# Unix Command-Line Kung Fu

Hal Pomeranz, Deer Run Associates



Hal Pomeranz
Deer Run Associates
PO Box 50638
Eugene, OR 97405

hal@deer-run.com
(541)683-8680
(541)683-8681 (fax)
*http://www.deer-run.com/*

*[I wish to thank everybody who's attended this presentation and given me suggestions for improving the content. I haven't been able to always get your name/email address to thank you explicitly in the course notes, but your contributions are appreciated by me and everybody who uses this course. --Hal]*

## Who Is Hal Pomeranz?

- Independent IT consultant
- Senior Unix Security faculty for SANS Institute
- Earlier episodes:
  - First root access: 1987 (Sun 3/160, SunOS 3.4)
  - Former board member: BayLISA, BBLISA, USENIX
  - SAGE Outstanding Achievement Award recipient
  - Technical consultant for CIS Unix security standards
  - Last Technical Editor for *Sys Admin Magazine*

Welcome! My name is Hal Pomeranz and I've been working with Unix systems professionally since 1987. By the way, when I say "Unix", I mean all Unix-like systems, including Linux. It's all rock'n'roll to me…

For the last 10 years my wife Laura and I have been running our own consulting practice (although she claims she's "not technical anymore", my wife was using Unix systems many years before I was and she's still a mean hand with the `vi` text editor). I also have the curious distinction of being the "oldest" current SANS Faculty member (in terms of longevity with the organization, not by age), having presented my first tutorial for SANS in 1994 and various other talks at SANS conferences from the early '90s. I'm currently the track lead and primary instructor for SANS' Unix Security certification track (aka SANS Sec506).

I've been active in the Unix community throughout my career and have served on the Boards of several different computing and system administration organizations, including BayLISA (San Francisco Bay Area), BBLISA (Boston), and USENIX. I was the last Technical Editor for *Sys Admin Magazine,* from Jan 2004 through Aug 2007 when the magazine ceased publication. I've also helped to develop many of the existing Unix security standards, including those from the Center for Internet Security (*http://www.CISecurity.org/*). I am also a recipient the annual SAGE Outstanding Achievement Award for my teaching and leadership in the field of System Administration.

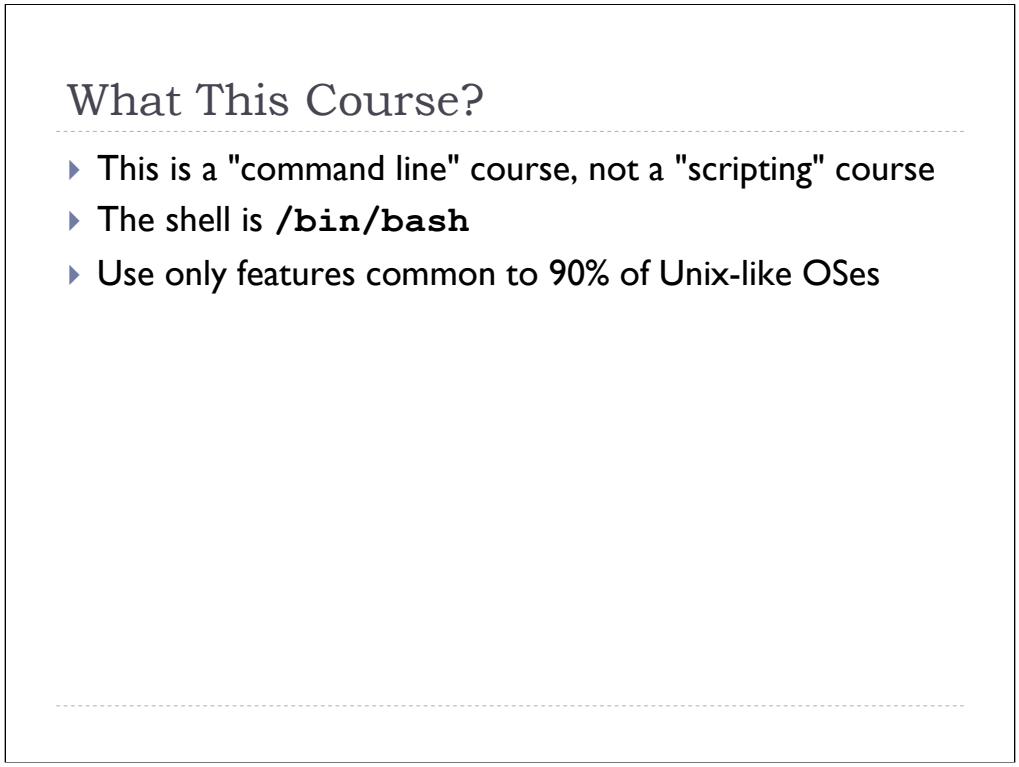
## Why This Course

- I teach Unix to several hundred people per year and see them struggling with the command line
- Little tricks provide massive productivity increases
- … really it's all Ed Skoudis' fault!

At SANS Conferences and other venues, I teach various Unix skills to hundreds of students every year. Many of them are relatively inexperienced with the Unix command line and I see them getting frustrated or taking round-about approaches to solving problems, when in reality just knowing a few simple tricks would make them vastly more productive.

I had considered putting a course together to help students learn some of these tricks in a systematic way, but never seemed to find the time. Then fellow SANS Faculty member Ed Skoudis developed a course he called *Windows Command-Line Kung Fu*. Frankly, it was galling to me that there should be such a course for the Windows folks, and nothing at all for the folks working with Unix, which is a much more command-line oriented OS. So thanks, Ed, for your advice in the early stages of this course and for kicking me in the posterior when I needed it.

Ed and I, along with Paul Asadoorian are now participating in a blog called "Command Line Kung Fu" (*http://blog.commandlinekungfu.com/*), where we solve problems and show you both the Unix and the Windows command-line version. We hope you'll check us out.



