DAVID N. BLANK-EDELMAN

practical Perl tools: Perl meets Nmap and p0f

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration*. He has spent the past 20+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

AS COMPUTER SECURITY CONCERNS start to become fodder for the more mainstream media and the digital world starts to converge in weird ways, I can easily envision the day where late-night channel flipping will land me on a commercial that goes something like this: "Hey Boys and Girls! Here at K-Tel Records and Software, have we got something special for you. That's right, a compilation of all of your favorite security tools. Who could forget such great hits as . . ." (Hey, don't scoff at the idea of this sort of convergence. Do a Google search on "Nmap porn" and then see if you are still snickering.)

If such a thing does come to pass, there are two tools that would definitely be on that album: Nmap and p0f. We're going to take a look at how to drive both of these tools from Perl so you can build some interesting applications that take advantage of their superpowers. For those of you who are new on the scene and haven't heard of either of these tools, let me give you a very quick rundown.

Nmap

Nmap (http://insecure.org/nmap/index.html) is one of the most impressive security scanners you'll ever encounter. This free tool can pull out virtually every known trick in the book to probe even huge networks quickly and return a list of devices present on your network. In most cases it can tell you what network ports are open, the services provided on those ports, and often even the version of the operating system these devices are running. If you've never played with Nmap, you should do so (on a network you have the legal and ethical right to explore).

Here's an excerpt from a sample Nmap scan of a host:

```
$ sudo nmap -O -sV 192.168.0.9
Starting Nmap 4.20 ( http://insecure.org ) at 2007-09-27 22:07 EDT
Interesting ports on 192.168.0.9:
Not shown: 1693 filtered ports
PORT      STATE  SERVICE        VERSION
22/tcp    open   ssh            OpenSSH 4.3 (protocol 1.99)
139/tcp   open   netbios-ssn
445/tcp   open   microsoft-ds   Microsoft Windows XP microsoft-ds
3389/tcp  open   microsoft-rdp  Microsoft Terminal Service
MAC Address: 00:0B:DB:54:3A:22 (Dell ESG Pcba Test)
```

```
Device type: general purpose
Running (JUST GUESSING) : Microsoft Windows 2000|XP|2003 (90%)
Network Distance: 1 hop
Service Info: OS: Windows
Nmap finished: 1 IP address (1 host up) scanned in 44.524 seconds
```

pof

p0f (http://lcamtuf.coredump.cx/p0f.shtml) describes itself as a "passive OS fingerprinting tool." In some regards it can do something even more impressive than Nmap. Nmap functions by spewing all sorts of interesting packets at its targets to gather information, but p0f is totally passive. It doesn't need to send a single packet to work its magic. p0f can sit on a network and just listen, showing you information about the hosts on the network based on what it hears.

Here's some sample output from p0f showing my home network being scanned by other hosts over my cable modem connection (thanks guys!):

```
$ p0f -i en1 -p
p0f - passive os fingerprinting utility, version 2.0.8
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'en1', 262 sigs (14 generic, cksum 0F1F5CA2), rule: 'all'.
202.163.213.11:64162 - FreeBSD 6.x (2) (up: 1148 hrs)
  -> 192.168.0.3:49153 (distance 22, link: ethernet/modem)
85.164.236.96:28029 - Windows 2000 SP2+, XP SP1+ (seldom 98) [priority1]
  -> 192.168.0.3:49153 (distance 27, link: pppoe (DSL))
90.193.162.36:4888 - Windows 2000 SP4, XP SP1+ [priority1]
  -> 192.168.0.3:49153 (distance 20, link: ethernet/modem)
```

Clearly these are really powerful tools. Wouldn't it be great to be able to harness their power from your own programs? Let's work on that now.

Nmap from Perl

There are two very capable Perl modules for Nmap control on CPAN: Nmap::Parser and Nmap::Scanner. They have roughly the same functionality. I tend to like Nmap::Parser better because it doesn't force you into using iterators (i.e., get_next_port()), but you should look at both and see which one catches your fancy.

Both Nmap::Parser and Nmap::Scanner have two modes of operation: batch and event/callback. With the first mode you specify the parameters for your scan using the conventions of that module (sometimes it is just passing explicit command-line arguments to Nmap), tell the module to kick off the scan, and then sit back and wait for the scan to complete. Once Nmap has done its work, the module makes available to you a few Perl data structures containing the results. With these answers in hand, your program can go off and make use of all of the yummy data you've collected.

