Os segredos menos secretos

Bem debaixo de nossos narizes!

O Linux possui inúmeras ferramentas de linha de comando. Cada um tem a sua favorita. Para cada *cat* ou *more*, há provavelmente outros dez comandos que raramente vêem a luz do dia. Este mês, Steven Goodwin revela algumas pérolas escondidas que não merecem permanecer ocultas por mais tempo. POR STEVEN GOODWIN

Encontrar informações normalmente não é muito difícil. As páginas de manual e eu somos bons amigos. O Google também. O verdadeiro problema da informação é saber *o quê* procurar exatamente. E é aqui que entra a primeira fornada de ferramentas desconhecidas.

apropos

Esse utilitário procura em cada página de manual por uma palavra ou frase em particular. Pode ser uma palavra-chave ou uma expressão regular, sendo assim bastante completa. A ferramenta devolve então cada comando apropriado, junto com sua descrição. Assim, se você está procurando um comando de algum modo relacionado a senhas (password), mas não sabe o nome dele, basta seguir os passos da listagem 1.

Listagem 1: apropos

```
$ apropos password
afppasswd (1) - netatalk
password maintenance utility
chage (1) - change user password
expiry information
chpasswd (8) - update password
file in batch
crypt (3) - password and data
encryption
```

A opção *-e* procura apenas palavras exatas (no exemplo da listagem 1, a frase *passwords* não seria encontrada). Como você provavelmente sabe, o número entre parênteses indica em que seção das páginas de manual existe aquele comando. Veja a tabela 1 para mais detalhes. É possível invocar uma seção em particular usando, por exemplo:

```
man 1 chage
```

Isso é importante, pois alguns manuais podem existir em mais de uma seção, como um programa e uma chamada de função de mesmo nome.

whatis

whatis é um utilitário simples que mostra a descrição de um comando:

```
$whatis apropos
apropos (1) - search the manual
page names and descriptions
```

Essa é a mesma descrição que aparece na primeira linha da página de manual e o mesmo texto que aparece com o comando *apropos*, aí ao lado. *whatis* também compreende expressões regulares e com frequência é usado para conferir rapidamente um comando sem precisar formatar e exibir a página de manual inteira.

Tabela 1: Seções das páginas de manual

Seção	Descrição
1	Programas executáveis ou comandos em shell
2	Chamadas de sistema (funções realizadas pelo kernel)
3	Chamadas à biblioteca (funções dentro das bibliotecas do sistema)
4	Arquivos especiais (normalmente encontrados em */dev*)
5	Formatos de arquivo e convenções, p. ex. */etc/passwd*
6	Jogos
7	Pacotes e convenções macro, p. ex. man(7), groff(7)
8	Comandos de administração de sistema (normalmente apenas para root)
9	Rotinas do kernel [não padronizadas]

file

Antes de ler um arquivo de texto com *cat* ou *more*, é de bom tom conferir se aquilo é, realmente, texto. Deixar de fazê-lo pode fazer com que sua tela se encha de lixo e exija um comando *reset* para que seu shell seja legível novamente. Esse elegante comando nos mostra com antecedência de que tipo de arquivo se trata. É um enorme avanço sobre a utilização de extensões (que podem estar erradas ou nem existir), pois lê um pedaço do arquivo e usa o conteúdo para determinar o tipo. Por exemplo, um arquivo de texto terá marcas de parágrafo e um arquivo gzip começará com um cabeçalho específico composto pelos bytes \037 e \213.

file determina o tipo usando um arquivo de números mágicos armazenados em */usr/share/misc/magic* e */usr/share/misc/magic.mgc*. Este último é uma versão pré-compilada dos arquivos mágicos originais e é usado para aumentar a velocidade. O arquivo mágico indica quais bytes precisam aparecer em qual posição para indicar um tipo particular de arquivo. A descrição do arquivo pode ser tão complexa ou simples quanto necessário. MP3s, por exemplo, podem indicar sua taxa de bits e freqüência de gravação e formatos de imagem podem detalhar sua profundidade de cores e dimensões.

```
$ file Vexations.mp3
Vexations.mp3: MPEG 1.0 layer
3 audio stream data, 48 kBit/s,
44.1 kHz, jstereo
```

