Introduction to Shell and Perl Scripting for Network Operators

John Kristoff jtk@cymru.com

overview

- We'll assume basic UNIX sysadmin competency
- We'll introduce shell and Perl syntax and concepts
- We'll learn to read and write “combat” scripts and tools
- We'll try to write portable scripts
- We'll use real world examples you can build upon
- We'll assume you'll be hacking while I talk
- We'll not be exhaustive since our time is limited
- We'll encourage you to write and share more tools
- Get this: http://www.cymru.com/jtk/code/nanog54.tar.gz

what shell scripting is and is not

- Programmatic interface to shell's built-in capabilities
- Existing system tools, utils and apps can be called
- Generally applied to sysadmin-related tasks
- Trade-offs in performance, ease-of-use, features
- Also see:
  - Portable Shell Programming, Blinn - Prentice Hall
  - Classic Shell Scripting, Nelson & Beebe - O'Reilly

shell scripting by example

- genpass.sh – pseudo-random 1-32 hex pass generator

genpass.sh [1/5]

```
$ chmod +x genpass.sh
$ ./genpass.sh
e5fe17c85c686060
$ ./genpass.sh
815dd6fc61f18671
$ ./genpass.sh
adeea2e87e2a961c
$ ./genpass.sh 8
98dad45b
```

genpass.sh [2/5]

```
#!/bin/sh

...

head /dev/urandom |
    ${MD5} | cut -c1-${LEN} -
```

genpass.sh [3/5]

```
#!/bin/sh

...

if test -z "${MD5}"
then
    echo no MD5 tool found, exiting...
    exit 1
fi
head /dev/urandom |
    ${MD5} | cut -c1-${LEN} -
```

genpass.sh [3/5] alternate

```
#!/bin/sh

...

if [ x“${MD5}” = x ]
then
    echo 'no MD5 tool found, exiting...'
    exit 1
fi
head /dev/urandom |
    ${MD5} | cut -c1-${LEN} -
```

genpass.sh [4/5]

```
#!/bin/sh

...

LEN=${1:-16}
MD5=$(get_md5_tool)
if test -z ${MD5}
then
    echo no MD5 tool found, exiting...
    exit 1
fi
head /dev/urandom |
    ${MD5} | cut -c1-${LEN} -
```

genpass.sh [5/5]

```
get_md5_tool() {
    # Linux
    command -v md5sum > /dev/null 2>&1↩
        && echo md5sum && return
    # BSD
    command -v md5 > /dev/null 2>&1↩
        && echo md5 && return
    # OpenSSL
    command -v openssl > /dev/null 2>&1↩
        && echo openssl md5 && return
}
```

hash-bang (aka shebang) #!

- Left-justified, first line in all self-contained scripts
- String to follow and parameters run by the OS
  - `#!/bin/sh`
  - `#!/usr/bin/perl -T`
- Sometimes you see this, but probably best to avoid
  - ~~`#!/usr/bin/env perl`~~
- Also see:
  http://www.in-ulm.de/~mascheck/various/shebang/
  http://en.wikipedia.org/wiki/Shebang_(Unix)

comments

- An unquoted hash (#) signals comments to follow
- Comments run to the end of the line
- Comments can follow commands

```
#!/bin/sh
TC=whois.cymru.com
dig -x $1 +short          # DNS
whois -h $TC " -f $1"     # ASN mapping
```

pipelines

- A pipe (|) ties two or more processes together
- STDOUT of predecessor is STDIN of successor
- STDERR messages generally merged to console
- All processes in a pipe run concurrently
  - `sort words.txt | uniq | wc -l`
- Also see:

  http://en.wikipedia.org/wiki/Pipeline_(Unix)

redirection

- Redirect STDIN with <
  - `psql -d db -c \`
    `“copy routes from stdin” < rib.dat`
- Redirect STDOUT with >
  - `strings suspect.exe > readable.txt`
- Redirect and append STDOUT with >>
  - `checkdb.sh >> /var/log/db.log`