For the event/callback-based way of working, in addition to defining the scan parameters at the start, you also provide the module with code that gets run as certain events are triggered (i.e., your code gets a callback at certain points in the scan run). These events can include things such as "found a host" or "found an open port."

There are a few advantages of using event-based approaches over batch approaches. For instance, if you are performing a scan on a huge network block an event-based program will likely be less memory-intensive, because

you have the chance to store only the data you care about instead of the results from the entire scan. Your program can be a bit more responsive because you can choose to act on partial results, perhaps spinning off tasks related to the results in parallel (see the February 2007 column) or even aborting early if you see fit. The downside of this approach is that your program is now responsible for the collection and storage of the data found, something the batch modes make easy. Just FYI: Nmap::Scanner has a few more event-based hooks than Nmap::Parser, but they both work roughly the same way.

Let's look at one batch scan and one event-based scan using Nmap::Parser.

This code looks for all of the hosts specified on the command line that have their HTTP, HTTPS, and SSH ports open all at the same time:

```
use Nmap::Parser;

my $nmapexec = "/opt/local/bin/nmap";  # location of executable
my $nmapargs = "-p 80,443,22";         # look for http, https, and SSH

my $np = new Nmap::Parser;

# scan the hosts listed on the command line
$np->parsescan( $nmapexec, $nmapargs, @ARGV );

# iterate over the result set looking for the hosts that have
# all 3 ports open
for my $host ( $np->all_hosts() ) {
    my @open = $host->tcp_open_ports();
    print "found: " . $host->hostname() .
         " (" . $host->addr() . ")" . "\n"
       if scalar @open == 3;
}
```

Here's the same code using a callback-based approach:

```
use Nmap::Parser;

my $nmapexec = "/opt/local/bin/nmap";  # location of executable
my $nmapargs = "-p 80,443,22";         # look for http, https, and SSH (all 3)

my $np = new Nmap::Parser;

# call print_all_open each time we finish scanning a host
$np->callback( \&print_all_open );

# scan the hosts listed on the command line
$np->parsescan( $nmapexec, $nmapargs, @ARGV );

# print only the hosts that have all three ports in question open
sub print_all_open {
    my $host = shift;
    my @open = $host->tcp_open_ports();

    print "found: " . $host->hostname() .
         " (" . $host->addr() . ")" . "\n"
       if scalar @open == 3;
}
```

So what can you do programmatically with the results of an Nmap scan? There are tons of applications, including network visualization, policy enforcement, and security testing. Let's look at an example that demonstrates the first two on that list. Let's say our employer is either really paranoid or just highly idiosyncratic and has a strict policy that the MySQL servers on site are not allowed to provide mail services. Here's some code that maps a

network block looking for the hosts with either an SMTP or a standard MySQL port open:

```
use Nmap::Parser;
use Graph::Easy;

my $nmapexec = "/opt/local/bin/nmap";    # location of executable
my $nmapargs = "-p 25,3306";
my %services = ( 25 => 'SMTP', 3306 => 'MYSQL' );
my $np = new Nmap::Parser;

# scan the hosts listed on the command line
$np->parsescan( $nmapexec, $nmapargs, @ARGV );

my $graph = Graph::Easy->new();

for my $host ( $np->all_hosts() ) {
   # graph the last octet of the IPv4 address
   my $hostnumber = ( split( /\./, $host->addr() ) )[3];
   for my $port ( $host->tcp_open_ports() ) {
      $graph->add_edge( $services{$port}, $hostnumber );
   }
}

print $graph->as_graphviz();
```

If we run this script like so:

```
$ perl nmap.pl 192.168.0.0/24|dot -Tpng -o fig1.png
```

we get a picture like the one in Figure 1.