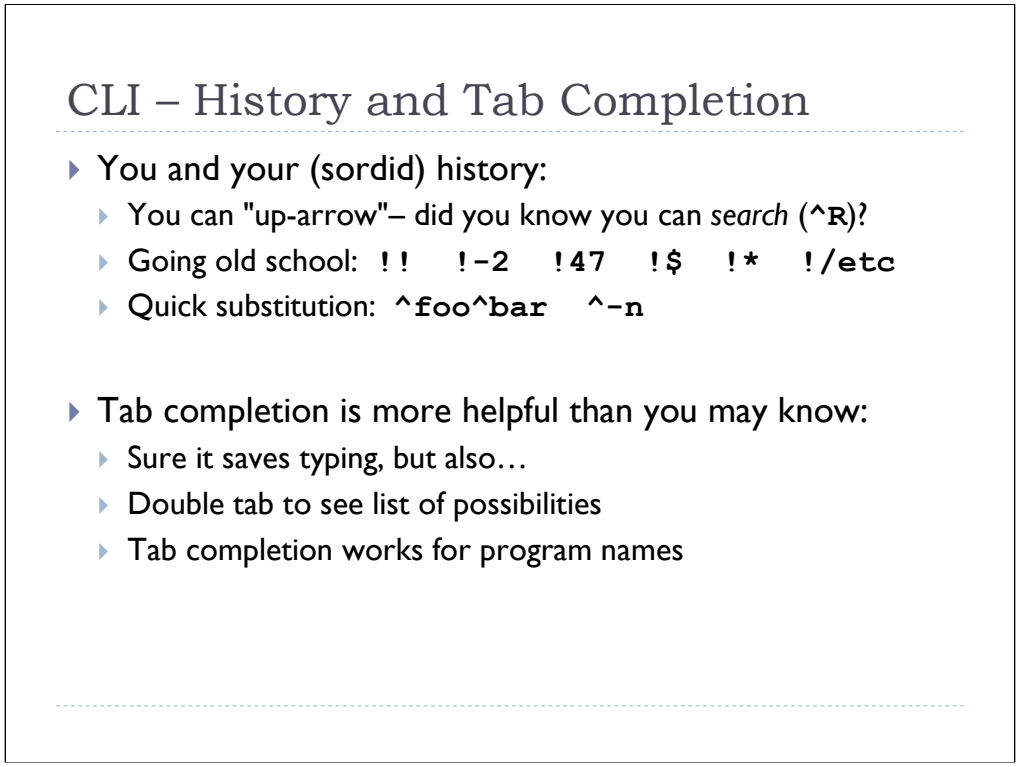

Before we get to the material, let's establish a few ground rules:

• This is a command-line course, not a scripting course. While sometimes the things you type on the Unix command-line can come perilously close to scripting, the focus of this course will be on tools and techniques that you would commonly use for one-shot, "on-the-fly" kinds of tasks. Also no pre-configured command aliases or other special environmental settings are assumed, and are expressly against the "rules" for all scripting challenges presented in the course.

• We will be using the command-line syntax for the Free Software Foundation's `bash` shell, which is widely available on all Unix-like operating systems. That being said, the techniques in this course are almost all portable to `ksh` and `zsh`.

• When we're using Unix commands, we will restrict ourselves to standard commands and command-line options that are present in the default install of the majority of standard Unix systems. In other words, it's against the "rules" to use esoteric options from the GNU versions of various commands, even if they are darn useful.

Now that we're clear on the rules, let's have some fun…




When working with the Unix command-line, one of the biggest productivity enhancements is to take advantage of the various features of your command-line history and the tab-completion feature in your shell. These features make building up complicated shell pipelines considerably easier, and save you lots of keystrokes.

## Command-Line History

If you've been using the shell for a while, you're probably aware that you can use the up and down arrows on your keyboard to move backwards and forwards through your history of previous command lines. But what if you want to re-run a command that you last did several dozen command-lines ago? Hitting "up arrow" that many times is tedious and you'll be banging that arrow key so fast that you're likely to "overshoot" and miss the command line you wanted.

The neat thing about the shell history is that you can search backwards using `<Ctrl>-R`. Just hit `<Ctrl>-R` and then start typing the string that you're looking for– the shell will show you the most recent matching command line that contains the string you've typed. You can hit `<Ctrl>-R` again and (and again and …) you'll be taken further back into your history of matching command lines. When you've found the command-line you want, just hit `<Enter>` to execute the command, or use the normal editing keys to modify the command-line as desired.

## Keyboard Accelerators

However, command-line history is a extremely old feature of Unix shells (having first appeared in the BSD `csh` back in the 80's. When command-line history was first introduced, the up/down arrow and backwards searching features were not even conceived of yet. Instead, there were various keyboard accelerators that have now mostly been forgotten. Still, these keyboard macros are often substantially faster and easier than using the arrows and `<Ctrl>-R`, especially if you're a touch typist and don't particularly care to go reaching for the arrow keys all the time.

For example, `!!` repeats the previous command:

```
$ ls -l /var/log/messages
-rw------- 1 root root 27127 Apr 29 08:32 /var/log/messages
$ !!
ls -l /var/log/messages
-rw------- 1 root root 27127 Apr 29 08:32 /var/log/messages
```

Similarly, `!-2` repeats the command *before* the previous command, and as you might expect `!-3` goes three command lines back, etc. This can be useful when you're repeating the same sequence of commands over and over, like when you're watching a log file or other fast growing file to make sure it's not filling up your file system:

```
$ ls -l /var/log/messages
-rw------- 1 root root 27127 Apr 29 08:32 /var/log/messages
$ df -h /var
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda3             996M  122M  823M  13% /var
$ !-2
ls -l /var/log/messages
-rw------- 1 root root 27127 Apr 29 08:32 /var/log/messages
$ !-2
df -h /var
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda3             996M  122M  823M  13% /var
```

You can also use `!!`, `!-2`, etc in the middle of subsequent command lines. For example:

```
$ ifconfig eth0
-bash: ifconfig: command not found
$ /sbin/!!
/sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:95:AB:90
          inet addr:192.168.127.129  Bcast:…
```

This is also an extremely useful technique when building up long shell pipelines– just keep using `!!` and adding little bits of code to the end of the pipeline until you get the results you want.

## History by the Numbers

Every command-line in your history is numbered (you can see the numbers in the left-hand column when you use the `history` command) and you can select a particular command-line using `!`*<n>* where *<n>* is the number of the command:

```
$ history
…
   41  ls -l /var/log/messages
   42  df -h /var
   43  ifconfig eth0
   44  /sbin/ifconfig eth0
   45  history
$ !41
ls -l /var/log/messages
-rw------- 1 root root 27127 Apr 29 08:32 /var/log/messages
```

The `!`*<n>* syntax is most useful when you find yourself running one particular command over and over again with a lot of other commands interspersed between executions.

## Specifying Arguments

There are also keyboard accelerators for extracting particular command-line arguments from previous command-lines. Perhaps the most useful one is `!$` which gets the last argument from the previous command-line:

```
# co -l named.conf
named.conf,v  -->  named.conf
revision 1.1 (locked)
done
# vi !$
vi named.conf
# ci -u !$
ci -u named.conf
named.conf,v  <--  named.conf
file is unchanged; reverting to previous revision 1.1
done
```

Like the previous example, there are any number of times that you will need to do a series of commands to a single file, and this is where `!$` really shines. By the way, you can use `!-2$` to get the last argument from the command-line prior to the previous command-line (and `!-3$`, `!-4$`, and so on also work like you'd expect).

`!*` gets you *all* of the previous arguments. This is often useful when you make a typo in your command name:

```
# cl named.conf named.conf-orig
bash: cl: command not found
# cp !*
cp named.conf named.conf-orig
```

In general, `!:`*<x>* will give you the *<x>*th argument from the previous command-line, but I don't find this syntax particularly useful. Actually, all of these accelerators we've been discussing are just degenerate cases of the generalized syntax "`!`*<n>*`:`*<x>*" (give me the *<x>*th argument of command-line *<n>*).

## Fast Searching

One last accelerator that's extremely useful is "`!<string>`", which means execute the last command-line that begins with *<string>*. For example, you might do "`/etc/init.d/httpd start`" trying to start your web server only to discover that some misconfiguration is preventing the server from starting. After fixing the problem you can just do "`!/etc`" to try starting the server again.

