
FAST
FOUNDATIONAL ANALYST SECURITY TRAINING
TM
Capgemini
Foundational Analyst Security Training
© Capgemini 2019. All rights reserved | 1

Module 5 – Working with the Command Line

Capgemini

Module Learning Objectives

Upon completion of this module, the student should be able to do the following:

- Understand the essential commands for moving around and working in the LINUX command line.
- Understand how to create and edit a document in the LINUX command line using text editors.
- Understand how scripting works and how it can be used to help automate routine tasks such as log analysis.

The next section will cover working with directories and files.

- Basic Linux Commands
- Input / Output (I/O) Streams
- Output Redirection
- Pipes

Topic Learning Objectives

Upon completion of this topic, the student should be able to do the following:

- Understand the basics of the LINUX command line and how to access additional information on each command.
- Use the LINUX command line to determine the location in the file structure and move around in the file structure.
- Understand how to create and move files and folders using the LINUX command line.

FAST Foundational Analyst Security Training

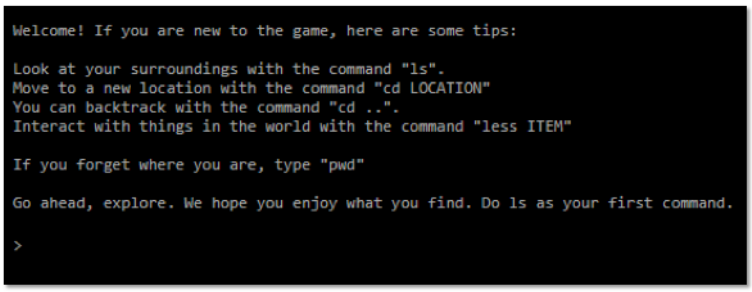
Upon successful completion of this module, students will be able to:

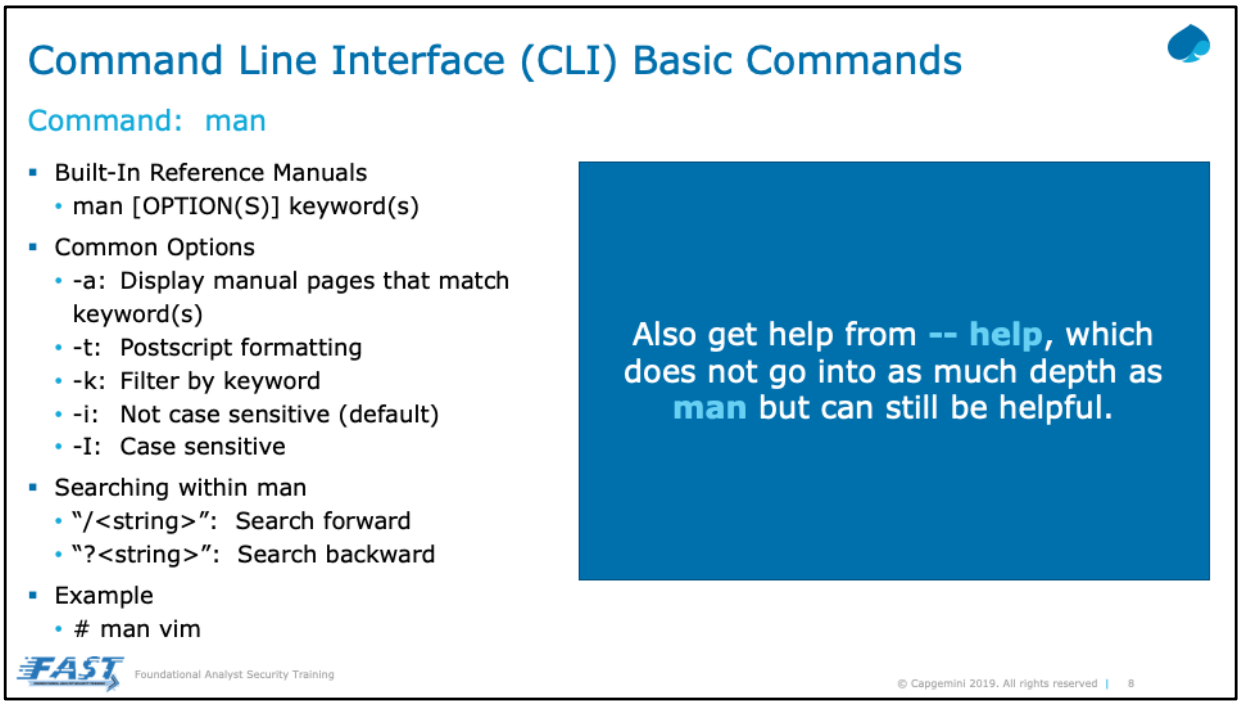
- Apply Unix tools and techniques to efficiently identify trends and extract indicators from large data sources
- Properly use basic Linux commands
- Use Linux commands for Input/Output (I/O) streams, output redirection, and pipes

