Monitoring LAN devices with Perl

LIGHT INTO THE DARKNESS

They say darkness is the friend of thieves, but the Perl daemon in this month's column illuminates dastardly deeds, exposing hidden activities and alerting the admin when things seem to be going awry.

BY MICHAEL SCHILLI

Users normally don't get to see what's going on under the covers of a LAN. One hidden activity is packet addressing on the last hop of a route, which includes discovering a device's unique MAC address to match an IP address. This activity is the domain of the ARP protocol.

Watching all MAC addresses currently in use can lead to interesting conclusions about who is using or abusing a local network.

You may recall my fellow columnist Charly Kühnast talked about arpalert [4] previously [2]. The daemon monitors ARP requests and compares their MAC addresses with a whitelist.

Unknown MAC addresses trigger an alert. However, duplicate alerts can occur for the same incident, and the accompanying documentation leaves much to be desired.

Luckily, the Net::Pcap and NetPacket::Ethernet modules from CPAN make it fairly easy to craft a Perl script to extract the MAC address from the packets whizzing around on your LAN.

Rose, the object-oriented database mapper, provides an easy way to store the data you harvest in a MySQL database, which you can review later when you have time. If you need to determine which devices have been active on your LAN in, say, the last 24 hours, you can simply call the script *lastaccess* (shown later in Listing 6), which produces the output shown in Figure 1.

Sniffing as Root

Just like the graphical network sniffer, *capture*, which I discussed in issue 49 ([3]), the *arpcollect* script in Listing 1 first switches the network card in your computer to promiscuous mode. In this mode, the card not only picks up packets addressed to it, but passes any packets it finds to the sniffing script.

Root privileges are required for this, and line 11 checks for them. If you do not have root privileges, the script simply quits with an error message.

The *lookupdev()* function called in line 16 returns the name of the first available network device. If you only have one NIC, this will be "*eth0*".

The following call to *open_live()* enters an infinite loop (the timeout has been disabled with a value of -1), which reads the first 128 bytes of every incoming packet, and then immediately calls the *callback* function. The function is passed both to the local network address and mask in *$user_data*, and the raw data for the packet in *$raw_packet*.

The NetPacket::Ethernet module decodes the

www.sxc.hu

Figure 1: The lastaccess script reveals which devices have been active on your LAN in the past 24 hours.

Ethernet frame, and it reveals the hex-formatted MAC source address in the *src_mac* hash key.

As the address does not yet include the typical colon separators after every second digit, line 50 uses a regular expression to insert the separators.

In lines 62 through 65, *arpcollect* references the packet's IP address to check whether the packet originated with a device on the local network.

It discovers the IP address by reading the Ethernet packet payload, which is extracted by the NetPacket::Ethernet module's *strip()* function. The resulting raw IP packet is unpacked by the *NetPacket::IP* module's *decode()* function and the IP source address is revealed by the *src_ip* hash key.

If a bitwise AND of the IP address and the network address returns the network address, we can then assume the packet was sent by a device on the local network and that it is relevant for further processing.

The *event_add()* method of the *WatchLAN* database object accepts the IP address and the MAC address and drops them into the database for later analysis.

One Minute Buffer

The WatchLAN.pm module implements the storage layer. It would be impractical to write each packet to the database straight away; this would involve multiple write operations per second, even on a low-activity network. Additionally, a table with millions of lines would consume valuable disk space and computational resources.

This is the reason why *WatchLAN.pm* (see Listing 3) first stores the incoming packet addresses in a temporary hash, which is then transferred to the database once every minute.

A counter is incremented for each IP/MAC combination and stored in the *counter* column of the *activity* database table by the *cache_flush()* method. The *flush_interval* parameter in the WatchLAN constructor determines how often the cache is flushed.

The future date of the next flush operation is calculated by adding *flush_interval* to the current time and stored in the instance variable *next_update*.

Listing 2 shows the shell commands that are needed in order to create a new MySQL database.

The SQL commands to set up all used tables (Figure 2) are in a separate file *sql.txt*, shown in Figure 3.

As shown in Figure 3, foreign keys are used to link the main *activity* table to the *device* and *ip_address* tables.

device stores the MAC addresses along with the device data; *ip_address* simply stores IP addresses and assigns them a sequence number.