more redirection

- 0,1,2 corresponds to STDIN, STDOUT, STDERR
- Often want to redirect STDOUT and STDERR

```
#!/bin/sh
nc -z $1 80 > /dev/null 2>&1
if [ $? -eq 0 ]
then
    echo [$1]:80 TCP up
else
    echo [$1]:80 TCP down
fi
```

here document

- Use << or <<- to remove leading tabs

```
#!/bin/sh
sendmail -t << MAIL_EOF
To: noc@example.net
From: code@combat.example.net
Subject: ### cisco Traceback messages

`grep Traceback /var/log/cisco.log`
MAIL_EOF
```

variables

- Used to store a string value
- First character must be a letter or underscore
- Assignment is simply name, equal sign, value
  - **email=**`jtk@cymru.com`
  - **email2=**`jtk@depaul.edu`
- Usage has dollar sign prefix, optional braces
  - `echo` **$email**
  - `echo` **$email2**
  - `echo` **${email}**`2`

special variables

- **`$?`** - exit status of the last command
- **`$$`** - process id of the current command
- **`$!`** - process id of the last command executed
- **`$0`**, **`$1`**, **`$2`**, …, $n – positional parameters
- **`$#`** - current number of parameters available
- **`$*`** - parameter list, single value when quoted
- **`$@`** - parameter list, separate values when quoted

conditionals

- ORed execution
  - `mkdir foo || exit 1`
- ANDed execution
  - `./configure && make`
- controlled flow with the `if` statement

```
if [ $# -ne 1 ]
then
        echo Usage: $0 input_file
        exit 1
fi
```

if then ... elif ... else ...

```
if [ x"$qtype" = AAAA ]; then
    dig aaaa $qname

elif [ x"$qtype" = PTR ]; then
    dig -x $qname

elif [ x"$qtype" = NS ]; then
    dig ns $qname

else
    dig $qname

fi
```

case

- Execute a set of commands for a pattern match

```
case $mode in
    -init)      rrdtool create foo.rrd \
                DS:bar:GAUGE
                ;;
    -update)    rrdtool update foo.rrd \
                bar:$1
                ;;
    *)          usage
                ;;
esac
```

loops

- Execute a set of commands for each value in a list

```
for file in `ls *.gz`
do
    gunzip $file
done
```

- Execute a set of commands while condition is true

```
while :
do
    ping $1
    sleep 30
done
```

functions

- Command set run when function name is invoked
- Functions must be defined before called

```
next_7_days() {
   nums="1 2 3 4 5 6 7"
   for x in $nums
   do
      DATE=`date -d "+$x days" +%Y%m%d`
      DATES="$DATE $DATES"
   done
}
...
next_7_days
```

miscellaneous

- RCS
- set options
- Quoting
- Sub-shells
- Line continuation (\)
- Command separation (;)

an aside: using crontab

- Default crontab shell may not be what you expect
- STDOUT/STDERR may result in noisy emails
- Careful of a prior cron job that hasn't finished

```
# abort if we are already running
if test -r $PIDFILE
then
    PID=`cat $PIDFILE`
    if [“$(ps -p $PID|wc -l)” -gt 1]
    then
        exit 1
    fi
fi
```

combat shell script toolbox

- drgenpass – another, better password generator
- initbind – skeleton init.d ISC BIND named start-up
- pcapr – rotating libpcap on a capture host
- devinfo – summary switch/router device via SNMP
- lostacls – identify unused ACLs on a Cisco
- qwikrrd – rudimentary graphing starter scripts

whirlwind tour of Perl

- Creating your first script
- Scalars, arrays and hashes
- Control structures
- I/O operations
- Regular expressions
- Subroutines and modules
- Sorting
- Miscellaneous

what Perl scripting is and is not

- Widely used interpreted programming language
- Mature community and “modules” at CPAN.perl.org
- Large sysadmin and netadmin market share
- Easy to write, easy to write ugly, read-once code
- Perhaps not as “cool” as some newer languages
- Also see:
  - http://www.perl.org/learn.html
  - Randal Schwartz, Introduction to Perl, Linux Pro (aka Linux Magazine) special issue, May 2010

helloworld.pl

- Create text file, first two characters left justified:

```
#!/usr/bin/perl
print “hello, world\n”;
```

