DAVID N. BLANK-EDELMAN

practical Perl tools: This column is password-protected.

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.edu

PASSWORDS SUCK. I SUPPOSE I COULD say that another way, beat around the bush, use a word that doesn't evoke an act between consenting adults, but let's face it. Passwords suck. I know it, you know it, everyone in our field knows it. And unfortunately, as much as I'd like to tell you about a Perl module that solves the problem (Data::MakePasswordsNotSuck, I suppose), there's no such module that I'm aware of. At best I can show you a number of different Perl modules that make dealing with passwords just a hair better. And that's just what we're going to do in this column. The actual Perl code in this column will be super-simple, because we're going to try and use the tools that make these improvements as easy as possible.

Start with Better Passwords

Many people would agree (four out of five dentists who chew gum, to be precise) that passwords aren't the problem per se; rather, the beef is with *bad* passwords. There are a number of psychological, sociological, and contextual factors for why people pick and keep bad passwords. One important factor is the "blank-page" problem. If someone says to you, "Quick, pick something you'll need to be able to remember, but don't make it something anyone else can guess," that's a lot of pressure. Having posed this question to a new crop of students every year for the last 13 or so years I can assure you that lots of people fail this test. Even the ones with reasonably high SAT scores. They will stare doe-eyed into space for a moment, stick the tip of their tongue out the corner of their mouth, ponder, and then come up with their middle name, or even "password." I've said this in print before, but I'll say it again: I'm fairly certain that Oog's password to get into his cave was probably "Oog." It wasn't until a later era that he changed it to 00g.

You can try to prevent some of this by screening for bad passwords (and we'll see how to do that in our next section), but having someone iterate through all of the bad passwords they know until they find one the system *will* accept isn't a particularly good password-picking algorithm. It might be better to try to provide something at the get-go that is reasonably "secure."

(As an aside, I used snarky quotes here because there's so much more to a secure password implementation beyond just the contents of the passwords. There are tons of important things one has to watch, such as how the password is stored, used, etc. The two words "rainbow tables" are enough to demolish lots of "secure" password schemes.)

There are several Perl modules designed to generate more secure passwords. Some of them create passwords that are truly random. Some of them produce passwords that are close to random, but have the nice property of being pronounceable in someone's native language (and hence perhaps more memorizable). Random passwords are in theory more secure, but there have been some good debates over the years in the security community about whether providing users with something they have to write down on a sticky note is better than something less random that they are more likely to be able to keep in their head.

All of the Perl modules in this space are very easy to use. You ask for a password, the module hands you one. You may have to or want to provide some parameters describing the kind of password you want (or perhaps provide some hints on what "pronounceable" entails in your language), but that's all the thinking you need to do to use them. Let's see a couple of examples. The first prints a random password of 10 characters in length:

```
use Data::SimplePassword;
my $dsp = Data::SimplePassword->new();

# 10 char long random password, we could specify which
# characters to use if we cared via the chars() method
print $dsp->make_password(10),"\n";
```

When I have to generate random passwords I tend to use Crypt::GeneratePassword, because it generates pronounceable passwords that are slightly more secure than those that rely strictly on the NIST standard (FIPS-181) for creating them. Plus it provides the functionality for screening the generated password for naughty words of your choice. One of the hazards of creating pronounceable passwords is that it is possible to generate passwords with character sequences (such as "passwords suck") that might offend those with delicate sensibilities. To use it, we call either the `word()` function for pronounceable passwords or the `chars()` function for purely random passwords. Both functions take two required arguments: the minimum and the maximum length of the password to return. For example, the following code:

```
use Crypt::GeneratePassword;

for (1..5) {
  print Crypt::GeneratePassword::word( 8, 8 ),"\n";
}
```

would print something like this:

```
eictumpu
orastbot
rnbuiltp
meagnell
vilieway
```

I'll stop here on this subject, but I should point out that trying to improve usability issues around authentication is a nontrivial problem. For a good treatise that looks at questions like this see Simson Garfinkel's PhD thesis at http://www.simson.net/thesis/.

Screening Out Bad Passwords

We don't always have the luxury of being the sole source for the passwords our users will use. It's very likely that they will want the opportunity to change it for themselves (although often you can suggest more secure temporary passwords as part of an "I forgot my password" request). In those cases and in the cases where generating more secure passwords is not feasible, it becomes even more important to have a mechanism for screening out bad passwords.

The real trick to this is deciding what constitutes a "bad" password in your environment. Is a password bad if it contains a dictionary word in any common language? (Probably.) If it contains some permutation of the user's personal information such as name or login name? (Yes.) If it is insufficiently random? (Maybe.) If it has ever been used before by this user? (That depends.) If it is palindromic? (Probably.) The Perl password-checking modules available can check for these things (or code can easily be added to do so).

As an aside to give you one measure of comparison, passfilt.dll, the optional "strong password enforcement" library Microsoft ships enforces (to quote their doc) the following rules:

- Passwords may not contain your user name or any part of your full name.
- Passwords must be at least six characters long.
- Passwords must contain elements from three of the four following types of characters: English uppercase letters, English lowercase letters, Westernized Arabic numerals, Non-alphanumeric characters, Unicode characters

That last restriction seems to give people (especially those for whom English is not a first language) a considerable amount of trouble. You'll want to think carefully about your policy when you implement password checking.