Terminus by MIT…

Terminus is a flash-based, LINUX Command Line Simulator that is great to practice with!

Feel free to play with Terminus when we have downtime.





The next command is the man command.

The man command will display a built-in reference manual.

The syntax for the man command is: man [OPTION(S)] keyword(s)

Common Options include:
- -a: Display manual pages that match keyword(s)
- -t: Postscript formatting
- -k: Filter by keyword
- -i: Not case sensitive (default)
- -I: Case sensitive

You can search within the man command using:
"/<string>": to search forward
"?<string>": to search backward

An example is:
man vim

CLI Basic Commands (cont.)

Command: ls

- Lists Directory Contents and Other File Information
  - ls [OPTION(S)]…[FILE]…
- Common Options
  - -a: List all files (includes hidden files)
  - -l: Long listing format
  - -t: Sorts by file modification times
  - -h: Human readable display
  - -r: Reverse order sorting
  - -S: Sort by file size
- Examples
  - # ls –l
  - # ls –a
  - # ls -al

The ls command is used to list directory contents and other file information.

The syntax for the ls command is: ls [OPTION(S)]…[FILE]…

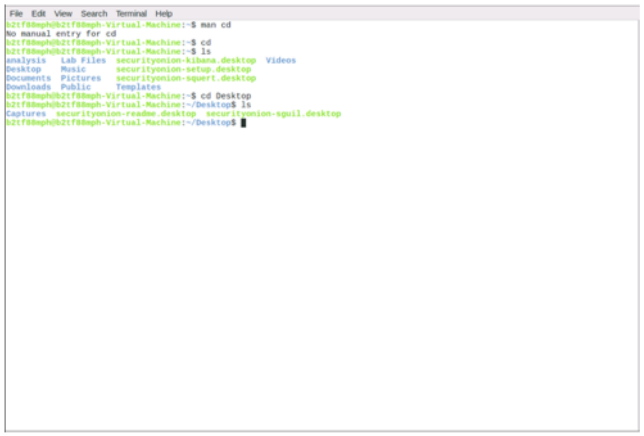
Common Options include:
- -a: List all files (includes hidden files)
- -l: Long listing format
- -t: Sorts by file modification times
- -h: Human readable display
- -r: Reverse order sorting
- -S: Sort by file size

Examples of using the ls command are:
- # ls –l to list files in a long listing format
- # ls –a to list all files including hidden files
- # ls –al to list all files in a long listing format

CLI Basic Commands (cont.)

Command: cd

- Change Directory
  - cd [OPTION(S)] [DIRECTORY]
- Common Options
  - .. : Move up one folder structure
  - ~: User's home directory
  - -P: Physical directory structure without following symbolic links
  - -L: Moves to directory the symbolic link points to
- Examples
  - # cd ..
  - # cd /documents
  - # cd ~/documents



FAST Foundational Analyst Security Training

The cd command is used to change the directory.

The syntax for the cd command is: cd [OPTION(S)] [DIRECTORY]

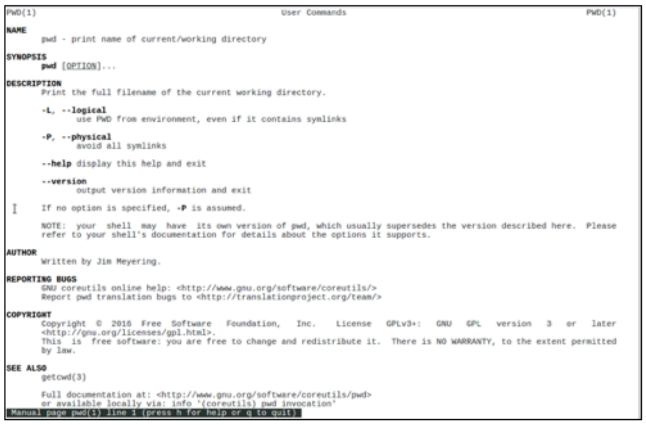
Common Options include:
- .. : Move up one folder structure
- ~ : User's home directory
- -P : Physical directory structure without following symbolic links
- -L : Moves to directory the symbolic link points to

Examples using the cd command are:
- # cd .. to move up one folder structure
- # cd /documents to go to the documents folder
- # cd ~/documents to go to the user's home directory's documents folder

CLI Basic Commands (cont.)