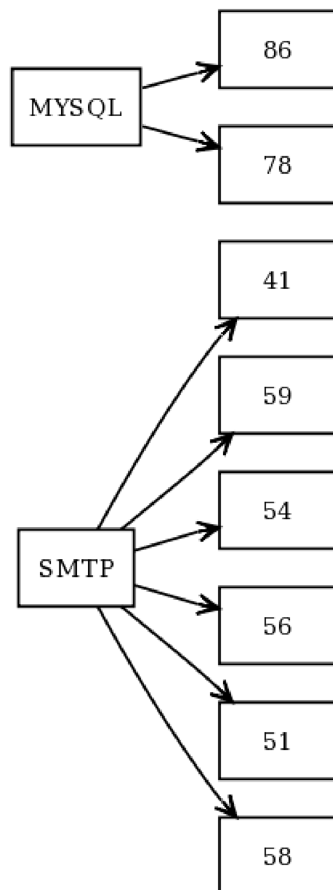

FIGURE 1

Even a simple picture like this can be helpful. Besides providing an easy way to see if our policy is being followed, it also provides an easy-to-scan display of the hosts providing a service. Should 192.168.0.41 be offering SMTP services when we did not expect it, it would likely be what we in the technical world call "a bad thing." It would be pretty easy to write code that compares the results between Nmap runs and highlights any differences (perhaps changing the color of a box in the final output). Nmap::Parser comes with a sample script called nmap2sqlite that stores the output of a scan in an SQLite database if you are looking for a little help in that direction.

There are plenty of visualization tools besides those offered by Graphviz/Graph::Easy that would be happy to eat the Nmap data we collected and spit out pretty diagrams of the network.

pof from Perl

Nmap is a fantastic tool for mapping a known (and potentially large) set of hosts, but what if you don't know which hosts are interesting to you? In the case of publicly visible hosts such as Internet-facing Web servers, you don't really have a good way of knowing which hosts will connect to you. p0f is a good choice for those situations where the network traffic comes to you. It can provide some basic information about a host simply by listening to the packets sent by that host.

This information can be useful for any number of reasons. Perhaps you want to know roughly what percentage of your loyal Web site users are using Linux. Maybe you want to create a policy that says machines running operating systems written near Seattle should be scanned for vulnerabilities when they first arrive on a network. p0f can help with these goals. I know of a commercial anti-spam product that relies on information from p0f to make decisions

about how to handle incoming connections (on the theory that hijacked Windows boxes make up the vast majority of a spammer's arsenal these days).

p0f itself can run in two different modes. The most common usage is to have p0f listen for packets from the wire or read a standard tcpdump capture file and output the fingerprint results to STDOUT or to another file. Another possibility is to run p0f as a daemon that receives packets from another program passed over a local stream socket. This works well in those cases where you already have something listening for packets that would like to consult p0f as it goes along. The Perl module Net::P0f can handle both of these modes. We're only going to look at code for operating in the first mode. See the submodule Net::P0f::Backend::Socket in the Net::P0f distribution for information on using p0f in socket-read mode.

Two meta things need to be said about Net::P0f before we see some code:

- The module hasn't been updated since 2005, although it seems to work even with recent versions of p0f.
- Net::P0f operates using callbacks, similar to what we saw for Nmap::Parser. You are expected to define a subroutine that will receive fingerprint information as it is produced. If you want to keep tabs on everything that has been seen since the program first started listening, you'll need to write the coalesce code yourself.

Let's look at a very simple example so you can get an idea of how things can work using this module. This code listens for 25 initial connect packets (SYN) destined for any host on the network and reports back an IP address and details for any machine p0f determines to be a Windows box:

```
use Net::P0f;

# listen on /dev/en1 for all packets
my $p0f = Net::P0f->new( interface => 'en1', promiscuous => 1 );

# listen for 25 packets and then exit, handing each packet to our callback
$p0f->loop( callback => \&show_win32, count => 25 );

# print information on host if it is identified as a Windows machine
sub show_win32 {
   my ( $self, $header, $os_info, $link_info ) = @_;

   print $header->{ip_src} . ":"
      . $os_info->{genre} . " "
      . $os_info->{details} . "\n"
      if $os_info->{genre} eq "Windows";
}
```

The good news is that using Net::P0f really doesn't get any more sophisticated than that. From here you can use any of your standard Perl techniques to slice, dice, collate, or analyze the data you collect. You could also easily add code from our last section to have Nmap scan hosts running certain operating systems or coming from selected link media identified by p0f.

Now you have the information you need to write your own applications that harness the power of Nmap and p0f. Create something interesting and report back! Take care, and I'll see you next time.