- Then:

```
$ chmod +x helloworld.pl
$ ./helloworld.pl
```

- Or simply:

```
$ perl -e 'print “hello, world\n”'
```

Perl syntax

- Statements terminated with a semicolon

```
$counter = $counter + 1;
$counter++;
```

- Line continuation not needed, white space ignored

```
@mynets = qw(
    192.0.2.0/24 198.151.100.0/24
);
```

- Comments preceded with hash character (#)

```
# IP address validation utilities
return if $int > 2**32; # out of range
```

scalars

- Fundamental data type in Perl, number or string
- Scalar literal
  - `42          # integer`
  - `'foo'       # string`
- Scalar variable, dollar sign followed by a name
  - `$hostname = 'localhost';`
  - `$etype    = 0x0800;`
- Perl generally does the right thing based on context
  - `42 * $hostname; # but likely a bug`

scalar operations

- Typical math operations and operators
  - `+  -  *  /  %  **`
- String concatenation
  - `$i = 'foo' . 'bar'; # $i = 'foobar'`
- String replication
  - `$i = 'foo' x 3;  # $i = 'foofoofoo'`
- Interpolation using double quotes
  - `$x = 'bar';`
  - `$i = “foo$x”;  # $i = 'foobar'`

arrays (aka lists)

- An ordered list of scalars

```
( 'lo', '::1', 100, undef, 'up' );
```

- Named using a leading at (@) symbol
- Individual list elements accessed with $ and index

```
my @protos = ( 1, 2, 6, 17, 89 );
print $protos[0], "\n"  # 1
```

array manipulation [1/2]

- Append or prepend items to an array

```
push @routers, 'phi-ge0', 'lax-ge0';
unshift @routers, 'sea-fe0';
```

- Remove items from the front or rear of an array

```
my $buf = shift @queue;  # $queue[0]
pop @queue;  # $queue[ $#queue ]
```

array manipulation [2/2]

- combine list of strings into one string with separator

```
my $v4addr = join( '.', @octets );
```

- divide string into a list based on a pattern

```
my @octets = split( /\./, $v4addr );
```

hashes

- A “keyed” list of scalars, keys are just strings

```
( e => 10, fe => 100, ge => 1000 );
```

- Named using a leading percent (%) symbol
- Individual list elements accessed with $ and key

```
my %proto = ( tcp => 6, udp => 17 );
print $proto{tcp}, “\n”;
```

complex data structures

- array of arrays
- hash of arrays
- array of hashes
- hash of hashes
- This is where Perl can get real ugly quick

```
@{$rtr{eth0}{.0}{acl}{in_rules}}
```

regular expressions (regex)

- search/find/match on strings using flexible patterns
- Some characters have special meaning
- Forward slash is default delimiter, $_ default string

```
/Traceback/
/(eth[0-9]+\.\d+)/
if ( $email =~ /^abuse@/ ) { # foo }
```

- Parentheses for grouping and “capture”

```
/%LINK-4-ERROR: (\S+)/;
print “interface errors on: $1\n”;
```

more on regex

- regex modifiers and quantifiers

```
m{ \A \s* (?: # .*)? \Z }x;
```

- inline transformation (aka replace)

```
s/nanog/NewNOG/ig;
```

- common syntax

```
next if $int !~ /^ge/;
last if $path =~ / $asn i$/;
/\d+\.\d+\.\d+\.\d+/
/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/
```

evaluating falsehood

- These scalars evaluate as false
  - `0  “”  undef  “0”`
- True is everything else
  - `if ($eth0) { # then do something }`
- But be careful, what if $util = “0.0”?
  - `if ( ! $util ) { # do something }`

if ... elsif ... else ...

```
if ( 4 == ipversion($addr) ){
    $bits = 32;
}
elsif ( 6 == ipversion($addr) ) {
    $bits = 128;
}
else {
    die “unknown IP version”;
}
```

