_RAW, `<*proto*>`)` raw socket will reassemble a fragmented IP packet before giving it to you via a raw socket (cf. "man 7 raw", section NOTES). You will need to use another form of a raw socket: `socket(AF_PACKET, SOCK_RAW, ETH_P_IP)`. See "man 7 packet" for the documentation on this kind of socket.

Alternatively, you can use Libpcap to capture packets exactly as they come in on the wire.

More extra credit for MacOS users: Implement the same functionality for MacOS, using Libpcap to sniff incoming ICMP Echo Request packets (MacOS and FreeBSD raw sockets will not produce it, unlike Linux raw sockets). Use PF to block the kernel's own ICMP responses.[5]

Terms and conditions: You are allowed to use any external materials, printed or electronic. You are allowed to discuss problems and technical/C tricks, but the code you submit must be your own: you are not allowed to copy solutions from other students. Abide by the Honor Code; if in doubt, ask.

Late submissions: Throughout the course, you will get *two* free extensions for a late submission, each of 48 hours after a deadline. Once you've used these up, you will lose 10% points of credit for each late assignment, for each late day.

[4]Sometimes packets with incorrect sums are sent deliberately to confused systems in-between, which may not be checking checksums too carefully. The paper `http://insecure.org/stf/secnet_ids/secnet_ids.html` explains how and why.

[5]A rule like `block in quick on en0 inet proto icmp from any to (en0)` would prevent ICMP packets incoming on the interface `en0` from being handed to the kernel, and so they will not be responded to by the kernel. But you will still be able to sniff them from this interface with Libpcap, and respond to them via raw sockets. Choose the interface appropriately for your own machine. Read up on PF syntax at `https://www.openbsd.org/faq/pf/filter.html`.