Listing 1: arpcollect

```
01 #!/usr/bin/perl -w
02 use strict;
03 use Net::Pcap;
04 use NetPacket::IP;
05 use NetPacket::Ethernet;
06 use Socket;
07 use WatchLAN;
08
09 die "You need to be root ",
10     "to run this.\n"
11   if $> != 0;
12
13 my($err, $netaddr, $netmask);
14
15 my $dev =
16  Net::Pcap::lookupdev(\$err);
17
18 Net::Pcap::lookupnet($dev,
19  \$netaddr, \$netmask, \$err)
20   and die
21 "lookupnet $dev failed ($!)";
22
23 my $object =
24   Net::Pcap::open_live($dev,
25         128, 1, -1, \$err);
26
27 my $db = WatchLAN->new();
28
29 Net::Pcap::loop($object, -1,
30  \&callback,
31  [ $netaddr, $netmask ]);
32
33 ################################
34 sub callback {
35 ################################
36  my ($user_data, $hdr,
37      $raw_packet)      = @_;
38
39  my ($netaddr, $netmask) =
40    @$user_data;
41
42  my $packet =
43    NetPacket::Ethernet
44    ->decode($raw_packet);
45
46  my $src_mac =
47    $packet->{src_mac};
48
49  # Add separating colons
50  $src_mac =~
51    s/(..)(?!$)/$1:/g;
52
53  my $edata =
54    NetPacket::Ethernet::strip
55    ($raw_packet);
56
57  my $ip =
58    NetPacket::IP->decode(
59   $edata);
60
61    # Coming from local network?
62  if (
63   ( inet_aton($ip->{src_ip}) &
64     pack('N', $netmask)
65   ) eq pack('N', $netaddr)
66    ) {
67   $db->event_add($src_mac,
68               $ip->{src_ip});
69  }
70 }
```

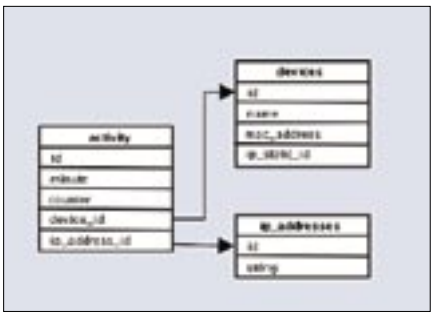

Figure 2: The three tables in the database schema.

Storing the addresses in the main table would not only waste storage space, but also generate redundant data.

Special Treatment for MySQL

MySQL doesn't make it easy for the Rose::DB loader to detect these relations automatically. According to Rose::DB author John Siracusa, you need to provide correct *REFERENCES* clauses for foreign key declarations and also define an *index* on both the referencing and referenced columns.

Once the SQL definition has been established as shown, WatchLAN.pm simply needs to call the *make_classes* method to have Rose::DB contact the database and autonomously define the complete object wrapper for all the tables and columns, including those referenced from separate tables.

What Do I Know?

The WatchLAN module calls the Rose loader whenever an application calls *use WatchLAN*. WatchLAN stores the database schema as an object-oriented abstraction in the Perl namespace below *WatchLAN::*.

When the *cache_flush()* method needs to save temporary hash data in the database, *WatchLAN* responds by creating a new *$activity* object of the WatchLAN::Activity class.

Listing 2: New database

```
01 #!/bin/sh
02 DBNAME=watchlan
03 mysqladmin -f -uroot drop ➥
   $DBNAME
04 mysqladmin -uroot create ➥
   $DBNAME
05 mysql -uroot $DBNAME <sql.txt
```

This not only facilitates updates in the *activity* table but also in referenced tables like *devices* and *ip_addresses*. The innocent-looking construct:

```
$activity->device({
  mac_address => $mac });
```

causes two things to happen later, when the object's *save()* method gets called. If the *devices* table does not have an entry for the device with the given MAC address yet, it creates a new record there. In the main table *activity*, it adds the newly created device C<id> as a foreign key in the *device_id* column.

In contrast to the method call above, curly braces are not required to create a new entry in the *activity* table that does not reach out to referenced tables.

A call to *$activity->counter($counter)* sets the counter column value *$counter* in the current *activity* record to the value of *$counter*. After a call to *save()* in line 93, it gets flushed to the database. By the time this is done, *cache_flush()* is finished and can clear its cache in line 96. It then calculates the next cache flush interval, and returns to the calling function. *device_add()* performs similar functions, it either inserts a new device along with a MAC address, or modifies the entry for an existing device.

The call to *$device->load(speculative => 1);* loads a record from the *devices* table to match the MAC address that was specified previously in the *WatchLAN::Device* constructor.

