Creating portable archives of favorite Perl modules

Pack Your Bags

Every developer uses a personal collection of scripts for recurring jobs. This month, I will be introducing you to four scripts from my own collection. All of them require Perl modules that are not available by default on most machines.

BY MICHAEL SCHILLI

The Perl Archive Toolkit resolves dependencies and puts scripts and modules into portable archives.

Developers who work on different machines will be familiar with this problem: Perl scripts tend to need a whole bunch of modules. Installing the modules means configuring the local CPAN module and takes time. Autrijus Tang's PAR (Perl Archive Toolkit) CPAN module provides an approach to packaging scripts – along with all the modules they require – in a practical archive, in a similar way to Java with *.jar* files.

Working on the assumption that it will not be installed on most systems, PAR can create executables to provide a surefire way of unpacking its archives. An executable includes an interpreter, all the modules you need, plus libraries and scripts. This means that users can run Perl scripts where modules would be missing under normal circumstances.

Figure 1: A typical browser window during basic authentication. Although the browser filled the password field automatically, the password is obfuscated.

PAR is extremely useful for developers who normally carry a wagon load of scripts around with them. Scripts that you need every day to handle those recurring tasks tend to use modules from the seemingly infinite CPAN collection. The four scripts we will be looking at are from my own personal toolkit.

Password Sniffer

One tool that I always find useful is a compact Base 64 encoder/decoder, if you use basic authentication on the Web. Just recently, I was confronted with a dialog like the one shown in Figure 1, and couldn't remember the password. Fortunately, the browser had a password manager that would automatically fill out the fields. The bad news was that the password field was obfuscated.

A proxy between the browser and the server soon told me that the browser was sending a string, such as *dGVzdDpzWNyZXQ=* (see Figure 2). Unfortunately, both the browser password and the username were Base 64 encoded. Thanks to Perl, decoding this was simple, and the script in Listing 1 soon gave me the clear text:

Listing 1: *b64.pl*

```
01 #!/usr/bin/perl
02 ##################################
03 # b64.pl - En/Decode Base64
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ##################################
07 use warnings;
08 use strict;
09
10 use Getopt::Std;
11 use MIME::Base64;
12
13 getopts "d", \my %opts;
14
15 die "usage: $0 [-d] string"
16   unless defined $ARGV[0];
17
18 if($opts{d}) {
19     print decode_base64(
20           $ARGV[0]), "\n";
21 } else {
22     print encode_base64(
23           $ARGV[0]);
24 }
```

```
$ b64.pl -d ➥
dGVzdDpzZWNyZXQ=
test:secret
```

The username is *test*, and the password *secret*. This also goes to show that Basic authentication is fairly useless if an attacker can sniff the authentication exchange. The following example shows that *b64.pl* is also capable of encoding:

```
$ b64.pl test:secret
dGVzdDpzZWNyZXQ=
```

There are many more uses you could put the script to: Base 64 encoding is the method of choice when binary data needs to be converted to a format suitable for displaying. For example, many email programs use Base 64 to encode binary attachments.

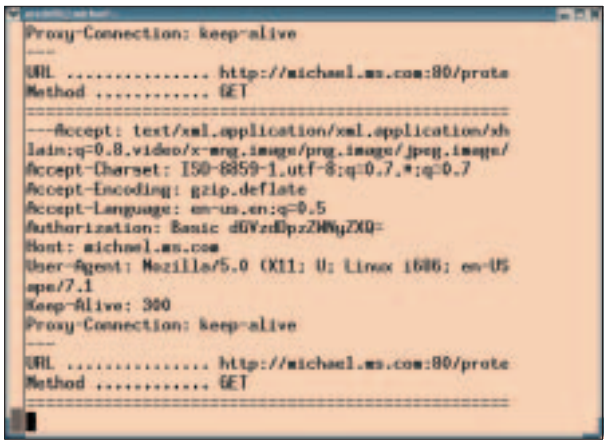

Figure 2: The Base 64 encoded username/password string from the Basic authentication dialog between the Web client and the server is easy prey for a sniffer running on a proxy.