Qualquer pessoa com privilégios de superusuário pode recompilar um arquivo *magic.mgc* executando o comando a seguir:

```
$ file -C
```

which

Os comandos do Linux estão espalhados por caminhos diferentes. Meu sistema, atualmente, tem 2242 comandos em sete diretórios diferentes! (pressione a tecla *tab* duas vezes e deixe que o *tab completion* lhe mostre quantos estão disponíveis em seu computador, seguido por *echo $PATH* para ver onde procurar por eles). O trabalho do comando *which* é determinar *qual* comando é executado a partir de *qual* diretório.

```
$ which file
/usr/bin/file
```

Se uma instalação ou compilação está causando problemas, ou se você não consegue ver porque um comando em particular não está sendo chamado, o *which* indicará o comando que está sendo chamado em seu lugar. Isso também é muito útil para descobrir quando versões mais antigas (ou mais novas) estão sendo usadas de forma errônea. Também pode ser usado em conjunto com *file* para determinar se o comando prestes a ser usado é um programa, um symlink ou um script (repare nas aspas simples reversas).

```
$ file `which which`
/usr/bin/which: Bourne-Again
Shell script text executable
```

Minha taça transborda

Como tudo em Linux é tratado como arquivo, não é de surpreender que um grande número de comandos tenha sido escrito para lidar com eles. Vamos dar uma olhada em alguns...

tr

tr é uma abreviatura para a palavra *translate* (traduzir) e substitui um conjunto de caracteres por outro. Pode também remover caracteres ou pegar múltiplas ocorrências deles e substituí-las por uma única. Em todos os casos, os dados são tirados do *stdin* (a entrada padrão) e mandados ao *stdout* (a saída padrão). Com freqüência usam-se traduções para demonstrar o processamento do rot13 (*www.rot13.com/info.php*) ou para converter texto em caixa baixa. Nesses casos, ambos os conjuntos de caracteres devem ter o mesmo tamanho.

```
# podemos usar a abreviação A-Z
# ao invés do alfabeto completo
$ tr [A-Z] [a-z]  <um_arquivo_➥
qualquer> o_mesmo_arquivo_todo_➥
em_minúsculas

$ tr [a-zA-Z] [n-za-mN-ZA-M] ➥
<um_arquivo_qualquer> o_mesmo_➥
arquivo_codificado_em_rot13
```

Porém, *tr* tem muito mais utilidades. Por exemplo, pode validar nomes de arquivo em potencial transformando caracteres não-alfanuméricos em nomes de arquivo compreensíveis pelo sistema. Nesse caso, é necessária a opção -c (complemento), que significa 'tudo, exceto o seguinte'. Como não temos meio de saber quantas letras e números podem existir no conjunto complementar, podemos apenas substituí-los por um único caractere.

```
$ uname -v | tr -c [a-zA-Z0-9] _➥
_44_SMP_Sun_Dec_28_19_07_54_➥
GMT_2003_
```

Há também um atalho rápido para caracteres alfanuméricos chamado *:alnum:*, que pode ser usado no lugar do verborrágico *[a-zA-Z0-9]*.

Outro recurso do *tr* permite apagar caracteres individuais. É um modo muito fácil e rápido de remover, por exemplo, códigos espúrios de quebra de linha de arquivos de texto originados no universo Windows.

```
tr -d "\r" < windows.txt > §§
linux.txt
```

Voltando a nosso reparador de nomes de arquivo de que tratamos acima, podemos também decidir remover os caracteres problemáticos em vez de trocá-los. Dado:

```
$ uname -v | tr -c -d [:alnum:]
44SMPSunDec28190754GMT2003
```

O *Squeezing* ("aperto") é uma técnica útil para comprimir diversos caracteres repetidos em um só. Podemos usá-lo para substituir múltiplas linhas em branco num arquivo por uma única quebra de linha.

```
$ tr -s '\n' < arquivo_com_➥
linhas_vazias > arquivo_com_➥
menos_linhas_vazias
```

Note que, quando redirecionamos tanto a entrada como a saída, os nomes devem ser diferentes. Isso ocorre porque o redirecionamento trunca o arquivo de destino para zero antes que o comando seja executado. Para substituir o arquivo original, você pode incluir esse

comando em um pequeno script que use *mktemp* para gerar um nome de arquivo temporário e único.