The *load* method works in this case because we defined the *mac_address* column as the unique key, using *UNIQUE(mac_address)* when creating the database.

Rose detects this and then allows us to load the record based on this criterion. If this were not so, it would then be necessary to formulate a query to search for the record.

The *speculative* parameter specifies that it is ok for a record not to exist. If so, a subsequent call to *save()* will create the record.

Avoiding Waste

Rose has a fairly wasteful approach to database connections. Each new WatchLAN::Activity class object calls the DBI module's *connect()* function, and Rose drops the connection when it is done with the object. This avoids undesirable side-effects when working with database transactions, but it is obviously a waste of time if you are working without them, as in our case.

Simply loading the *Apache::DBI* module causes it to interfere with how Perl's DBI module handles connections and ensures that only a single persistent database connection behind the scenes is being used.

The *devices* table not only holds the MAC addresses, but it also assigns expressive device names. Thus, *00:11:11:5b:ed:46* becomes "Mike's Linux Box". At the same time, the entry proves that this is a trusted device on the local network.

On the other hand, if your neighbor tries to steal bandwidth off your wireless LAN, *arpcollect* will detect the intrusion and enter the MAC address in the *devices* table but leave the *name* empty.

The monitoring script *arpemail*, which we will be looking at later, notices this irregularity and notifies the admin by email. To enter MAC addresses for known devices on the LAN, the script *namedev* reads the entries in its DATA section line by line. The format used

```
CREATE TABLE ip_addresses (
  id      INTEGER PRIMARY KEY
          AUTO_INCREMENT,
  string VARCHAR(15),
  UNIQUE(string)
) Type=InnoDB;

CREATE TABLE devices (
  id              INTEGER PRIMARY KEY
                  AUTO_INCREMENT,
  name            VARCHAR(255),
  mac_address     VARCHAR(17),
  ip_static_id    INTEGER,

  FOREIGN KEY (ip_static_id)
    REFERENCES ip_addresses(id),
  INDEX(ip_static_id),
  UNIQUE(mac_address)
) Type=InnoDB;

CREATE TABLE activity (
  id              INTEGER PRIMARY KEY
                  AUTO_INCREMENT,
  minute          DATETIME,
  counter         BIGINT,
  device_id       INTEGER NOT NULL,
  ip_address_id INTEGER NOT NULL,

  FOREIGN KEY (device_id)
    REFERENCES devices(id),
  FOREIGN KEY (ip_address_id)
    REFERENCES ip_addresses(id),
  INDEX(device_id),
  INDEX(ip_address_id)
) Type=InnoDB;

-- vim:filetype=sql
"sql.txt" 38L, 861C 38,1          All
```

Figure 3: These SQL commands create the required MySQL database.

here is exactly what the original *arpalert* script [4] expects in its configuration file.

Alarm in Sector B

After running *namedev*, only unknown devices on the LAN will have a *name* value of NULL in the *device* table. To determine all *activity* entries referencing *device* entries with a NULL *name* field, we must JOIN the two tables. If we also need the IP address for the entry, no less than three tables are involved. Rose handles this behind the scenes. The *arpemail* script (see Listing 5) notifies the system administrator whenever a previously unknown MAC address is detected in the *device* table. *arpemail* uses the *WatchLAN::Activity::Manager* class to search for records by running an SQL query. The *get_activity()* method in line 14 queries the *activity* table, and the