Of course it can be dangerous to just blindly go around doing things like "`!/etc`" or whatever. So you can do "`!<string>:p`" to display (print) the last command-line that starts with *<string>* before executing it:

```
$ !/sbin:p
/sbin/ifconfig eth0
$ !/sbin
/sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:95:AB:90
          inet addr:192.168.127.129  Bcast:…
```

In the example above, we use "`!<string>:p`" followed by "`!<string>`" to execute the command. But in fact you can just use "`!!`":

```
$ !/sbin:p
/sbin/ifconfig eth0
$ !!
/sbin/ifconfig eth0…
```

## Quick Substitutions

Another boon for people who make lots of typos is the ability to do quick substitutions on the previous command line using the caret (^) operator. Earlier we used `!*` to fix things when we made a typo on a command name, but you can also use the caret for this:

```
# cl named.conf named.conf-orig
bash: cl: command not found
# ^cl^cp
cp named.conf named.conf-orig
```

The caret operator replaces the *first* instance of the provided string on the command line, but only the first (unfortunately there's no global replacement option as there is with `sed` or Perl). This is sometimes an annoying limitation:

```
# cp passwd passwd.bak
# ^passwd^shadow
cp shadow passwd.bak
```

The above outcome– overwriting the `passwd.bak` file with a copy of the `shadow` file– is probably not what you wanted.

Actually, believe it or not, the following does what you want:

```
# cp passwd passwd.new
# !!:gs/passwd/shadow/
cp shadow shadow.new
```

Rather than using the caret operator, we're using the more general substitution modifier ("`:s/…/…/`") on the previous command ("`!!`"). The leading "`g`" means to apply the substitution "globally" throughout the entire previous command, rather than to just the first instance (all you `sed` and Perl folks are probably boggling now because you're used to the "`g`" appearing at the end of the substitution rather than the beginning). The above syntax is quite a lot to type– I'm not sure it's much faster than just editing the previous command-line directly.

The "^*<string>*" syntax is a useful because it simply removes *<string>* from the previous command line (basically you're saying replace *<string>* with an empty string). I often use this with `make` or other Unix commands that have a "`-n`" option for showing you what would happen if you ran the command. Once you're sure that everything looks correct, you can quickly strip the "`-n`" option and actually execute the command:

```
$ make -n dlstubs
cc     dlstubs.c   -o dlstubs
$ ^-n
make  dlstubs
cc     dlstubs.c   -o dlstubs
```

## Tab Completion

Tab completion really saves you a lot of typing because it quickly fills in pathnames for you without your having to type the entire string. For example, if you type "`ls -l /var/log/me<Tab>`" and the shell would automatically complete the pathname as `/var/log/messages`.

However, if you do this you'll probably hear a beep after the shell completes the pathname. This means that `/var/log/messages` is the longest unique sequence of characters that the shell could match, but that there are multiple matching pathnames that begin with `/var/log/messages`. At any time you can hit the tab key twice (`<Tab><Tab>`) to see all possible completions:

```
$ ls -l /var/log/messages<Tab><Tab>
messages    messages.1  messages.2  messages.3  messages.4
$ ls -l /var/log/messages
```

Notice that after displaying the different possible matches, the shell puts you back at the end of the command line you were working on when you hit the double tab.

What many people don't know is that you can also use tab completion with executable names:

```
$ ls<Tab><Tab>
ls              lsb-release.d  lsmod          lspcmcia
lsattr          lsdiff         lsof           lspgpot
lsb_release     lshal          lspci          lss16toppm
$ ls
```

Aside from just saving a few keystrokes, this can also help you remember a command name you've forgotten.

## Traversing File Systems w/ **find**

- By type:

```
find /dev -type f -print
```

- By name:

```
find / -name '.* *' -print
```

- By size:

```
find / -size +10000000c –print
```

- By "last modified time":

```
find / -mtime –7 –print
```

## `find` Command Basics

Traversing and searching file systems and directories is a very common operation in Unix. Normally we use the `find` command for this, though many Unix commands have a "recursive" option (typically "`-r`" or "`-R`") for operating on an entire directory tree, such as "`rm -r …`" or "`chown -R …`".

The syntax of the `find` command is a little odd, but it helps if you think of breaking the arguments into chunks as follows:

find [*list of dirs*] [*search option(s)*] [*action(s)*]

The standard action is "`-print`" which means to display the names of all files that match the search option(s). In fact, on most modern versions of `find` you can leave off the action specifier and "`-print`" will be assumed.

There are a lot of different search options out there, and in fact different versions of `find` on the various Unix flavors will often support search options that may not be supported on other platforms. That being said, there tends to be a core group of common options that are universally supported:

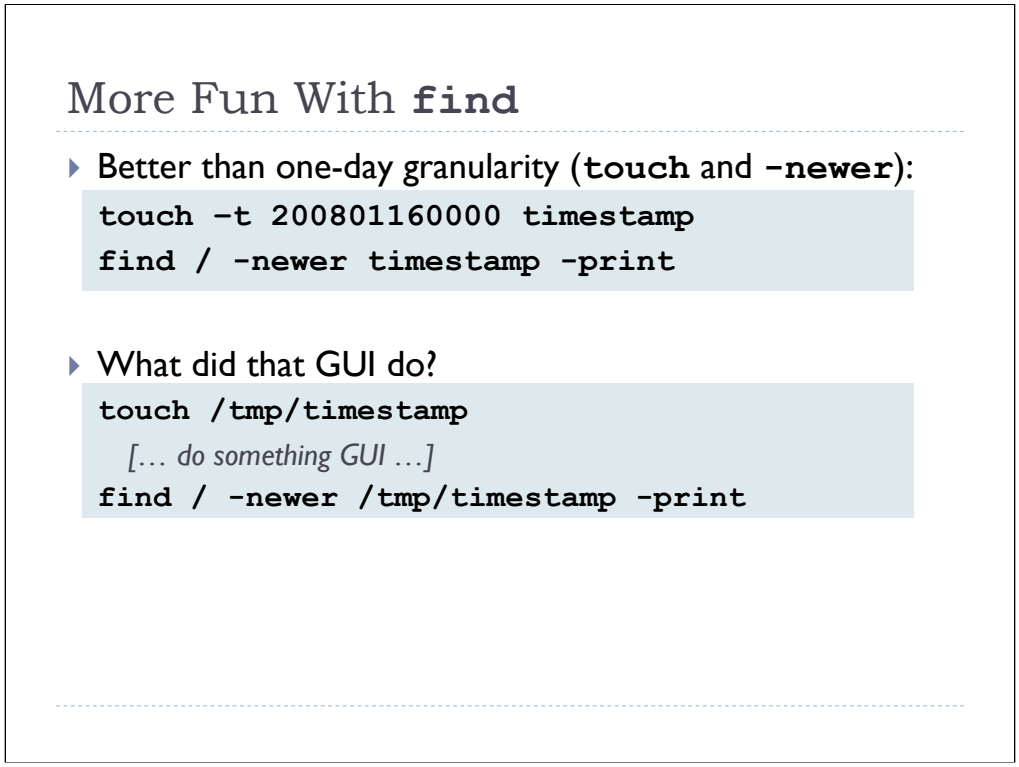
• You can use "`-type`" to look for certain types of objects: "`f`" means regular files, "`d`" for directories, "`l`" for symlinks, etc. The first example on the slide is a very useful `find` command to run if you think your system has been compromised. Many rootkits will put files into `/dev` in an attempt to hide them from system admins. However, since regular files under `/dev` are not expected (except the `MAKEDEV` script on some Unix flavors and various files under `/dev/.udev` on Linux), the `find` command shown here can help pinpoint signs of a break-in.

