DAVID N. BLANK-EDELMAN

practical Perl tools: attachments

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 24+ 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

"From attachment springs grief, from attachment springs fear. From him who is wholly free from attachment there is no grief, whence then fear?"

—*The Dhammapada: The Buddha's Path of Wisdom*, translated from the Pali by Acharya Buddharakkhita

THE BUDDHA WAS PROBABLY NOT talking about mail attachments in the Dhammapada. But there's enough fear and loathing around mail attachments that it would be worthwhile to see whether we can use Perl to reduce the suffering around this issue. In this column we're going to look at three different tasks related to attachments and how to work through each using some precision Perl modules.

Before we get started we first have to understand a little background information about attachments. To have an (interoperable) attachment, a mail message has to be constructed in a certain way. The blueprint for how this message is constructed is defined using the Multipurpose Internet Mail Extensions (MIME) standards. I say "standards" not simply to watch Nick Stoughton's ears perk up (hi, Nick!) but because it takes at least six documents to construct this particular snake pit (there are more):

- RFC2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, N. Freed and N. Borenstein, 1996.
- RFC2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, N. Freed and N. Borenstein, 1996.
- RFC2047: MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text, K. Moore, 1996.
- RFC2077: The Model Primary Content Type for Multipurpose Internet Mail Extensions, S. Nelson and C. Parks, 1997.
- RFC4288: RFC 4288—Media Type Specifications and Registration Procedures, N. Freed and J. Klensin, 2005.
- RFC4289: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures, N. Freed and J. Klensin, 2005.

Don't run screaming yet; wait for another paragraph or two. Let me give you the four-sentence summary of these documents necessary to understand the rest of the column:

MIME messages are composed of "parts" surrounded by separators in the text. Each part is labeled with a type and a subtype to help the mail client understand what kind of part it is (is it a picture? a movie? a pdf file? etc.). If a part contains information that can't be represented in ASCII (e.g., a .jpg or a .pdf file), it is encoded into a format that can be represented that way. Oh, and for extra special fun, parts of a MIME message can contain other parts (e.g., you forward someone an entire mail message as an attachment that itself had attachments).

This doesn't sound so bad until you start to read the specs and realize that there are so many edge cases and places where different people can read the specs in different ways. Add on top of that the beautiful agony of HTML messages (MIME messages with HTML parts that a mail reader is expected to display instead of plain text) and you'll quickly realize why many a mail client author has gone barking mad.

Hopefully I've scared you away with this preface so I can simply copy Lorem Ipsum text into the rest of the column instead of having to write more about it. On the off-chance you are still reading, let's take a look at the first task.

How Do You Send Email with Attachments?

Since misery loves company, it makes sense to first figure out how we can spread the MIME joy. It didn't used to be this way, but people who program in Perl are now in the (enviable?) position of having quite a few good modules for sending email messages with attachments. They fall into roughly two categories: those that let you play Vulcan and forge the bits yourself and those that make as many decisions as possible for you so you can wave an impatient "just do it" hand. We'll look at examples from both categories. I should mention that the choice of module for each category is based on my experience, but I would encourage you to look at the other possible choices. For example, Mail::Sender, MIME::Lite, MailBox, Mail::Builder, and MIME-tools all have things going for them that make them worth your consideration.

I've become fond of the mail-related modules from the Perl Email Project (http://emailproject.perl.org/), so the example from the do-it-yourself category is Email::MIME::Creator with some help from Email::Send to actually send the message. Email::MIME::Creator can be considered a bit of an add-on to the more general-purpose Email::MIME package. It is designed specifically for the creation of new MIME messages. Let's see a very simple example of it in action. The first part of the procedure is to load our modules and then create the MIME parts we'll need for the message. In this case we'll need a MIME part for the body of the message and another for the picture we're going to attach:

```
use Email::Simple;
use Email::MIME::Creator;
use File::Slurp qw(slurp);
use Email::Send;

my @mimeparts = (
  Email::MIME->create(
    attributes => {
      content_type => 'text/plain',
      charset => 'US-ASCII',
    },
    body => 'Sending mail from ;login: magazine.',
  ),
  Email::MIME->create(
    attributes => {
      filename => 'picture.gif',
      content_type => 'image/gif',
      encoding => 'base64',
            name => 'picture.gif',
        },
        body => scalar slurp('picture.gif'),
    ),
);
```

Each call to `Email::MIME->create()` creates a new MIME part. We need to specify the MIME attributes and the data for each part. In the second call above we use File::Slurp to easily set the body field to the contents of the GIF file being sent.