Listing 3: WatchLAN.pm

```
001 ################################
002 package WatchLAN;
003 ################################
004 use strict;
005   # share a single DB conn
006 use Apache::DBI;
007 use Rose::DB::Object::Loader;
008 use Log::Log4perl qw(:easy);
009 use DateTime;
010
011 my $loader =
012   Rose::DB::Object::Loader
013   ->new(
014    db_dsn =>
015  'dbi:mysql:dbname=watchlan',
016    db_username => 'root',
017    db_password => undef,
018    db_options  => {
019      AutoCommit => 1,
020      RaiseError => 1
021    },
022    class_prefix =>
023      'WatchLAN'
024   );
025
026 $loader->make_classes();
027
028 ################################
029 sub new {
030 ################################
031  my ($class) = @_;
032
033  my $self = {
034   cache          => {},
035   flush_interval => 60,
036   next_update    => undef,
037  };
038
039  bless $self, $class;
040  $self->cache_flush();
041
042  return $self;
043 }
044
045 ################################
046 sub event_add {
047 ################################
048  my ($self, $mac, $ip) = @_;
049
050  $self->{cache}->
051   {"$mac,$ip"}++;
052
053  $self->cache_flush()
054    if time() >
055      $self->{next_update};
056 }
057
058 ################################
059 sub cache_flush {
060 ################################
061  my ($self) = @_;
062
063  for my $key (
064   keys %{ $self->{cache} })
065  {
066   my ($mac, $ip) =
067     split /,/, $key;
068
069   my $counter =
070     $self->{cache}->{$key};
071
072   my $minute =
073     DateTime->from_epoch(
074    epoch =>
075      $self->{next_update} -
076      $self->{flush_interval},
077    time_zone => "local",
078     );
079
080   my $activity =
081     WatchLAN::Activity->new(
082    minute => $minute);
083
084   $activity->device({
085     mac_address => $mac });
086
087   $activity->ip_address({
088     string => $ip });
089
090   $activity->counter(
091    $counter);
092
093   $activity->save();
094  }
095
096  $self->{cache}       = {};
097  $self->{next_update} =
098    time() - (
099   time() %
100     $self->{flush_interval})
101    + $self->{flush_interval};
102 }
103
104 ################################
105 sub device_add {
106 ################################
107  my ($self, $name,
108      $mac_address) = @_;
109
110  my $device =
111    WatchLAN::Device->new(
112   mac_address =>
113     $mac_address);
114  $device->load(
115   speculative => 1);
116
117  $device->name($name);
118  $device->save();
119 }
120
121 1;
```

Listing 4: namedev

```
01 #!/usr/bin/perl
02 use strict;
03 use warnings;
04 use WatchLAN;
05
06 my $db = WatchLAN->new();
07
08 while (<DATA>) {
09  if (/^#\s+(.*)/) {
10   my $name     = $1;
11   my $nextline = <DATA>;
12   chomp $nextline;
13   my ($mac, $ip, $ip_change)
14     = split ' ', $nextline;
15   $db->device_add($name,
16    $mac);
17  }
18 }
19
20 __DATA__
21
22 # Slimbox
23 00:04:20:03:00:0d 192.168.0.74
   ip_change
24
25 # Laptop Wireless
26 00:16:6f:8d:58:db 192.168.0.75
   ip_change
27
28 # Laptop Wired
29 00:15:60:c3:44:10 192.168.0.71
   ip_change
30
31 # Mike's Linux Box
32 00:11:11:5b:ed:46 192.168.0.18
33
34 ...
```

with_objects parameter ensures that the data referenced in the *device* and *ip_address* tables are also extracted. Rose enumerates the tables as *t1* (*activity*), *t2* (*device*), and *t3* (*ip_address*); thus the abstracted SQL query:

```
query => [ "t2.name" =>undef ]
```

in line 19 refers to the *device* table and checks for entries with a value of NULL in the *name* column. The result of this query is a reference to an array of matching database entries, each of which is a *WatchLAN::Activity*-type object, which provides methods for querying its own column values and the values in the referenced tables.

arpemail "remembers" devices it has tagged as being suspicious in a file-based *Cache::File*-type cache to avoid issuing repeat messages with the same warning. If a cache entry exists for the MAC address *$mac*, then the following construct returns a false value:

```
 !$cache->get($mac) &&
($cache->set($mac, 0) || 1);
```

If *$mac* is unknown, the cache's *get* method will return false, which will then get negated to true, which causes the statement following the logical AND to be executed.

The subsequent *set* method is used to add the new value to the cache, and the following *||1* always makes it return a true value, no matter what the actual return value of *set* is.

The enclosing *grep* command in line 24 ff. uses this twisted logic and filters MAC addresses stored in the cache from the list of potential bandwidth thieves stored in *$events*.