• You can, of course, find files by name with the "`-name`" option. Notice that you can use normal shell globbing characters like "`*`" in your expressions, but you have to be careful to quote your search strings so that the shell doesn't try to interpolate the wildcards before they get to the `find` command.

• Sometimes searching for files by size can be useful– for example when you're looking for runaway log files and data files that might be filling up a partition. Or perhaps an attacker has had a long-running packet sniffer going on your system to capture passwords and you want to find its capture file. Large files on Unix systems are just not that common. In the example on the slide we're searching for all files that are larger than (the "`+`" means "greater than", "`-`" means "less than") 10 million bytes ("`c`" for "character", which is a one-byte data type).

• Or perhaps after a break-in you might want to get a list of recently modified files. The example finds all files that have been modified ("`-mtime`") less than (again "`-`" generally means "less than" to `find`) 7 days ago. While it's possible that the attacker may have modified your file timestamps back to their original value, many don't bother.

Note that the examples on this slide are taken from Ed Skoudis' excellent *Intrusion Discovery Cheat Sheet for Linux* available for free from the SANS Institute (*http://www.sans.org/score/checklists/ID_Linux.pdf*). I highly recommend this document for your operations staff and system admins. There's also a Windows version available (replace "Linux" with "Windows" in the previous URL).

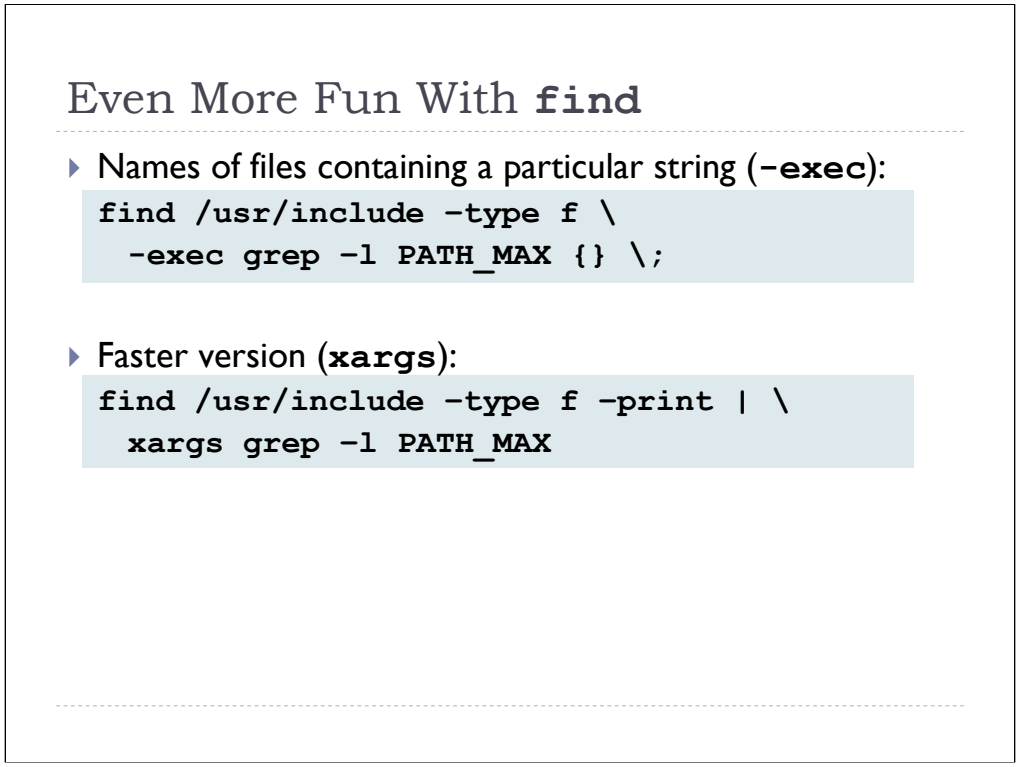



## A Trick for Better Time-Based Searches

One of the problems with the "`-mtime`" option is that it only works in terms of one day values. But what if you were able to pinpoint the time of your break-in by looking at your IDS logs (or some other reference point) and know that the break-in occurred 36 hours ago? Sure, you could do "`-mtime -2`", but on a busy system that might generate lots of extra noise.

It turns out that the superuser can use the `touch` command to set timestamps on files (which is what attackers do to reset the timestamps on files that they modify) and/or create new files with arbitrary timestamps. So if you know exactly when your break-in occurred, just use `touch` to create a new file with a timestamp that matches the time of the break-in and then use find with the "`-newer`" option to `find` all files with more recent last modified timestamps.

Note that you can use a similar trick to figure out what's going on under the covers with some of these GUI-based admin tools that are becoming so prevalent. Just create a timestamp file before you do the operation with the GUI. When you're done, just find everything that's been changed since you created your timestamp file.




## Running Arbitrary Commands ("-exec")

It's often useful to execute a particular command (or set of commands) on the matching files discovered by `find`. You can use the "`-exec`" action for this. Here are a couple of simple (but useful) examples:

```
find /tmp -mtime +7 -exec rm -rf {} \;
find /var/log -mtime +7 -exec gzip {} \;
```

The syntax of `-exec` is a little weird. After the `-exec`, you specify the command line you want to run but you use curly braces ("`{}`") to indicate where in the command line you want `find` to substitute the matching file names. The command after `-exec` must be terminated with "`\;`" (it's possible that you might have other actions or expressions after `-exec`, though usually the "`\;`" is the last thing on the line).

The example on the slide is a useful little expression for displaying the names of files that contain a particular string– I often use this for searching directories of source code for a particular item. Normally, of course, `grep` would display the matching lines, but the "`grep -l`" command means "only display the file names".

## Improving Performance

It turns out that the first `find` example on the slide is pretty inefficient, because `find` will end up running `grep` on each individual file, which is a whole lot of separate executions of `grep`. Instead, you might consider piping the output of "`find … -print`" into the `xargs` program. `xargs` gobbles up the file names from its standard input and uses them to construct and execute command lines, subject to the built-in argument list length limitations in the shell. The result is that the `grep` command will end up being executed many fewer times by `xargs` than it will with the `find` command.