Once we have the parts made, we create the actual mail message, this time with an `Email::MIME->create()` call that includes the headers for that message and an anonymous array with pointers to the MIME parts created in the previous step:

```
my $message = Email::MIME->create(
    header => [
        From   => 'loginauthor@usenix.org',
        To => 'dnb@usenix.org',
        Subject => 'Email::MIME::Creator demonstration',
    ],
    parts => [@mimeparts],
);
```

Email message in hand, we can now use Email::Send to actually send it out by handing it to the MTA on our machine:

```
my $sender = Email::Send->new({mailer => 'Sendmail'});
$Email::Send::Sendmail::SENDMAIL = '/usr/lib/sendmail';
$sender->send($message) or die "Unable to send message!\n";
```

And away it goes . . .

If you don't want to bother putting together a MIME message part by part, there are modules such as Email::Stuff available. The name sounds pretty casual, and so is the module. It lets you send the same kind of message with just a single line:

```
use Email::Stuff;

# to give the underlying module a clue where to find the MTA binary
$Email::Send::Sendmail::SENDMAIL = '/usr/lib/sendmail';

Email::Stuff->from('loginauthor@usenix.org)
    ->to('dnb@usenix.org')
    ->text_body('Sending mail from ;login magazine.')
    ->attach_file('picture.gif')
        ->send;
```

And away this message goes . . . Is this easier to use? Sure. But it doesn't give you the same level of control as Email::MIME::Creator. This may or may not be important to you.

How Do You Deal with Attachments You've Received?

Now that I've shown you how to send email with attachments, all of the cool kids will want to do it too, and soon you'll be awash in a sea of messages. Your first idea might be, "Hey, these attachments must all be valuable, how do I save them all?" To indulge this naive response, we could use one of the generic MIME processing packages such as Email::MIME or MIME-tools, but here too there is at least one module that is precisely targeted to the task: Email::MIME::Attachment::Stripper. Here's some sample code:

```
use Email::MIME;
use Email::MIME::Attachment::Stripper;
use File::Slurp qw(slurp write_file);

my $m = Email::MIME->new( scalar slurp 'message.eml' );
my $s = Email::MIME::Attachment::Stripper->new( $m, \
        'force_filename' => 1 );

foreach my $attachment ( $s->attachments ) {
   write_file( $attachment->{filename},
      { buf_ref => \$attachment->{payload} } )
   or die "Can't write $attachment->{filename}: $!\n";
}
```

After loading the modules we parse the email message into an Email::MIME object. This isn't strictly necessary, because Email::MIME::Attachment::Stripper can take other formats if the right module is in place, but I prefer not to leave that to chance. The Email::MIME object is then fed to Email::MIME::Attachment::Stripper so it can do its stuff. We give that call an extra argument of `force_filename` because we want the module to either extract a filename from the message or, if the sending client was sloppy and didn't include one, to make it up. This is important because the next few lines of code expect to be able to write to a file with a specific name.

We then ask Email::MIME::Attachment::Stripper to provide a list of the attachments it has found. It returns a list of hashes, with each hash containing both the information about that attachment and the actual data sent. This payload gets handed off to `File::Slurp's write_file` method and a file is created with that information in it.

One quick related aside: The code is very trusting. It takes the name of the file from data someone else has sent us and is happy to create files with whatever name it is handed. In general you should make your code a little less sanguine and have it only allow filenames that pass some sort of vetting process.

Now, what if you quickly tired of receiving copies of cutepuppy.jpg but wanted to keep the messages sans their attachments? Email::MIME::Attachment::Stripper has a `message()` method that will hand you back an Email::MIME message object that represents the original message without any of the attachments. That's one way of going about the task, but I want to show you another, perhaps more interesting method that gets us a more sophisticated result.