Listing 5: arpemail

```
01 #!/usr/bin/perl -w
02 use strict;
03 use WatchLAN;
04 use Mail::Mailer;
05 use Cache::File;
06 use Template;
07 my $cache =
08   Cache::File->new(
09  cache_root =>
10    "$ENV{HOME}/.arpemail");
11
12 my $events =
13   WatchLAN::Activity::Manager
14   ->get_activity(
15  with_objects => [
16   'device', 'ip_address'
17  ],
18  query =>
19    [ "t2.name" => undef ],
20  sort_by => ['minute'],
21   );
22
23 $events = [
24  grep {
25   my $mac =
26     $_->device()
27       ->mac_address();
28   !$cache->get($mac)
29     && ($cache->set($mac, 0)
30    || 1);
31    } @$events
32 ];
33
34 exit 0 unless @$events;
35
36 my $mailer =
37   new Mail::Mailer;
38 $mailer->open(
39  {
40   'From' => 'me@_foo.com',
41   'To' => 'oncall@_foo.com',
42   'Subject' =>
43    "*** New MAC detected ***",
44  }
45 );
46
47 my $t = Template->new();
48 $t->process(\*DATA,
49  { events => $events },
50  $mailer)
51   or die $t->error();
52
53 close($mailer);
54
55 __DATA__
56 [% FOREACH e = events %]
57  When: [% e.minute %]
58  IP: [% e.ip_address.string %]
59  MAC: [% e.device.mac_address
   %]
60
61 [% END %]
```

Listing 6: lastaccess

```
01 #!/usr/bin/perl -w
02 use strict;
03 use WatchLAN;
04
05 my $reachback =
06   DateTime->now(
07  time_zone => "local")
08   ->subtract(
09  minutes => 60 * 24);
10
11 my $events =
12   WatchLAN::Activity::Manager
13   ->get_activity(
14  query => [
15   minute =>
16     { gt => $reachback },
17  ],
18  sort_by => ['minute'],
19   );
20
21 my %latest = ();
22
23 for my $event (@$events) {
24  $latest{ $event->device_id()
25    } = $event;
26 }
27
28 for my $id (keys %latest) {
29  my $event = $latest{$id};
30  my $name  =
31    $event->device()->name();
32  $name ||=
33    "unknown (id=$id)";
34  printf "%23s: %s ago\n",
35    $name, time_diff(
36   $event->minute());
37 }
38
39 ################################
40 sub time_diff {
41 ################################
42  my ($dt) = @_;
43
44  my $duration =
45    DateTime->now(
46   time_zone => "local") -
47    $dt;
48
49  for (
50   qw(hours minutes seconds))
51  {
52   if (my $n =
53    $duration->in_units($_))
54   {
55    my $unit = $_;
56    $unit =~ s/s$//
57      if $n == 1;
58    return "$n $unit";
59   }
60  }
61 }
```

If the array referenced by *$events* appears to be empty, then the *arpemail* surveillance script, which is called as a regular Cronjob, will simply terminate.

In case there are new devices to be reported, the alert message is formatted by the template toolkit. The template that is stored in the *DATA* section at the end of the script is passed a reference to the *$events* array and uses a *FOREACH* loop to iterate over the entries. The template toolkit's quirky but very practical syntax enables us to call the *$e->ip_address()->string()* method chain as *e.ip_address.string*.

arpemail then uses the CPAN *Mail::Mailer* module to connect to the local mailer. It then mails the message to the system administrator listed in the *To* field in line 41.

What's Been Going On?

To see which devices have visited your LAN over the past 24 hours, the *lastaccess* script uses the CPAN *DateTime* module to specify a point in time that was exactly 24 hours ago.

The Rose manager then fires off an SQL query that will return every single event that has occurred since this point, sorted by the event time rounded to minutes as stored in the *minute* column in the database.

The *%latest* hash only stores the last events for various MAC addresses; *lastaccess* continually overwrites the same MAC addresses with the latest values. It would be preferable to leave calculations of this kind to the database, however, the Rose object wrapper does not support aggregation functions such as *MAX()* with *GROUP BY* yet. Judging by the current pace of development, however, this feature might well have been implemented by the time this issue hits the news stands.

listaccess defines a *time_diff* function in line 40 in order to calculate the human-readable time difference between the second values.

The text substitution in line 56 transforms the plural time units to singular units if the result is a single unit. The output from *lastaccess* looks much like Figure 1. You could extend the *arpemail* script to set static IPs for specific devices in the *device* table, following *arpalert*'s example ([4]), and send an alert whether a device with a static IP is suddenly using a different address.

As always, there simply are no bounds to the developer's ingenuity now that the framework has been well established, and fortunately, you will be able to access the data in the database to your heart's content. ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/76/Perl*

[2] "Arp Watch" by Charly Kühnast *Linux Pro Magazine*, December 2006, pg. 55

[3] "Perl: Traffic Control" by Michael Schilli, *Linux Pro Magazine*, December 2004, *http://www.linux-magazine.com/issue/49*

[4] The original Arpalert script: *http://arpalert.org*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.