find e xargs

Esses programas ficam muito bem juntos, mas não é impossível vê-los separados. Sozinho, o *xargs* executa um comando usando argumentos passados pela entrada padrão. Quando esses dados de entrada são redirecionados a partir de um arquivo, é muito fácil realizar processamento idêntico em muitos arquivos. Por essa razão, podemos encontrar algumas similaridades com o (mais antigo) programa *fmt*.

```
$ cat lista
alfa
beta
gama
$ xargs md5sum < lista
```

Aqui *xargs* executará o comando *md5sum* nos arquivos chamados *alfa*, *beta* e *gama*. Esses argumentos são anexados ao final da instrução *md5sum* e seriam equivalentes a

```
$ md5sum alfa beta gama
```

Porém, também é possível combinar esses argumentos em grupos de tamanho específico. Isso não apenas é bom para evitar o erro 'too many arguments', como pode ter também outros usos produtivos. Considere um arquivo chamado *copiararquivos*, parecido com isso:

```
$ cat copiararquivos
arquivo1
novoarq1
arquivo2
novoarq2
arquivo3
novoarq3
```

É muito fácil, nesse caso, agrupá-los em pares e realizar um backup rudimentar.

```
$ xargs -n 2 cp <
copiararquivos
```

o que seria equivalente a:

```
$ cp arquivo1 novoarq1
$ cp arquivo2 novoarq2
$ cp arquivo3 novoarq3
```

Para um plano de backup mais abrangente, cada arquivo pode ser copiado em outro diretório e receber uma extensão .bak. Até onde aprendemos, isso não é possível, pois os argumentos são apenas adicionados ao fim do comando. Porém, isso *é* possível graças a uma opção que nos permite colocar o argumento em qualquer lugar dentro da cadeia de caracteres.

```
$ xargs -i  cp {} backup/{}.bak
 < listabecape
```

A opção -i diz duas coisas. Primeiro, indica qual símbolo marcador é usado no lugar do argumento. Se nenhum for fornecido, o padrão será {}. Em segundo lugar, pega cada *linha* dos dados de entrada e a usa como um argumento, substituindo-a sempre que aparece {}. Não é *-n 1*, mas outra opção *-l 1*, que divide o argumento de acordo com *linhas*, não com palavras. Essa é uma importante distinção para nomes de arquivos que incluem espaços, que de outra forma seriam tratados como dois arquivos diferentes.

Menos secreta é a aliança entre *xargs* e *find*. Ao usar *find* para gerar um certo número de arquivos para a saída padrão, podemos jogá-los no *xargs* e processar cada arquivo de acordo. Uma vez que o *xargs* permite que os argumentos sejam agrupados, ele é muito mais flexível do que usar a opção *-exec* do próprio *find*. Essa combinação permite que os administradores de sistema realizem uma bela mágica, algo como ser capazes de ficar de olho em arquivos malévolos com o bit suid ligado.

```
find /home -perm +4000 | xargs
ls -l > arq.perigosos
```

Ou criar uma lista de checksums de referência para os arquivos de sistema.

```
find / -user root -type f |
xargs -l 1 md5sum
```

Por padrão, *xargs* usa um espaço para separar os nomes de arquivo. Uma vez que é possível incluir espaços nos nomes de arquivo do Linux, isso pode causar um problema (como já vimos) se um arquivo se chamar 'meu arquivo', já que o *xargs* o tratará como dois arquivos separados ('meu' e 'arquivo'). Para evitar isso, precisamos mudar o separador para um outro caractere. Tanto o *find* quanto o *xargs* têm a opção de usar o caractere NUL, evitando assim o problema.

```
find . -print0 -type f |
xargs -0 echo
```

O comando *find* é um companheiro melhor para o *xargs* do que o *ls* porque informa o caminho completo do arquivo. Para usar *ls* e *xargs* é necessário permanecer no diretório atual.

cut

cut é um programa usado para extrair colunas de dados da saída. Um uso popular desse comando é extrair informação de comandos como *ls* e *ps*, naturalmente tabulados. Por padrão, as colunas são consideradas separadas pela presença de uma tabulação. Porém, pode-se mudar isso para qualquer caracter ou símbolo arbitrário com a opção

Tabela 2: Argumentos Secretos

Argumento	Descrição
unzip -a	Converte todos os arquivos de texto para as quebras de linha no estilo Unix (sem ^M) ao descomprimir.
echo -n	Usado sozinho produzirá arquivos com tamanho zero (sem o *-n* o arquivo conterá um único caractere de quebra de linha).
cat -n	Mostra números antes de cada linha no arquivo. Útil para verificar listagens de programas. Para numerar apenas as linhas não vazias use *-b*.
tail +2	Exibe todo o arquivo a partir da segunda linha. Útil para remover os cabeçalhos de programas como *ls* e *ps*.
grep -v	Em vez de exibir todas as linhas que incluem a palavra sendo pesquisada, a opção *-v* força o grep a mostrar as linhas que NÃO contêm o padrão. Por exemplo, *grep -v grep* ignora a palavra grep na pesquisa – útil para filtar a saída do comando *ps*.
ls -1	Lista cada nome de arquivo em sua própria linha, sem o cabeçalho 'total'.