while

```
while ( <> ){
    chomp;
    $traceback++ if /Traceback/;
}

if ($traceback)
    print “Traceback messages found\n”;
```

for (C-style)

```
for ( $int = 0; $int < 2**32; $int++ ) {
    print
        join( '.',
            reverse unpack('C4',
                           pack('I',
                                $int
                               )
                          )
            ),
        “\n”;
}
```

foreach

```
for ( @interfaces ) {
    s/loopback/lo/;
}

foreach (1 .. 10 ) { ping($router)}

for my $router (@routers) {
    my $rtt = ping($router);
    print “$router: $rtt\n”;
    last if $rtt > $THRESHOLD;
}
```

control structure notes [1/2]

- ternary operator (? :), use sparingly if possible
  - `$bits = $ver == 4 ? 32 : 128;`
- single statement blocks don't need braces

```
if ( $count > $MAX_PREFIX )
    alert( 'max prefix exceeded' );
else
    log( 'accepting $count prefixes');
```

control structure notes [2/2]

- Unless
- Until
- Abbreviated control blocks
- statement … (if | while | unless | until | for);

I/O operations

- Send string(s) to file handle, by default to STDOUT
- printf, like print, but with formatted output

```
printf "%-5s | %-15s \n", $asn, $addr;
```

- reading STDIN

```
while (<>) {# do something with ARGV}
```

- open, close and file handles

```
open( my $CONF, '<', $cfg_file )
    or die “open error: $!”;
# . . .
close($CONF) or die “close error: $!”;
```

subroutines (aka user-defined functions)

- Instruction set that can be called independently
- Often easier to build and debug small code blocks

```
sub get_v4ptr_name {
    my $addr = shift || return;
    $addr    = join( '.', reverse
               split( /\./, $addr ) );
    return $addr;
}

my $qname = get_v4ptr_name($ipv4addr);
```

sorting

- sort function by default sorts a list of strings

```
@hosts = qw ( bob alice carol );
sort @hosts;  # alice, bob, carol
```

- so, this probably not what you want

```
@ttls = ( 300, 5, 900, 3600, 1800 );
sort @ttls; # 1800 300 3600 5 900
```

sort with subroutine

- sort strings with subroutine (same as before)

```
@hosts = qw ( bob alice carol );
sort { $a cmp $b } @hosts;
```

- sort numerically with subroutine

```
@ttls = ( 300, 5, 900, 3600, 1800 );
sort { $a <=> $b } @ttls;
# 5, 300, 900, 1800, 3600
```

- sort numerically with subroutine, descending

```
sort { $b <=> $a } @ttls;
```

strict, warnings and taint

- Most of us are better off coding defensively
- Three common pragmas I use in Perl scripts
- **`use warnings`** – helps identify likely bugs
- **`use strict`** – declare variables, limits barewords
- **`-T`** – (taint mode) various security checks

special variables

- **$|** - if nonzero, immediately flush on output
- **$_** - default input and regex string to match
- @_ - array of parameters passed to a subroutine
- **$<**, **$>** - real user id, effective user id
- **$1**, **$2**, … - ordered parentheses regex capture
- ***$/*** - input record separator
- **$@** - error message from last eval()

modules for network operators

- Net::Patricia
- Net::DNS
- NetPacket
- NetAddr::IP
- Net::IP
- Net::Pcap
- Data::Dumper

combat Perl script toolbox

- tweet – command line Twitter update
- tosyslog – send a string to server
- ptrforward – reverse address verification
- ospfdb – consistency check for OSPF routers
- cinfo – cisco device summary
- lostrules – identify unused ACLs/firewall rules
- cislog – Cisco log summary audit report
- bhrs – black hole route server
- pcapsum – summarize a libpcap (tcpdump) file

in closing

- “I don't know Perl, I know combat Perl”
- “Don't run this as root”
- Perl Best Practices, Damien Conway
- Please send questions, suggestions or scripts to:

jtk@cymru.com

PGP key 0xFFE85F5D

http://www.cymru.com/jtk/