Command: pwd

- Print Working Directory (full filename)
  - pwd [OPTION]…
- Common Options
  - -L: Include symbolic links
  - -P: Avoid symbolic links
- Examples
  - # pwd
  - # pwd -L
  - # pwd -P



The pwd command is used to print the working directory, showing the full path.

The syntax for the pwd command is: pwd [OPTION]…

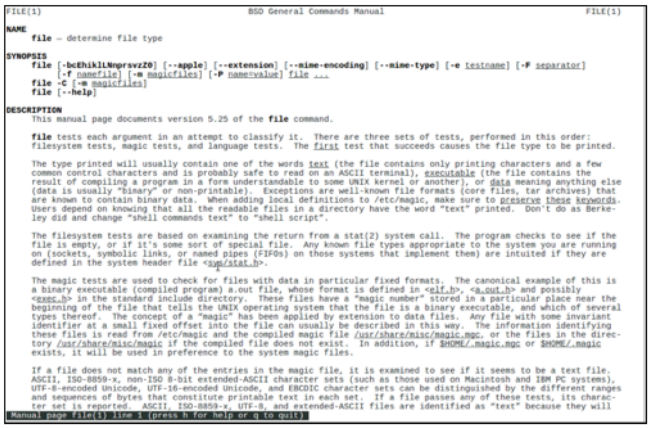
Common Options include:
- -L: Include symbolic links
- -P: Avoids symbolic links

Examples using the pwd command are:
pwd to print (or display) the working directory
pwd –L to print the working directory including symbolic links
pwd –P to print the working directory avoiding symbolic links

CLI Basic Commands (cont.)

Command: file

- Determines File Type
  - file [OPTIONS(S)] [FILE]
- Common Options
  - -b: Do not prepend filename to output line
  - -k: Do not stop at first match
- Example
  - # file audit.txt



The file command is used to determine the file type.

The syntax for the file command is: file [OPTIONS(S)] [FILE]

Common Options include:

- -b: Do not prepend filename to output line
- -k: Do not stop at first match

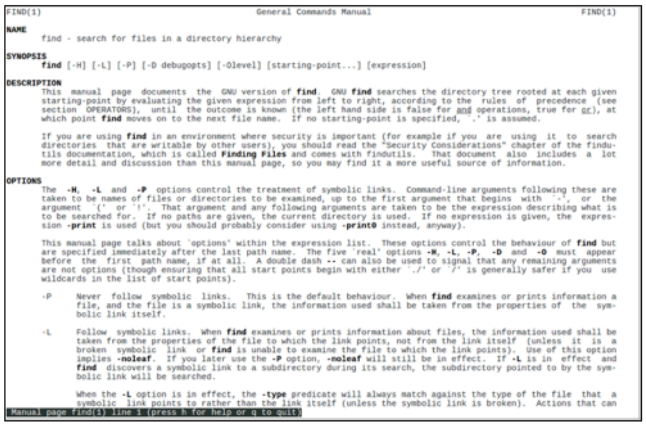
An example using the file command would be:

- # file audit.txt

CLI Basic Commands (cont.)

Command: find

- Find files, and allow command execution on results
  - find [OPTION(S)] [PATH] [EXPRESSION]
- Common Options
  - -L: Follow symbolic links
  - -P: Do not follow symbolic links (default)
  - -H: Do not follow symbolic links, except when processing the command line arguments
- Expressions
  - -name pattern: Find files with pattern in filename
  - -size [+/-]n: Find files greater than (+)/smaller than (-) n size
  - -type: Find files of a specified type (such as f = regular file, d = directory)
- Actions
  - -print: Print the path name of the current file to stdout
  - -exec <command>: Execute a command on the files found by the find search
- Example – Find files that are larger than 25 megabytes (k – kilobyte, M – Megabyte, G – Gigabyte).
  - # find / -type f -size +25M -exec ls –lh {} \;



The find command is used to find files and allow command execution on results.