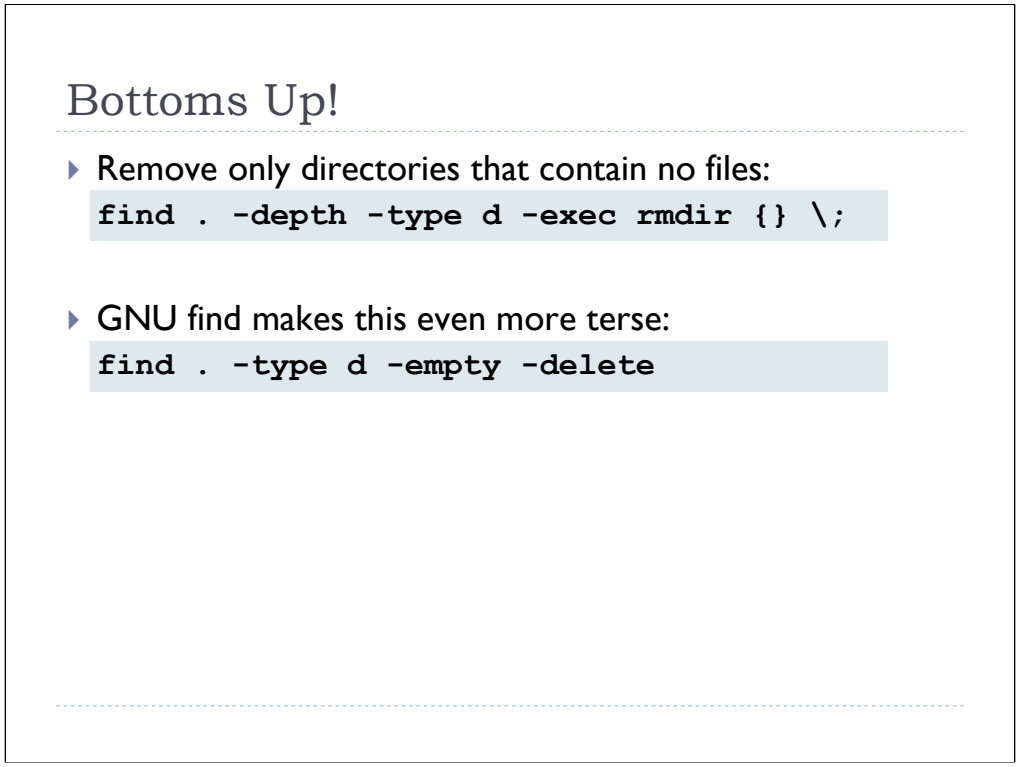
You can see the performance improvement using the built-in "`time`" function in the shell, which is useful for doing quick benchmarks like this:

```
# time find /usr/include -type f \
   -exec grep -l PATH_MAX {} \; >/dev/null
real    0m11.488s
user    0m1.570s
sys     0m10.732s
# time find /usr/include -type f -print | \
   xargs grep -l PATH_MAX >/dev/null
real    0m0.300s
user    0m0.076s
sys     0m0.270s
```

What's interesting to me is that the "`find … | xargs …`" example actually appears to be slightly faster than "`grep -rl …`":

```
# time grep -rl PATH_MAX /usr/include >/dev/null
real    0m0.437s
user    0m0.074s
sys     0m0.345s
```

Of course not all versions of Unix ship with a `grep` command that supports the "`-r`" option anyway…

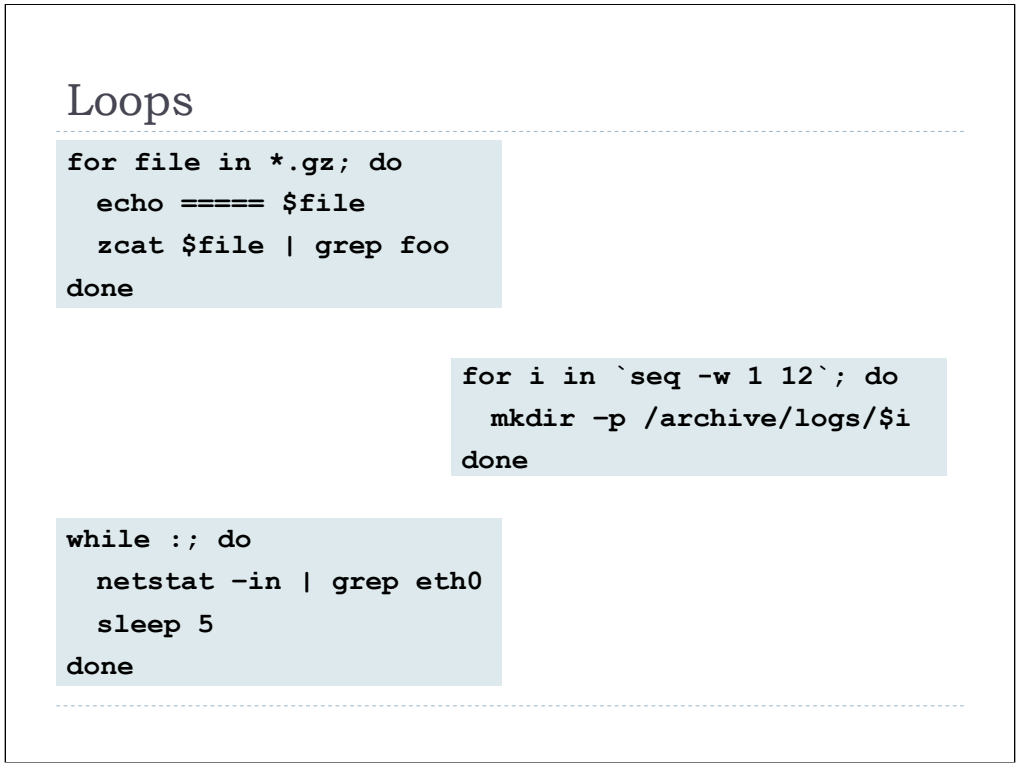



## Depth-First Traversal

One of our Command Line Kung Fu Blog readers, Bruce Diamond, presented us with an interesting problem. If you have a directory structure where many of the directories are empty, can you remove just the empty directories without removing the ones containing files? The trick is that if removing all of the subdirectories of a given directory leaves that directory empty then that directory should be cleaned up as well. Basically you want to start at the bottom of the directory structure and work your way back up, removing empty directories as you go.

Normally the find command does what's referred to as an "in-order traversal" of the directory structure– it starts at the top and works its way down each sub-tree. But with the "`-depth`" option ("depth-first traversal"), you can force find to start at the bottom of the directory tree and work its way back up. Since `rmdir` will only remove empty directories, using the combination of "`-depth`" and "`-exec rmdir …`" is exactly what we want.

Note that the GNU version of `find` includes the "`-empty`" condition and the "`-delete`" action, which simplify our command even further, but of course this violates our rule of using options that are common to most Unix systems. Nevertheless, they're darn useful.




## Loop Constructs

The `find` program is essentially an interator over directories of files, but sometimes you need a more general looping construct. `bash` actually has several different types of loops available, but we'll just discuss a couple of them here.