-d. Cada coluna é chamada *campo*; é possível liberar campos individuais ou grupos deles usando *-f*.

```
$ ls -l | tail +2 | tr -s ' ' | ➥
cut -d ' ' -f 3 | sort | uniq
root
estevao
```

O exemplo acima usa um pouco mais de massagem, primeiramente para remover a linha inicial de *ls* (*tail +2*, ver Tabela 2) e em seguida para comprimir os espaços entre cada campo em um só (*tr -s*). A partir daí, o *cut* simplesmente exibe o campo três (o nome do usuário) de cada arquivo do diretório. Como provavelmente há uma porção de duplicadas, devemos separar com o comando *sort* os nomes em ordem alfabética e usar *uniq*, que funciona como a compressão do *tr*, mas age sobre linhas sucessivamente idênticas.

cut pode exibir múltiplos campos aplicando a opção *-f* com uma vírgula (que dá apenas as colunas pedidas) ou um hífen (que mostra cada coluna e todas as intermediárias).

```
# Mostra a hora e a data de
# todos os arquivos
$ ls -l | tail +2 | tr -s ' ' | ➥
cut -d ' ' -f 6-8
```

strings

Esta prática ferramenta busca por strings de texto em um arquivo. Diferente do *grep*, que funciona muito bem para isolar pedaços específicos de texto, a fama do *strings* está na força bruta. Ele pode extrair todo o texto de qualquer arquivo, incluindo binários. Analisa o arquivo inteiro e exibe qualquer caractere imprimível que apareça consecutivamente com, ao menos, três outros.

Além de procurar mensagens ocultas em bibliotecas e "hackear" arquivos binários, *strings* pode ser usado, mais sensatamente, para descobrir quais são os arquivos de configuração usados por um executável.

```
$ strings `which dict` | ➥
grep conf
sysconf
-c --config <file>  specify ➥
configuration file
/etc/dict.conf
No configuration
```

Ao Serviço Secreto de Sua Majestade

Nossa fornada seguinte de ferramentas tem um alcance muito maior. Raramente as usamos encadeadas com outras ferramentas (usando como "duto" o caracter "|") ou redirecionando suas entradas e saídas (com "<" e ">"), pois elas possuem um completo (e numeroso) conjunto de instruções cada uma. Porém, não se afastam muito daquela velha filosofia de fazer uma coisa só – mas fazê-la bem.

script

O utilitário *script* tem uma grande semelhança com *tee*. Ambos pegam a entrada padrão e a retransmitem em duas "streams" diferentes de saída (a saída padrão e um arquivo). Porém, o primeiro ganhou um lugar nesta lista porque ainda se tem um shell interativo à disposição. Desse modo, você pode continuar a trabalhar e tudo o que for ecoado na tela (o que inclui os comandos, mas não as senhas, que você digitar) será armazenado num arquivo de registro para exame e estudo posterior. O *script* é muito simples de usar. Basta digitar o comando:

```
$ script
Script started, ➥
file is typescript
```

para começar e control+D (o código EOF) para terminar.

A página de manual dá um exemplo muito bom de *script* através do qual você pode permitir que uma pessoa monitore seu trabalho em tempo real. Ele faz isso com a criação de um FIFO (um "named pipe" ou "duto identificado", arquivo que permite a comunicação entre dois processos) e, enquanto o programa de script escreve informação nele, outro usuário pode lê-lo, dessa forma:

```
# Primeira pessoa
$ mkfifo watch_me; script -f ➥
watch_me
# Segunda pessoa
$ tail -f watch_me
```

screen

Esse item foi tratado mais detalhadamente em [1], [2] e [3], mas, para recapitular, *screen* permite criar diversos consoles virtuais de dentro de uma única sessão de terminal. Cada console é capaz de rodar programas, independentemente dos outros, em sua própria janela – mesmo depois de o usuário sair (log out) da sessão. Usando o *screen* você pode controlar diversos consoles com apenas uma conexão e, caso se desconecte, reconectar-se a cada um deles posteriormente.