The syntax for the find command is: find [OPTION(S)] [PATH] [EXPRESSION]

Common Options include:
- -L: Follow symbolic links
- -P: Do not follow symbolic links (default)
- -H: Do not follow symbolic links, except when processing the command line arguments

Additionally, there are expressions and actions that you can define with the file command.

Expressions include:
- -name pattern: Find files with pattern in filename
- -size [+/-]n: Find files greater than (+) / smaller than (-) than n size
- -type type: Find files of a specified type (such as f = regular file, d = directory)

Actions include:
- -print: Print the path name of the current file to stdout
- -exec <command>: Execute a command on the files found by the find search

An example using the file command to find files that are larger than 25 megabytes (note that k is used for kilobyte, M is used for Megabyte, and G is used for Gigabyte):
- # find / -type f -size +25M -exec ls –lh {} \;

CLI Basic Commands (cont.)

Command: chmod

- Change permissions of files or directories
  - chmod [OPTION(s)] [PERMISSIONS] [FILE]
  - Permissions can be alphanumeric or octal
- Common Options
  - -c: Verbose output only when a change is made
  - -f: Suppress most error messages
  - -R: Change files and directories recursively
  - -v: Verbose mode – output diagnostic for every file processed
- Example – Users can read, write, and execute; group can read and execute, and others can only read
  - # chmod u=rw,g=rx,o=r *.txt
  - # chmod 754 *.txt

Octal Permissions

4 – read
2 – write
1 – execute

400	read by user
040	read by group
004	read by other
200	write by user
020	write by group
002	write by others
100	execute by user
010	execute by group
001	execute by others

Common Modes

400	-r--------
644	-rw-r--r--
755	-rwxr-xr-x
777	-rwxrwxrwx

FAST Foundational Analyst Security Training

The chmod command is used to change the permissions of files or directories.

The syntax for the chmod command is: chmod [OPTION(s)] [PERMISSIONS] [FILE]

The permissions can be alphanumeric or octal.

Octal Permissions include:
- 4 – read
- 2 – write
- 1 – execute

- 400 read by user
- 040 read by group
- 004 read by other
- 200 write by user
- 020 write by group
- 002 write by others
- 100 execute by user
- 010 execute by group
- 001 execute by others

Common Modes include:
- 400 -r--------
- 644 -rw-r--r--
- 755 -rwxr-xr-x
  777 -rwxrwxrwx

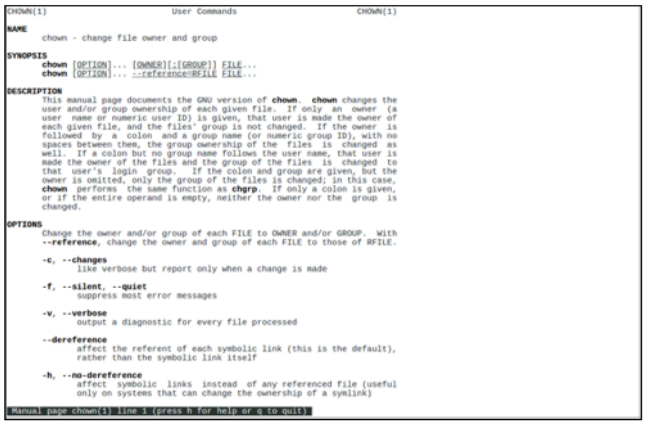
Common Options include:
- -c: Verbose output only when a change is made
- -f: Suppress most error messages
- -R: Change files and directories recursively
- -v: Verbose mode - output diagnostic for every file processed

An example of the chmod command where users can read, write, and execute; group can read and execute; and others can only read:
- # chmod u=rw,g=rx,o=r *.txt
- # chmod 754 *.txt

CLI Basic Commands (cont.)

Command: chown

- Change Ownership (files and directories)
  - chown [OPTION(S)] [FILE]
- Common Options
  - -R: Operate recursively
  - -f: Suppress most error messages
  - -v: Output diagnostic for each file
- Ownership
  - user: Name of user to own file
  - user:group: Name of user and group to own file
  - :group: Name of group to own file
- Example – Change the file ownership of "audit.log" to user "jsmith" and group "management"
  - # chown jsmith:management audit.log



The chown command is used to change Ownership of files and directories.