Percent-Hex for URL code

Many URLs also include encoded sequences, although in an entirely different format. When a browser uses a GET request to send a parameter to a server, setting the query string parameter *p* to the value *a b/c* the URL looks like this:

```
http://host.com/cgi/foo?p=➥
a%20b%2Fc
```

The browser has converted the special characters into a *%XX* style format, where *XX* is the hexadecimal value of the corresponding ASCII code. Listing 2, *urlcode.pl* lets you decode the hex string:

```
$ urlcode.pl -d a%20b%2Fc
a b/c
```

urlcode.pl also handles encoding. The following syntax encodes a string for test purposes to URL format:

```
$ urlcode.pl "a b/c"
a%20b%2Fc
```

The script is trivial, thanks to the CPAN *URI::Escape* module; it simply uses *Getopt::Std* to test if *-d* is present on the command line, and then calls *uri_escape* or *uri_unescape* from *URI::Escape* as appropriate.

Atari Hexdump

Binary files fill up the screen with special characters, and can even cause a terminal to hang, if an unsuspecting user calls *cat* to display them. *less* is better, but it won't help you much as it does not tell you how long specific byte sequences are. The CPAN *Data::Hexdumper* module gives you a display format like the one I had on my Atari ST1040 15 years ago with hexcode on the left in groups of 16 bytes, and printable characters on the right – if equivalent printable characters exist. Figure 3 shows *hd.pl* from Listing 3 displaying its own source code as a hexdump on screen.

The *hd.pl* program may be a painfully trivial adaption of the *Data::Hexdumper* manpage, but even a simple tool like this can save you a lot of time.

Perl Switcheroo

If you do a lot of work with Perl, can hardly wait to get your fingers on the latest distribution, or simply want to know if a module will run with an ancient version such as 5.00503, you need an option for jumping back and forth between various installations. You should avoid installing your Perl distributions below */usr* and instead use a dedicated directory, below your home directory for example. In this case, you would use the following syntax to call configure for Perl 5.8.4,:

```
./Configure -D prefix=$HOME/➥
perl-installs/perl-5.8.4 -d
```

Now, when you call *make install*, the distribution will be installed below the specified directory. To switch between versions, you need to create a symbolic link called *perl-current* in *$HOME/perl-installs*, and point the symlink at the

Listing 2: *urlcode.pl*

```
01 #!/usr/bin/perl
02 ##################################
03 # urlcode.pl: URLen/decode
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ##################################
07 use warnings;
08 use strict;
09
10 use Getopt::Std;
11 use URI::Escape;
12
13 getopts "d", \my %opts;
14
15 die "usage: $0 [-d] string"
16   unless defined $ARGV[0];
17
18 if($opts{d}) {
19     print uri_unescape(
20           $ARGV[0]), "\n";
21 } else {
22     print uri_escape(
23           $ARGV[0]), "\n";
24 }
```

Listing 3: *hd.pl*

```
01 #!/usr/bin/perl
02 ##################################
03 # hd.pl-Hexdump
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ##################################
07 use strict;
08 use warnings;
09
10 use Data::Hexdumper;
11
12 my $data = join '', <>;
13
14 my$results=hexdump(
15     data          => $data,
16     number_format => 'C',
17 );
18
19 print$results;
```

currently active version, conveniently installed in the *perl-installs* directory, *perl-5.8.4* in the example. If */usr/bin/perl* is then deleted and replaced by a link to *$HOME/perl-installs/perl-current/bin/perl*, scripts with a Shebang line that reads *#!/usr/bin/perl* will automatically find your current choice of Perl installation.

Of course, you will also need to redirect any other perl-related programs or scripts under */usr/bin/*, *perldoc* for example. After completing these preparatory steps, you are ready to run the *sp.pl* (for Switch Perl) script shown in Listing 4. As Figure 4 shows, the script gives you a selection of versions installed below *$HOME/perl-installs*, before going on to set the *perl-current* symbolic link to reflect your choice. This makes switching a breeze!