If you've ever had to deal with spam from a mail administrator's perspective, it is entirely likely that you've crossed paths with the open source package Apache SpamAssassin (http://spamassassin.apache.org/). The basis of this package is a set of Perl modules called Mail::SpamAssassin(::*). There are two things that many people don't know about this module set: (1) It contains a really robust MIME parser (because it has to, since spammers throw all sorts of malformed data at it); (2) that parser and some other very handy utility methods can be used for other purposes. If you haven't read the documentation for Mail::SpamAssassin::PerMsgStatus yet, you should.

One of the utility methods related to this column is `get_decoded_stripped_body_text_array()`. This method (according to the doc) "returns the message body, with base-64 or quoted-printable encodings decoded, and non-text parts or non-inline attachments stripped" and "HTML rendered, and with whitespace normalized." If you've ever wanted the "essence" of a message (e.g., for indexing), this method can provide it. It is used like this:

```
use Mail::SpamAssassin;
use File::Slurp qw(slurp);

my $sa = Mail::SpamAssassin->new();

# check_message_text actually attempts to determine the text is spam
my $status = $sa->check_message_text( scalar slurp 'message.eml' );

# but we need its status response, with which we can:
my $body = $status->get_decoded_stripped_body_text_array();

print @{$body};
```

How Do You Know If an Attachment Is, Umm, Icky?

I suppose there is both a Buddhist and a technical answer to this question. Let me try to address the latter. Attachments get a bum rap because they are a major vector for viruses and other malware.

If you want to determine whether an attachment is unsavory, you usually need to pass it through some other package that tries through a variety of methods to determine whether it is unpleasant or not. In the open source world, one very popular package is ClamAV (www.clamav.net). There are a few Perl packages that are designed to use the ClamAV engine. I'd like to mention three of them because they each take a different approach, only one of which may be appropriate for your needs:

- ClamAV::Client talks to a running ClamAV daemon for its scans. This works great if you already are using ClamAV in some fashion or would like something higher-performing than the next module.
- Mail::ClamAV is a Perl wrapper around the ClamAV scanning library API. This is good if you want your Perl code to actually perform the scan or need fine-grained control over the options used for a scan. If your code is going to scan a file and quit each time it is run it is probably going to be less efficient than ClamAV::Client because it needs to spin up the ClamAV engine and load the virus database each time. If your code performs the ClamAV initialization work and then begins to scan lots of files at a time, it is probably a wash.
- File::VirusScan is a "unified interface for virus scanning of files/directories," according to the documentation. It is a wrapper around quite a few (13 as of this writing) commercial and open source anti-virus packages. The same Perl code can scan a file or directory using one or even several anti-virus engines (in order). This is useful if you plan to use a battery of anti-virus checkers or if you'd like to write code that isn't tied to any one of their APIs.

Here's an example of using ClamAV::Client to check the attachments we get back from Email::MIME::Attachment::Stripper:

```
use Email::MIME;
use Email::MIME::Attachment::Stripper;
use File::Slurp qw(slurp);
use ClamAV::Client;

my $msg = Email::MIME->new( scalar slurp 'message.eml' );
my $strip =
  Email::MIME::Attachment::Stripper->new( $msg, 'force_filename' => 1 );
```

```
# we can also connect to a ClamAV clamd listening over a TCP socket if we

# use different options
my $scan = ClamAV::Client->new( socket_name => '/var/run/clamd-socket' );

die "unable to talk to ClamAV daemon"
   unless defined $scan and $scan->ping();

my $ans; # the results back from ClamAV
foreach my $attach ( $strip->attachments ) {
   $ans = $scan->scan_scalar( \$attach->{payload} );
   ( defined $ans )
       ? print "$attach->{filename} is infected with $ans!\n"
       : print "$attach->{filename} is clean.\n";
}
```

The Email::MIME::Attachment::Stripper lines of this code should be familiar from the previous examples. The key new thing we added was the call to ClamAV::Client's `scan_scalar` that passes the contents of a scalar pointed to by a scalar ref off to a ClamAV daemon for processing. If the answer it gets back is anything but `undef` (clean), the code prints the name of the infection as identified by the daemon.

Now that you know a few ways to evaluate your attachments, it's time to end the column. Take care, and I'll see you next time.