The syntax for the chown command is: chown [OPTION(S)] [FILE]

Common Options include:
- -R: Operate recursively
- -f: Suppress most error messages
- -v: Output diagnostic for each file

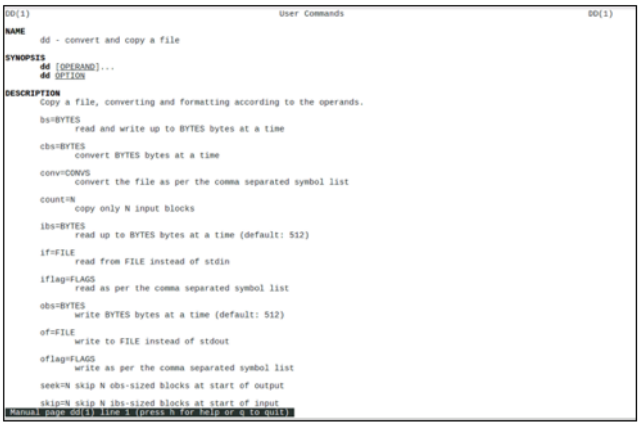
Ownership can be of a user or group:
- user: Name of user to own file
- user:group: Name of user and group to own file
- :group: Name of group to own file

An example of the chown command to change the file ownership of "audit.log" to user "jsmith" and group "management":
- # chown jsmith:management audit.log

CLI Basic Commands (cont.)

Command: dd

- Convert and copy a file
  - dd [OPTION(S)]
- Common Options
  - -if=file: Input file
  - -of=file: Output file
  - -bs=BYTES: Read and write size of blocks
  - -count=n: Number (n) of blocks to read and write
- Example – Create an image of /dev/sda in current user's home directory:
  - # dd if=/dev/sda of=~/disk1.img



The dd command is used to convert and copy a file.

The syntax for the dd command is: dd [OPTION(S)]

Common Options include:

- -if=file: Input file
- -of=file: Output file
- -bs=BYTES: Read and write size of blocks
- -count=n: Number (n) of blocks to read and write

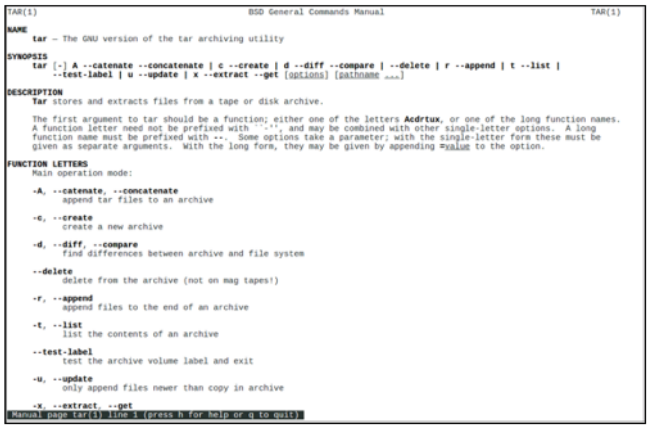
An example using the dd command to create an image of /dev/sda in current user's home directory:

- # dd if=/dev/sda of=~/disk1.img

CLI Basic Commands (cont.)

Command: tar

- Create, maintain, modify, and extract files archived in the tar format (.tar, .tar.gz, .tar.bz2)
  - tar [FUNCTION] [OPTIONS] [ARCHIVE] [FILE(S)]
- Common Functions
  - A: Append tar files to archive
  - C: Create new archive
  - --delete: Delete from archive
  - r: Append files to the end
  - u: Append files newer than the file in the archive
- Common Options
  - -c: Create an archive
  - -x: Extract an archive
  - -f: File mode
  - -v: Verbose
  - -z: gzip
  - -j: bzip2
- Example – Extract a .tar file to the current directory:
  - # tar –xvf archive1.tar



