The journal of Apple technology.

Volume Number: 18 (2002)
Issue Number: 9
Column Tag: Mac OS X

ckpath

Analyze a file's permissions, using Perl

by Rich Morin

In order to know who can do what to a file, you have to understand the permissions on the file itself and on each directory leading to it. Locking down write permission on a file, for instance, keeps miscreants from writing into the file, but it doesn't keep them from removing and replacing it. To prevent that, you have to set the right permissions on the enclosing directory.

Or, let's say that your file path contains some symbolic links. In order to reach the file, a program must traverse the path up to the symlink, then backtrack and traverse the path up to the symlink's target. If the path is /A/B/C and B is a symlink to /X/Y, the program will need access to /, /A, /A/B, / (again), /X, /X/Y, and /X/Y/C.

The BSD command "ls -dl" will show the permissions on a specified file or directory, but typing in a long sequence of commands is both tedious and error-prone. Consider:

```
% ls -ld /
drwxrwxr-t  49 root  admin  1622 Jul 29 11:11 /
% ls -ld /Applications
drwxrwxr-x  36 root  admin  1180 Jul 28 10:34 /Applications
...
```

Fortunately, it's quite possible to automate this procedure. My ckpath script examines each element in the requested file path, back-tracking as necessary to handle symbolic links. It handles "white space" in file names (uncommon in BSD, but common in Mac OS X) and fiddles a bit with the output format.. Here's some sample output:

```
% ckpath "/Applications/AppleScript/Example Scripts"
"/Applications/AppleScript/Example Scripts"
1775 drwxrwxr-t  49 root  admin  2002.07.29 /
0775 drwxrwxr-x  36 root  admin  2002.07.28 Applications
0775 drwxrwxr-x   5 root  admin  2002.02.14 AppleScript
0775 lrwxrwxr-x   1 root  admin  2002.02.14
    "Example Scripts" -> /Library/Scripts
/Library/Scripts
1775 drwxrwxr-t  49 root  admin  2002.07.29 /
0775 drwxrwxr-x  28 root  admin  2002.07.16 Library
0775 drwxrwxr-x  12 root  admin  2001.09.14 Scripts
```

The first two output fields (e.g., 0775 and drwxrwxr-x) contain the octal and symbolic representations of the node's permissions. For a complete explanation of BSD permission codes, see the ls(1) manual page. Briefly, however, the

story is that each entity in the file system has a type (e.g., directory, file, symlink) and three sets of permissions bits (for user, group, and other). Some ancillary bits control special features such as set[ug]id execution.

A string such as drwxrwxr-x indicates that this is a directory and that anyone can read and execute (pass through) it. Any "other" user (not the owner, nor in the directory's group) cannot write (i.e., create, remove, or rename files) in the directory.

The following three fields (links, owner, and group) are taken directly from the ls output. The date has been normalized into YY.MM.DD format, improving line-to-line consistency and easing date calculations. The remainder of the line contains the node name, quoted if it contains spaces. As in ls output, symlinks are listed with their targets.

Code Walkthrough

This walkthrough is neither an attempt to teach Perl in one sitting, nor a truly detailed explanation of the intricacies of ckpath. Instead, it touches on both language and design issues, trying to hit some of the high points of each. The references listed in this month's "Section 7" column can help you with the Perl issues; I hope to explain the program's general flow in the following text.

The first line of ckpath allows for the possibility that we may have installed a copy of the Perl interpreter in a non-standard location. /usr/bin/env walks down our search path, finding the same copy of Perl that the shell would.

If ckpath is run with no argument, it examines the current working directory. Otherwise, it uses the argument as a path name, prepending the current working directory unless the path begins with a slash. This is fairly traditional behavior for a BSD command.

Some advocates of structured programming entirely refuse to use gotos. I avoid them in general, but use them (as in this case) when the alternative would be even uglier. Interested readers are invited to attempt a goto-free formulation.

After tidying up the incoming path name, we print it out for the user (in quotes, if it contains any white space). We then create a "todo" list, containing the full path names for each node in the input path name. This is the putative task list, but it may be abandoned if we encounter a symlink or an error.

After formatting the node name and determining that the node actually exists, we examine it in two ways. First, we run "ls -ald", discarding everything but the symbolic permission information. We then use lstat to retrieve the rest of the information we want.

This isn't particularly elegant or efficient, but it's a lot easier than generating the symbolic permission codes ourselves or, worse, trying to parse the output of ls. Interested readers, again, are welcome to try coding alternative approaches.

Using getpwuid and getgrgid, we try for symbolic versions of the user and group names, falling back to numeric forms if need be. localtime gives us a printable list of time values, from which we grab the year, month, and day.