There are several Perl modules that cover the password-checking territory. From the list, I'd probably recommend choosing between Data::Password::Check and Crypt::Cracklib. The first is a pure-Perl module that comes with a set of basic tests but allows the programmer to add more at will. The second module I mentioned is my first choice, but it requires the ability to compile and link an external library. Crypt::Cracklib links against Alec Muffett's excellent Cracklib library (the current distribution site of which is http://sourceforge.net/projects/cracklib). According to the README:

> CrackLib makes literally hundreds of tests to determine whether you've chosen a bad password.
>
> - It tries to generate words from your username and gecos entry and match them against what you've chosen.
> - It checks for simplistic patterns.
> - It then tries to reverse-engineer your password into a dictionary word and searches for it in your dictionary.

If you add the boatload of additional dictionaries and wordlists available on the Net and for purchase (e.g., the CD from www.openwall.com for Jack the Ripper) you get a very effective password-checking tool. I've had the pleasure over the years of watching new Indian students in our college be amazed and somewhat frustrated that our system won't let them use Hindi or Urdu words or names in their password.

A demonstration of how to use Crypt::Cracklib is almost embarrassingly simple:

```
use Crypt::Cracklib;

my $result = fascist_check($inputpasswd,'/path/to/your/bighonkin_dictionary');

    if ($result eq "ok"){
        print "Password provided is accepted.\n";
    }
    else {
        print "Can't accept password because it $result.\n";
    }
```

All of the magic takes place in the `fascist_check` call. It will return "ok" if Cracklib deems the password to be ok; otherwise it returns Cracklib's reason for rejecting it (e.g., "contains a dictionary word").

Better Password Input

This is a fairly simple notion, so I'll make our discussion of it really quick. If you are going to be handling the actual input of passwords yourself at a command line (versus having a browser take it in for you), you should do what you can to make the experience safe and pleasant. Not showing the actual password as you type is a notion almost as old as the first password prompt, but people have also come to expect some feedback as they type. Just last week we had a new student come in asking for help because of a problem logging into our Solaris machines in our student lab. She was sure her password wasn't being accepted, because the XDM login session did not show anything at all as she typed on the password line. That's the first time we've seen this issue, but I'm willing to bet it isn't the last. If you choose the right Perl module you can easily head off this sort of naive issue at the password pass.

Again, there are a number of Perl modules we could use. The two special-purpose modules I'd recommend are IO::Prompt and Term::ReadPassword{::Win32}. The former is Damian Conway's module. On the plus side, it does a good job of providing most everything you would need for a general-purpose prompting module (including things such as per-keystroke feedback). On the negative side, IO::Prompt provides most everything you would need, but only under a UNIX-ish system. It doesn't work so well on a Windows system, for example. It also could use better documentation. Here's an example of how you would request a password using it:

```
use IO::Prompt;

my $passwd = prompt "Password: ", -echo => '*';
```

Term::ReadPassword and its Windows equivalent, Term::ReadPassword::Win32, have fewer features (since it isn't meant to be a general-purpose prompting module) but do have the multi-platform reach if that's important to you. Using them is equally easy:

```
use Term::ReadPassword;

# turn on the * for each char typed feature(?)
$Term::ReadPassword::USE_STARS = 1;
my $passwd = read_password('Password: ');
```

Before we head to the last section, I do feel compelled to mention that the UI questions around prompting for a password aren't all straightforward, no matter how windy it got in here because of all of the hand-waving I did in the introductory paragraph. If your code prints an asterisk for every character typed, this gives anyone watching the process a quick idea of the length of the entered password. If you think that's an unacceptable disclosure, you may want to hack the module code to be a bit more circumspect (e.g., display twice as many or a random number of asterisks per keystroke).

One Step in a Better Direction

I didn't want to end a column about passwords that started with gloom and doom without suggesting that there might be other alternatives available or at least on the horizon. If we ignore the work people are doing on more interesting password-like tests (visual passwords constructed with faces or favorite pictures, scratch-and-sniff passwords, etc.) there are still some current-day possibilities for improving the situation. The one I want to mention is Steve Gibson's take on one-time password systems. This is a scheme where you somehow arrange for your system to accept a different password for a user each time that person logs in. As soon as you use a password, it is "used up" and hence not useful to someone trying break into your account. Gibson came up with something he calls (with the usual humility) "Perfect Paper Passwords." Documented at http://www.grc.com/ppp, it is a pretty spiffy (read: usable) system, which he describes like this: "GRC's 'Perfect Paper Passwords' (PPP) system is a straightforward, simple and secure implementation of a paper-based One Time Password (OTP) system." When used in conjunction with an account name and password, the individual "passcodes"

contained on PPP's "passcards" serve as the second factor ("something you have") of a secure multi-factor authentication system.

The system allows you to generate little paper passcards for your users to print out that contain a sequence of one-time codes. It's a nice way to provide this sort of security on the cheap. It's not actually perfect in all situations (see the paper by A. Wiesmaier et al. [1] as a start for more details) but it may give you a little more peace of mind.

The Perl tie-in to PPP is the module Crypt::PerfectPaperPasswords, which can generate passcodes and passcards for the system. I haven't seen a Perl-only implementation of the server that accepts these passcodes, but having a generator you can use from your Perl programs (e.g., a Web app) could be useful.

And with that refrigerator lightbulb ray of hope, we have to bring this column to an end. Take care, and I'll see you next time.

REFERENCE

[1] A. Wiesmaier, M. Fischer, M. Lippert, and J. Buchmann, "Outflanking and Securely Using the PIN/TAN-System," *Proceedings of the 2005 International Conference on Security and Management (SAM '05)*, June 2005.