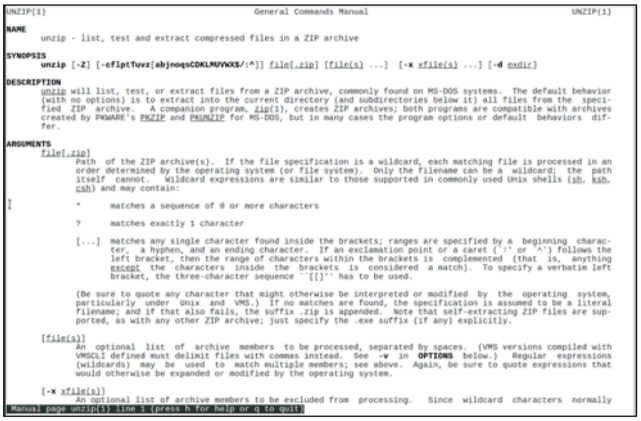
The tar command is used to create, maintain, modify, and extract files archived in the tar format (.tar, .tar.gz, .tar.bz2).

Tar files are not in themselves compressed, but can be compressed as a function of the command.

CLI Basic Commands (cont.)

Command: unzip

- List or extract a compressed .zip archive
  - unzip [OPTION(S)] [ARCHIVE] [DESTINATION]
- Common Options
  - -l: List archive files
  - -v: Verbose
  - -c: Extract files to stdout
  - -u: Update existing files (create new file if one does not exist)
- Examples
  - # unzip accounting.zip
  - # unzip –l accounting.zip



```
UNZIP(1)                    General Commands Manual                    UNZIP(1)

NAME
       unzip - list, test and extract compressed files in a ZIP archive

SYNOPSIS
       unzip [-Z] [-cflptTuvz[abjnoqsCDKLMUVWX$/:^]] file[.zip] [file(s) ...]  [-x xfile(s) ...] [-d exdir]

DESCRIPTION
       unzip will list, test, or extract files from a ZIP archive, commonly found on MS-DOS systems.  The default behavior
       (with no options) is to extract into the current directory (and subdirectories below it) all files from the  speci-
       fied  ZIP  archive.  A companion program, zip(1), creates ZIP archives; both programs are compatible with archives
       created by PKWARE's PKZIP and PKUNZIP for MS-DOS, but in many cases the program options or default  behaviors  dif-
       fer.

ARGUMENTS
       file[.zip]
              Path  of the ZIP archive(s).  If the file specification is a wildcard, each matching file is processed in an
              order determined by the operating system (or file system).  Only the filename can be a  wildcard;  the  path
              itself  cannot.  Wildcard expressions are similar to those supported in commonly used Unix shells (sh, ksh,
              csh) and may contain:

              *      matches a sequence of 0 or more characters

              ?      matches exactly 1 character

              [...]  matches any single character found inside the brackets; ranges are specified by a  beginning  charac-
                     ter,  a hyphen, and an ending character.  If an exclamation point or a caret (`!' or `^') follows the
                     left bracket, then the range of characters within the brackets is  complemented  (that  is,  anything
                     except  the  characters  inside  the  brackets  is  considered  a match).  To specify a verbatim left
                     bracket, the three-character sequence ``[[]'' has to be used.

              (Be sure to quote any character that might otherwise be interpreted or modified  by  the  operating  system,
              particularly  under  Unix  and  VMS.)  If no matches are found, the specification is assumed to be a literal
              filename; and if that also fails, the suffix .zip is appended.  Note that self-extracting ZIP files are sup-
              ported, as with any other ZIP archive; just specify the .exe suffix (if any) explicitly.

       [file(s)]
              An  optional  list  of  archive  members  to be processed, separated by spaces.  (VMS versions compiled with
              VMSCLI defined must delimit files with commas instead.  See  -v  in  OPTIONS  below.)  Regular  expressions
              (wildcards)  may  be  used  to  match multiple members; see above.  Again, be sure to quote expressions that
              would otherwise be expanded or modified by the operating system.

       [-x xfile(s)]
              An optional list of archive members to be excluded from  processing.  Since  wildcard  characters  normally
 Manual page unzip(1) line 1 (press h for help or q to quit)
```


The unzip command is used to list or extract a compressed .zip archive.

The syntax for the unzip command is: unzip [OPTION(S)] [ARCHIVE] [DESTINATION]

Common Options include:

- -l: List archive files
- -v: Verbose
- -c: Extract files to stdout
- -u: Update existing files (create new file if one does not exist)

Examples

- # unzip accounting.zip to unzip the accounting.zip file
- # unzip –l accounting.zip to unzip the accounting.zip file, using the list archive files option

CLI Basic Commands (cont.)