Macaco Simão

Além dos programas listados acima, há muitos outros brindes disponíveis dentro do próprio shell bash. Pipes, redirecionamentos e substituição de comandos, para citar apenas três. Para mostrá-los todos, seria preciso escrever um livro. Destacar alguns pareceria favoritismo. Porém, antes de terminar, deixem-me mencionar dois de meus segredos favoritos...

(Parênteses)

Colocar um comando entre parênteses faz com que ele seja rodado em um subshell. Esse comando pode incluir várias instruções diferentes separadas por ponto e vírgula ou novas linhas. Isso os torna muito úteis para combinar a saída padrão de diferentes fontes.

Listagem 2: Chaves

```
# Primeiro com um subshell...
$ ( PATH="/usr/special/bin:"$PATH; special_command; )
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/games
# ...e agora sem.
$ { PATH="/usr/special/bin:"$PATH; special_command; }
$ echo $PATH
/usr/special/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/games
```

```
$ (echo “1”; echo “2”; )
1
2
```

Se for preciso combinar vários comandos, mas você não considera necessária a codificação extra de criar um subshell, pode usar chaves no lugar. Como não se cria nenhum subshell, variáveis de ambiente e caminhos de diretório podem ser modificados e também se manifestarão fora das chaves. Veja um exemplo na listagem 2.

Substituição de processo

Há um método no qual os resultados de um comando podem ser enviados a outro, como se a saída padrão fosse um arquivo. Isso pode ser representado numa escala, de forma que a saída de dois comandos diferentes pode ser enviada a um outro para mais processamento. As cadeias de saída são colocadas num arquivo temporário (o já conhecido *named pipe*), possibilitando que sejam usadas com comandos que exigem um nome de arquivo ou em lugares em que as crases não funcionam.

Portanto, para determinar a diferença que a opção *-e* faz no *apropos*, poderíamos digitar:

```
$ diff <(apropos passwd) ➥
<(apropos -e passwd)
```

Note que não há espaço entre < e (em nenhum caso.

Tocata e fuga

Neste artigo vimos uma porção de comandos muito úteis. Há mais por aí para você descobrir. Alguns são mencionados brevemente no texto (e também no quadro 1: Não apenas, mas também). Além disso, há muitos segredos escondidos que, se olharmos bem, estão debaixo de nossos narizes!

Afinal de contas, a maioria das ferramentas tem muitos argumentos de linha de comando. Muitos deles jamais serão necessários. Alguns foram provavelmente esquecidos por todos, exceto pelo autor original. Porém, há algumas jóias escondidas que merecem ser descobertas, como se pode ver na tabela 2. Você já sabe onde começar a procurar; desejo que tenha sucesso nas buscas. Divirta-se! ■

Tabela 3: Os Top 10

Comando
apropos
whatis
file
which
tr
find e xargs
cut
script
screen
strings

Quadro 1: Não apenas, mas também

Dentre os muitos programas apresentados neste artigo, há alguns a que apenas aludimos. Como tudo o mais que foi exposto aqui, você colherá melhores frutos através de suas próprias experiências. Para saber onde começar a procurar, dê uma olhada nesses comandos:

`reset`
`uname`
`mktemp`
`fmt`
`md5sum`
`sort`
`uniq`
`mkfifo`
`grep`
`dict`

INFORMAÇÃO

[1] Screen: Linux Magazine Internacional, Edição 41, página 46

[2] Screen (janelizando o modo texto – parte 1): *http://www.aurelio.net/coluna/colunao7.html*

[3] Screen (janelizando o modo texto – parte 2): *http://www.aurelio.net/coluna/colunao8.html*

[4] Curso de Shell Script – Julio Cezar Neves Linux Magazine Brasil, ed. 1 a 5.

SOBRE O AUTOR

Quando operários vão a um bar, eles falam sobre futebol. Portanto, presume-se que, quando jogadores de futebol vão a um bar, eles falem sobre os operários! Mas quando Steven Goodwin vai a um bar ele não fala nem sobre futebol, nem sobre operários. Ele, invariavelmente, fala sobre computadores...