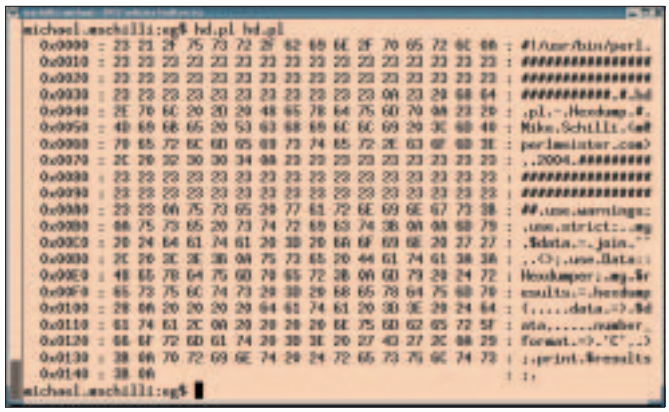

Figure 3: If you need to view binary data, check out the *Data::Hexdumper* Perl module. Here, we used the *hd.pl* script to view its own source code.

Pack Your Bags with PAR

Now, all of the tools we have looked at so far need a few additional modules on your machine: *b64.pl* uses *MIME::Base64*, and *urlcode.pl URI::Escape*. Neither of these modules are part of the plain vanilla Perl distribution, and it is safe to assume that many machines on which you want to run the scripts will not have the modules installed.

To create an archive with all four of the scripts discussed in this article, you simply need to call *pp* from the CPAN PAR distribution:

```
pp -output=toolbox.exe b64.pl ➥
urlcode.pl hd.pl sp.pl
```

This syntax creates the *toolbox.exe* binary, which includes all four scripts and the required modules. Despite the *.exe* handle, the binary will have the right format for the operating system on which you run *pp*. Assuming that the target machine has the same operating system platform, it is very simple to install the toolbox. The following few lines below, install everything you need in the user's *~/bin/toolbox* directory:

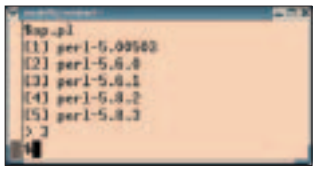

Figure 4: Switch between various Perl versions installed on your machine.

```
mkdir ➥
~/bin/toolbox
cp toolbox.exe ~/bin/toolbox
cd ~/bin/toolbox
for i in b64 urlcode hd sp ; do
    ln -s toolbox.exe $i
done
export ➥
PATH=$PATH:~/bin/toolbox
```

The newly created symbolic links *b64*, *urlcode*, *hd*, and *sp* (note the missing *.pl* extensions) all point to *toolbox.exe*. If the user now calls *b64*, *toolbox.exe* will start and automagically detect that the packed script *b64.pl* needs to be invoked. It extracts *b64.pl* from the archive, loads the required modules (also from the archive), and runs the script.

PAR can support multiple platforms. To leverage this feature, you need specify the *--multiarch* parameter, and run the *pp* tool on each target platform to add the required files to the archive. Of course, you can not create an executable in this way, as each operating system has a different format. To unpack a multiarch archive, users need to install the PAR module on their machines. The Tutorial, *PAR::Tutorial included with PAR*, has more tips on using the tool. There is one trap to look out for, however: developers should use the oldest machine they have to build the PAR archive, thus avoiding any potential problems that could occur with libc backward compatibility issues. ■

Listing 4: *sp.pl*

```
01 #!/usr/bin/perl
02 ##################################
03 # sp.pl - Select perl install
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ##################################
07 use strict;
08 use warnings;
09
10 use File::Basename
11                 qw(basename);
12
13 my $PERL_HOME = "$ENV{HOME}".
14             "/perl-installs";
15
16 my(@versions, $count);
17
18 for (<$PERL_HOME/perl-*>) {
19     next if -l or ! -d;
20     push @versions,
21          basename($_);
22 }
23
24 foreach my $v (@versions) {
25     print "[", ++$count,
26           "] $v\n";
27 }
28
29 $| = 1;
30 print "> ";
31 my $number = <>;
32 chomp $number;
33
34 die "Invalid choice" unless
35  exists $versions[$number-1];
36
37 unlink("$PERL_HOME/" .
38        "perl-current") or
39    warn "unlink failed ($!)";
40
41 symlink("$PERL_HOME/" .
42   "$versions[$number-1]",
43   "$PERL_HOME/perl-current")
44   or die "symlink ($!)";
```