The most common type of loop I find myself doing on the command-line is the "foreach" type of loop that processes a list of file names or other values. As you can see in the first example, you can use shell wildcard globs to create lists of file names to process. This first example is an idiom I use frequently for finding a particular string in collections of compressed/gzipped files. The `echo` statement outputs an easily recognizable header before the matching output from each file so that it's easy to see which file(s) the matches occur in.

In the second example we're using the `seq` command to generate a list of numeric values from 01 to 12 (the "`-w`" option forces `seq` to produce consistent width values, zero-filling as necessary). We then use backticks to substitute the output of `seq` as the list of values in our for loop

Actually, bash has a C-style `for` loop, so we could do this without `seq`:

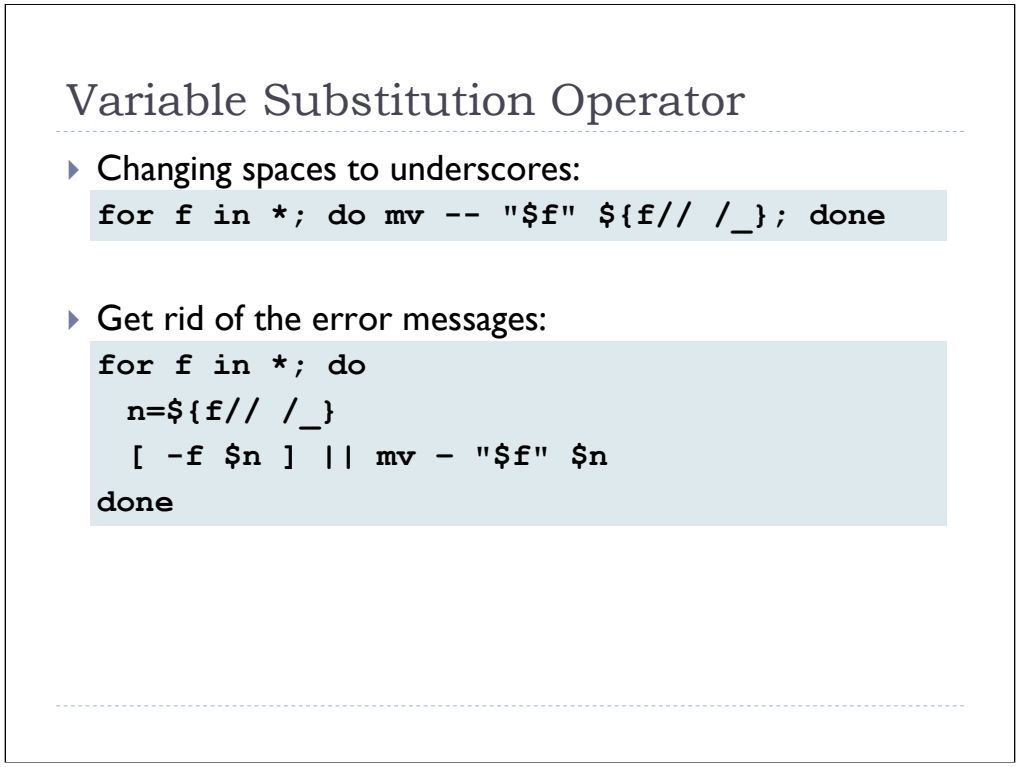
```
for ((i=0; $i <= 12; i++)); do
    mkdir -p /archive/logs/`printf %02d $i`
done
```

Frankly, I think the version with `seq` in backticks is a lot clearer, but your mileage may vary.

Just to tie a bow on this discussion, I should point out that this is really a fairly poor example since you could do it without a loop at all:

```
mkdir -p /archive/logs
cd /archive/logs
mkdir `seq -w 1 12`
```

Sometimes infinite loops are useful. The last example shows an idiom that I use frequently when I want to monitor the output of a command at regular intervals over a long period of time. For example, suppose you wanted to watch how much traffic was going out your ethernet interface. You can use the last loop on the slide to watch the `netstat` output for this interface at five second intervals.




## Variable Substitution Operator

Now that we've got loops under our belt, we can start using them for interesting things. In this case we're going to combine a loop with another useful bash feature: the variable substitution operator. For example, suppose you had a lot of file names with spaces in them and you wanted to replace the spaces with underscores to make the files easier to deal with. In the bad old days, you would have had to do something cumbersome like calling `sed` to do the substitution, but bash has a built-in operator for this now.

The basic form of the operator is "`${`*variable*`/`*pattern*`/`*replacement*`}`", but here we're adding an extra "`/`" at the beginning of the pattern which means "replace all instances of the pattern", as opposed to only replacing the first instance which is the default (if you're familiar with `sed`, think of the extra "`/`" like the "`g`" operator in `sed`).

Notice we're careful to use quotes around the unmodified file name variable in the `mv` command so that the spaces in the file name don't mess up the command. We've even included a "`--`" to indicate to `mv` that it shouldn't expect any further command-line options, just on the off chance that the file name we're dealing with happens to start with a "`-`".

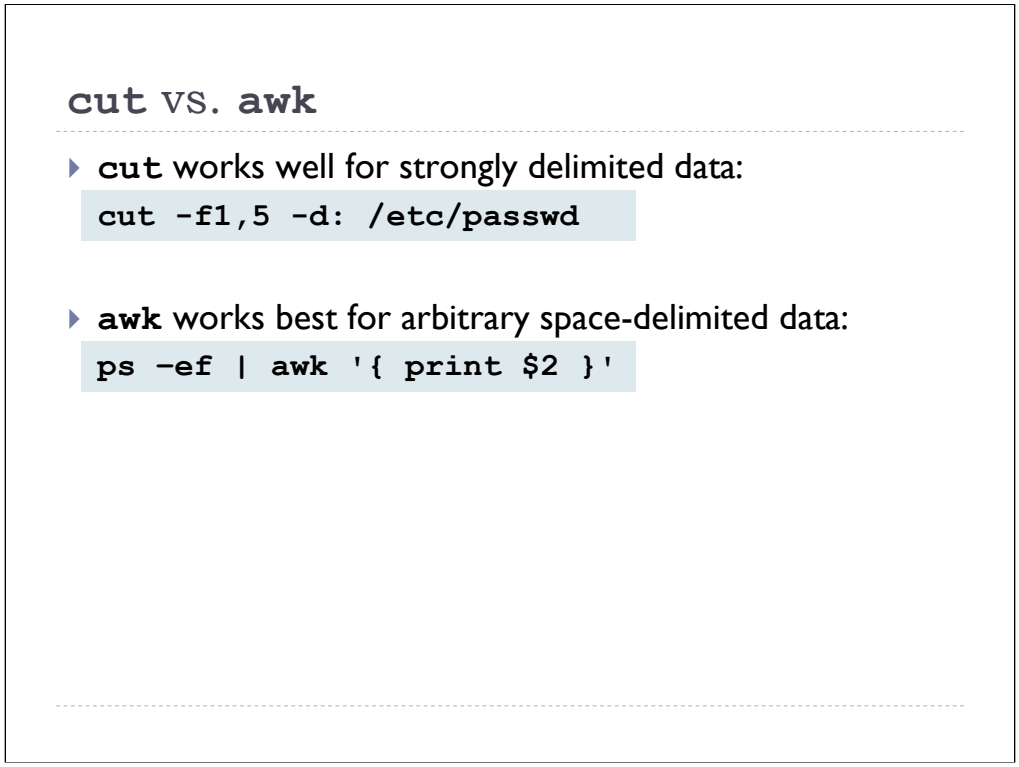
Unfortunately, there are a couple of problems with our solution:

```
$ touch 'foo bar' foobar
$ echo foobar >foo_bar
$ for f in *; do mv -- "$f" ${f// /_}; done
mv: `foobar' and `foobar' are the same file
mv: `foo_bar' and `foo_bar' are the same file
$ cat foo_bar
```