Command: cat/zcat

- Concatenate files and print files (zcat to compress or extract)
  - cat [OPTION(S)] [filename]
  - zcat [OPTION(S)] [compressed filename]
- Common Options
  - -n: Number output lines
  - -b: Same as –n, except to ignore blank lines
  - -s: Skip files not found
- Examples
  - # cat File1.txt
  - # cat File1.txt File2.txt > Files1_2.txt

Use > to overwrite and >> to append.

FAST Foundational Analyst Security Training

The cat/zcat commands are used to concatenate files and print files (use zcat to compress or extract).

The syntax for the cat/zcat command are:

- cat [OPTION(S)] [filename]
- zcat [OPTION(S)] [compressed filename]

Common Options include:

- -n: Number output lines
- -b: Same as –n, except to ignore blank lines
- -s: Skip files not found

Examples using the cat command include:

- # cat File1.txt to print file1.txt
- # cat File1.txt File2.txt > Files1_2.txt to concatenate file1.txt and file2.txt and send the output to files1_2.txt

NOTE: use > to overwrite and >> to append

CLI Basic Commands (cont.)

Command: diff

- Find the difference between two files
  - diff File1 File2
- Example
  - # diff File1.txt File2.txt

The diff command is used to Fnd the difference between two files.

The syntax for the diff command is: diff File1 File2

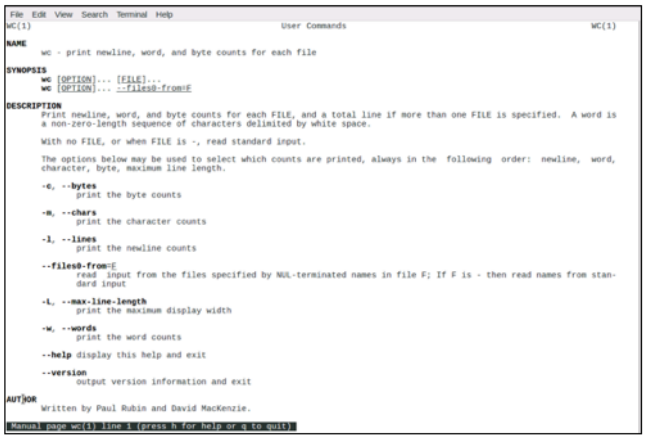
An example using the diff command:

- # diff File1.txt File2.txt

CLI Basic Commands (cont.)

Command: wc

- Print the number of new lines, words, and/or bytes in a file
  - wc [OPTION(S)] [FILENAME]
- Common Options
  - -c: Print byte count
  - -m:Print character count
  - -l: Print new line count
  - -w: Print word count
- Example
  - # wc -lwc File1.txt



The wc command is used to print the number of newlines, words, and/or bytes in a file.

The syntax for the wc command is: wc [OPTION(S)] [FILENAME]

Common Options include:
- -c: Print byte count
- -m: Print character count
- -l: Print newline count
- -w: Print word count

Am example using the wc command to print the newline count, word count, and byte count:
- # wc -lwc File1.txt

CLI Basic Commands (cont.)

Command: grep/egrep

- Processes text line by line, and prints any lines that match a specified pattern
  - grep [OPTIONS] pattern [FILE]
  - egrep [OPTIONS] pattern [FILE]
- Common Options
  - -f <file>: Obtain patterns from <file>
  - -i: Ignore case
  - -v: Invert match (non-matching lines)
  - -w: Select whole word matches
  - -x: Exactly match whole line
  - -c: Print count of matching lines
- Examples
  - # grep "10.170." ./datafile
  - # egrep "10.170.[2-3]" ./datafile

The grep/egrep commands are used to processes text line by line and prints any lines which match a specified pattern.

The syntax for the grep/egrep commands are:
- grep [OPTIONS] pattern [FILE]
- egrep [OPTIONS] pattern [FILE]

Common Options include:
- -f <file>: Obtain patterns from <file>
- -i: Ignore case
- -v: Invert match (non-matching lines)
- -w: Select whole word matches
- -x: Exactly match whole line
- -c: Print count of matching lines

Examples using the grep/egrep commands are:
- # grep "10.170." ./datafile
- # egrep "10.170.[2-3]" ./datafile

CLI Basic Commands (cont.)