If the node is a symlink, we add the target to the output line, fudge the path name to reflect the symlink's target, and jump back to REDO. Otherwise, we simply print a closing newline and go back for the next node.

Observations

Perl is particularly facile at handling this sort of problem. It has good string-handling capabilities, powerful and convenient data structures, and access to assorted system calls and library functions. I can't see doing this program as a shell script; the shell isn't powerful enough. Nor would I want to try writing it in C (no string-handling, regular expressions, etc.).

The strict and warnings pragmas are a bit like using lint(1) on C code. They tell Perl to look for all sorts of incipient problems, such as variables which are only used once. I've started using these more frequently than I once did, partly as a consequence of writing larger scripts where the scope of variables can become a real issue. The extra typing (and, occasionally, redesign) that the pragmas require seems to be more than compensated by the problems they uncover.

```
CKPATH SOURCE CODE
#!/usr/bin/env perl
#
# Usage: ckpath [file node]             # defaults to .
#
# Rationale:
#
# Let's say that you have a file which is having permissions
# problems.  In order to find out ALL the relevant
# permissions, you will have to run "ls -ld" on each element
# of the path, then back-track for each symbolic link you
# encounter.  Not fun.  This script automates the process,
# allowing you to see the entire path's permissions at once.
# It also tweaks the output format a bit (e.g., printing the
# octal modes and making the date format consistent).
#
# Written by Rich Morin, CFCL, 2002.06
use strict;
use warnings;
{
  my(@stat, @todo,
     $cwd, $grp, $mday, $mode, $mon, $name, $node,
     $save, $sm, $tgt, $tmp, $todo, $usr, $year
  );
  $cwd = `pwd`; chomp($cwd);
  if ($#ARGV == -1) {             # Get path, if any.
    $todo =  $cwd;
  } else {
    $todo = $ARGV[0];
    $todo = "$cwd/$todo" if ($todo !~ m|^/|);
  }
REDO:
  $todo =~ s|/[^/]+/\.\./|/|g;    # "/foo/../" -> "/"
  $todo =~ s|/\./|/|g;            # "/./"      -> "/"
  $todo =~ s|//+|/|g;             # "//"       -> "/"
  $todo =~ s|/$||;                # ".../foo/" -> ".../foo"
  $save = $tmp = $todo;           # Print current task.
  $tmp = "\"$tmp\"" if ($tmp =~ m|\s|);
  print "\n$tmp\n";
  undef @todo;                    # Get list of nodes.
  while ($todo ne '') {
    push(@todo, $todo);
    $todo =~ s|/[^/]+$||;
  }
```

```
  push(@todo, '/');
  while ($name = pop(@todo)) {     # Print info on node.
                                   # Format node name.
    ($node = $name) =~ s|^.*/([^/]+)$|$1|;
    $node = "\"$node\"" if ($node =~ m|\s|);
    if (! -e $name) {
      printf("%-48s %s\n",
        'Warning!  No such file or directory:', $node);
      last;
    }
                                   # Protect white space.
    ($tmp = $name) =~ s|(\s)|\\$1|g;
                                   # Get symbolic mode info.
    $sm   = substr(`ls -ald $tmp`, 0, 10);
                                   # Get info on node.
    @stat = lstat($name);
                                   # Get numeric mode info.
    $mode = $stat[2] &  07777;
                                   # Get user name.
    $usr  = (getpwuid($stat[4]))[0];
    $usr  = $stat[4] if ($usr eq '');
                                   # Get group name.
    $grp  = (getgrgid($stat[5]))[0];
    $grp  = $stat[5] if ($grp eq '');
                                   # Get modification time.
    (undef, undef, undef, $mday, $mon, $year,
     undef, undef, undef) = localtime($stat[9]);
    printf("%04o %10s %3d %-8s %-8s %s.%02d.%02d %s",
      $mode, $sm, $stat[3], $usr, $grp,
      $year+1900, $mon+1, $mday, $node);
    if ($sm =~ m|^l|) {            # Eeek, a symbolic link!
      $tmp = $tgt = readlink($name);
      $tmp = "\"$tmp\"" if ($tmp =~ m|\s|);
      printf(" -> %s\n", $tmp);
      ($todo = $save) =~ s|^$name|$tgt|;
      if ($tmp !~ m|^/|) {
        ($tmp  = $name) =~ s|^(.*/)[^/]+$|$1|;
        $todo  = "$tmp$todo";
      }
      goto REDO;
    }
    print("\n");
} }
```

Rich Morin has been using computers since 1970, Unix since 1983, and Mac-based Unix since 1986 (when he helped Apple create A/UX 1.0). When he isn't writing this column, Rich runs Prime Time Freeware (www.ptf.com), a publisher of books and CD-ROMs for the Free and Open Source software community. Feel free to write to Rich at rdm@ptf.com.