Here I'm creating a directory containing three files: empty files "`foo`*<space>*`bar`" and "`foobar`" plus a file called "`foo_bar`" that contains the string "`foobar`". Notice that our loop generates an error for each of the two file names that don't contain a space. Furthermore, the empty "`foo`*<space>*`bar`" file ends up overwriting the "`foo_bar`" file, which is probably not what we want.

So the second form of our loop handles things a bit more carefully. First we use our variable substitution operator to compute what the new file name should be and tuck that value into the variable `$n`. Next we use the file test operator, "`[ -f $n ]`", to see if a file named `$n` already exists. The `mv` command on the other side of the "`||`" (read this as "or") will only execute if the file test is not true, i.e. if a file with the new name doesn't already exist. This not only prevents us from overwriting existing files but also prevents the spurious errors when the original file name contains no spaces, resulting in `$f` and `$n` being equal.

The idiom "`[` *testsomething* `] ||` *somecommand*" is a convenient short-hand for an "if" style conditional. Note that there is also a "`&&`" ("and") operator in addition to "`||`" ("or"). Choose the appropriate operator depending on the logic in your conditional expression.




## cut vs. awk

It's often useful to pull particular fields out of lines of input, and the most common command-line tools for doing this in Unix are `cut` and `awk`. `cut` is most useful when the input you're dealing with is strongly delimited, as in the `/etc/passwd` file where every field is separated with colons. In the first example on the slide, we're pulling the first (user name) and fifth (user full name, or GECOS) field from `/etc/passwd`. The fields will be colon-delimited in the output:

```
# cut -f1,5 -d: /etc/passwd
root:root
bin:bin
daemon:daemon
…
```

Note that `cut` also allows you to select a range of characters ("`-c3-7`"), but I don't find myself using this feature that often.

On the other hand, there's an awful lot of files and command outputs in Unix that are delimited by arbitrary amounts of whitespace. `cut` doesn't handle this kind of input very well, but this kind of parsing is exactly what `awk` was designed to do. `awk` is obviously a full-blown scripting language in its own right, but we'll just restrict ourselves to simple `awk` idioms that are useful on the command line.

At its simplest, `awk` merely breaks up each line of input on whitespace and makes the various fields available in numbered variables `$1`, `$2`, and so on. So if you want all of the process IDs (second column) from some `ps` output, just pipe the output of `ps` into "`awk '{ print $2 }'`". It's usually necessary to quote the `awk` code to protect it from interpolation by the shell.

## More **awk** fun

- Look for extra UID 0 accounts:

```
awk -F: '($3 == 0) { print $1 }' /etc/passwd
```

- Accounts with no password set:

```
logins -p    # not available on all Unix systems
awk -F: '($2 == "") { print $1 }' /etc/shadow
```

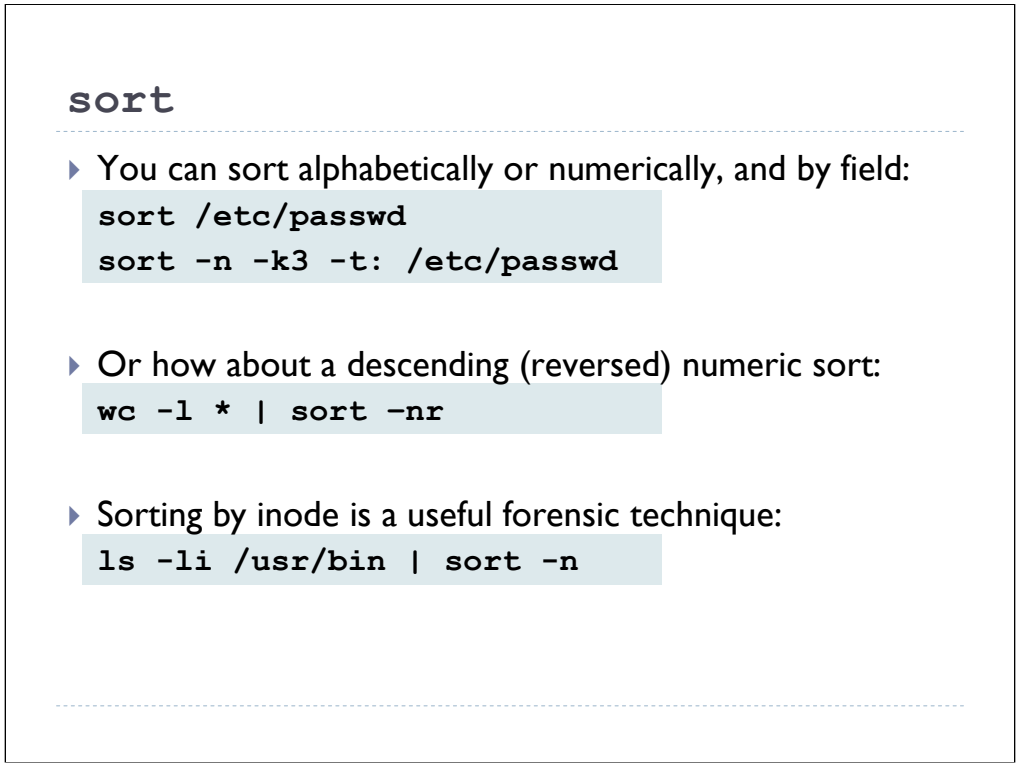
- "… **|** **grep** … **|** **awk** …" considered stupid:

```
kill `ps -ef | grep sshd | awk '{ print $2 }'`
kill `ps -ef | awk '/sshd/ { print $2 }'`
```

However, you can also use conditional operators with `awk` to select particular lines from the output and take action only on those lines. In the first example we're printing the user names (field 1) from all lines in the `passwd` file where the UID (field 3) is zero. This can help you discover if attackers have added extra superuser accounts in the middle of a large `passwd` file. Notice that `awk` is perfectly capable of dealing with delimiters other than whitespace– just specify the delimiter character after `-F` (similar to the `-d` option with `cut`).

Detecting accounts with null password entries is another good auditing procedure, and on some Unix operating systems the `logins` program can help with this. However, `logins` is not available on a wide variety of Unix OSes (like Linux and the BSDs), but `awk` can be used to accomplish the same thing. Just emit the user names of all accounts that have a null second field in `/etc/shadow`.

Note that "`ps -ef | grep <processname> | awk '{ print $2 }'`" is a very common idiom. You typically see it used inside of backticks with the `kill` command to terminate a particular process by name. However, the `grep` in this expression is really a waste of time, since `awk` has built-in pattern matching. So please leave out the `grep`– this is a pet peeve of mine…




## Sorting

Earlier we saw that the `ls` command has options for sorting its output in various ways, but Unix also provides a `sort` command for sorting arbitrary inputs. By default `sort` will do an alphabetic sort, but "`sort -n`" provides numeric sorting instead. `sort` is actually a very powerful program with a wide array of different options. For example, the second example shows how you can specify a delimiter character (similar to `cut` again) and sort on a particular field (in fact, `sort` actually lets you sort on multiple different fields at the same time if you want to). The "`-r`" option allows you to "reverse" the default sort order to do descending sorts.

The last example on the slide is an extremely useful forensic technique. "`ls -li`" produces the typical "`ls -l`" output, but puts the inode number of each file in the first column of output. Every time a file is replaced it gets a new inode, and since inodes are generally assigned in numerical order, sorting the directory by inode will allow you to see the order in which files in that directory have been installed.

The reason this is useful is that if an attacker installs a rootkit, the files installed by that rootkit will all be sorted together in the command output and all have inodes in the same small range of values. So even if the attacker has reset the timestamps on the files, you'll still be able to quickly see the files that got replaced by the attacker.

## uniq

- **uniq** eliminates duplicate lines from *sorted* data:

```
cut -f3 -d: /etc/passwd | sort | uniq
cut -f3 -d: /etc/passwd | sort –u
```

- Use "**uniq -c**" to get a count of repetitions:

```
ps -ef | awk '{ print $1 }' | \
  sort | uniq -c | sort -nr
```

```
cut -f3 -d: /etc/passwd | \
  sort | uniq -c | grep -v ' 1 '
```

## uniq

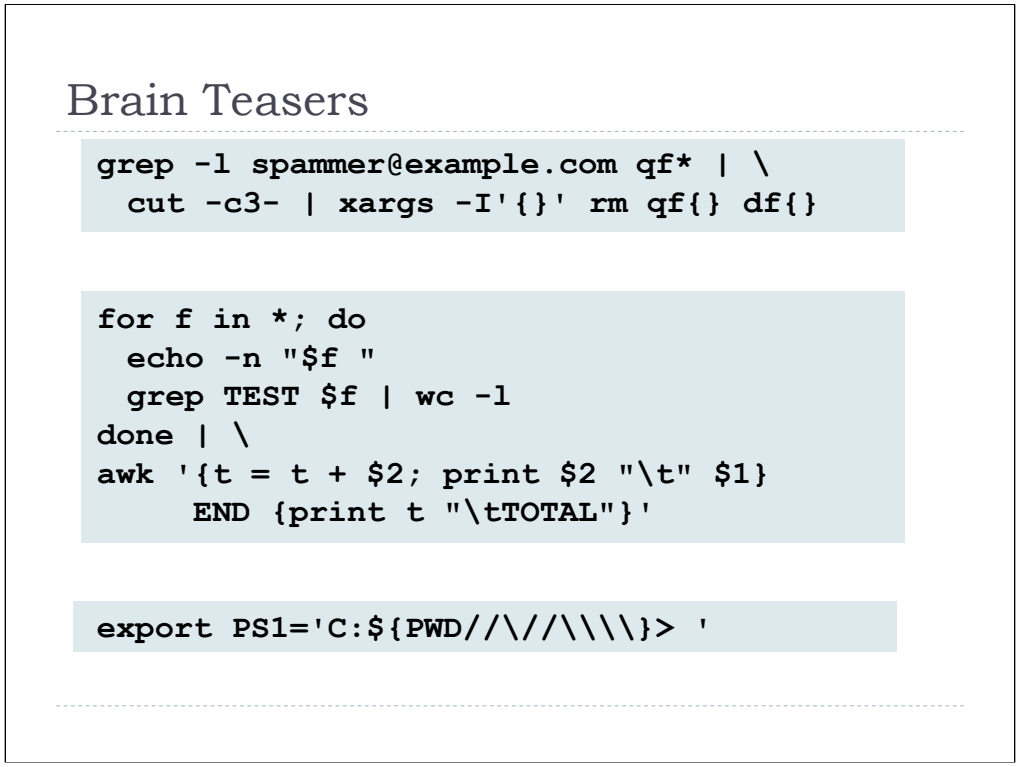
The `uniq` utility removes duplicate lines from its input. The trick is that the input needs to be sorted first, since `uniq` will only remove duplicate lines that are right next to one another in the input. So "`… | sort | uniq`" is a very common idiom– so common in fact that most versions of `sort` have a `-u` option that does the same thing as "`… | sort | uniq`". So do we really need a separate `uniq` program?

It turns out that `uniq` has a number of useful options. Perhaps the most useful is the '`-c`' flag that displays a count of the duplicate lines from its input. In the middle example on the slide we're using `awk` to pull all of the user names from the output of `ps` and piping this to "`sort | uniq -c`" to get a count of the number of processes for each user. "`sort -nr`" gives us a nice descending sort:

```
$ ps -ef | awk '{ print $1 }' | sort | uniq -c | sort -nr
     34 root
      8 apache
      7 hal
      1 UID
      1 rpc
      1 ntp
      1 mysql
      1 dbus
```

The "`1 UID`" line is a result of the initial header line from `ps`. If we wanted to get rid of that we could do something like "`ps -ef | tail +2 | awk …`", but the above is good enough for most purposes.

In the last example on the slide, we're pulling the UID values out of the `/etc/passwd` file and sending them to "`sort | uniq -c`". The last `grep` command discards any UIDs where the count from "`uniq -c`" is 1. The resulting output, therefore, is any duplicate UIDs (UIDs that appear more than once) in the `passwd` file (similar to "`logins -d`" on Unix operating systems that support the `logins` command). Since you shouldn't ever have duplicate UIDs in your password file, the output of this shell pipeline should normally be null. But obviously it's very interesting to you if the output *isn't* null.




## What Do These Do?

See if you can figure them out on your own. You can find the answers and full explanations on our Command Line Kung Fu blog:

*http://blog.commandlinekungfu.com/2009/03/episode-12-deleting-related-files.html*
*http://blog.commandlinekungfu.com/2009/06/episode-46-counting-matching-lines-in.html*
*http://blog.commandlinekungfu.com/2009/04/episode-28-environment-list.html*

## Finishing Up

- Any final questions?
- Any problems you think will stump Hal?
- Thanks for participating!
- Please fill out your surveys

http://www.deer-run.com/~hal/ (Slides)
http://blog.commandlinekungfu.com/ (More Fu)

Thank you for your time and attention. If you have any questions about the material in this presentation, here's my contact info again:


Hal Pomeranz
Deer Run Associates
PO Box 50638
Eugene, OR 97405

hal@deer-run.com
(541)683-8680
(541)683-8681 (fax)
*http://www.deer-run.com/*