Command: strings

- Print strings of printable characters in files
  - strings [OPTION(S)] [FILENAME]
- Common Options
  - -a: Scan the whole file
  - -e: Select the character encoding of the strings to be found
  - -n: Change minimum length
  - -T: Specify object code format
- Example – Search /bin/cat for strings contain "Accounting"
  - # strings /bin/cat | grep Accounting

The strings command is used to print strings of printable characters in files.

The syntax for the strings command is: strings [OPTION(S)] [FILENAME]

Common Options include:
- -a: Scan the whole file
- -e: Select the character encoding of the strings to be found
- -n: Change minimum length
- -T: Specify object code format

Am example using the strings command to search /bin/cat for strings containing "Accounting":
- # strings /bin/cat | grep Accounting

CLI Basic Commands (cont.)

Command: sort

- Sort contents of a file, line by line
  - sort [OPTION(S)] [FILE]
- Common Options
  - -b: Ignore leading blanks
  - -f: Ignore case
  - -r: Reverse sort results
  - -t[DELIM]: Use DELIM as a field delimiter
  - --sort=WORD: Sort according to WORD: general-numeric -g, human-numeric -h, month -M, numeric -n, random -R, version –V
- Example: Sort a four-column data set on the third field in reverse order
  - # sort –r +2 –3 File1 > File2
- Example: Sort a four-column data set on the fourth field, and ignore case
  - # sort –t, –f +3 –4 File1 > File2

FAST Foundational Analyst Security Training

The sort command is used to sort contents of a file, line by line. Good Example: IP Addresses

The syntax for the sort command is: sort [OPTION(S)] [FILE]

Common Options include:

- -b: Ignore leading blanks
- -f: Ignore case
- -r: Reverse sort results
- -t[DELIM]: Use DELIM as a field delimiter
- --sort=WORD: Sort according to WORD: general-numeric -g, human-numeric -h, month -M, numeric -n, random -R, version –V

An example using the sort command to sort a 4 column data set on the 3rd field in reverse order:

- # sort –r +2 –3 File1 > File2

An example using the sort command to sort a 4 column data set on the 4th field and ignore case:

- # sort –t, –f +3 –4 File1 > File2

CLI Basic Commands (cont.)

Command: uniq

- Print or filter out repeated lines.
  - uniq [OPTIONS] [FILE]
- Common Options
  - -i: Ignore case
  - -u: Only print unique lines
  - -d: Only show multiples
  - -c: Count the number of occurrences
- Example
  - # uniq –c File1

The uniq command is used to print or filter out repeated lines.

The reason for the unique command is the -c option. Note you MUST run sort before unique, you do not need any flags with sort.

The syntax for the uniq command is: uniq [OPTIONS] [FILE]

Common Options include:

- -i: Ignore case
- -u: Only print unique lines
- -d: Only show multiples
- -c: Count the number of occurrences

An example using the uniq command, using the option to count the number of occurrences:

- # uniq –c File1

CLI Basic Commands (cont.)

Command: xargs

- Build and execute command lines from standard input
  - xargs [OPTIONS] command [initial arguments]
- Common Options
  - -a: Read from file
  - -d[DELIM]: Set delimiter character
  - -e: Set End of File (EOF) string
  - -n: Max arguments, execute
  - command for every n args
- Example – Run a Network Mapper (Nmap) scan on a list of Internet Protocol (IP) addresses
  - # cat FinanceIPs | xargs –n1 nmap -sV

FAST Foundational Analyst Security Training

The xargs command is used to build and execute command lines from standard input.

The syntax for the xargs command is: xargs [OPTIONS] command [initial arguments]

Common Options include:

- -a: Read from file
- -d[DELIM]: Set delimiter character
- -e: Set EOF (end of file) string
- -n: Max arguments, execute command for every n args

An example using the cat command combined with the xargs command to run an nmap scan on a list of Ips:

- # cat FinanceIPs | xargs –n1 nmap -sV

CLI Basic Commands (cont.)

Command: cut

- Cut out selected portions of each line using a delimiter
  - cut [OPTION(S)] [FILE]
- Common Options
  - -d: Specify delimiter (tab is default)
  - -c: Specify character position
  - -f: Specify field by number
  - -s: Suppress lines when no delimiter
- Example – Extract fields 1 and 7 from /etc/passwd
  - # cut –d : -f 1,7 /etc/passwd

The cut command is used to cut out selected portions of each line using a delimiter.

The syntax for the cut command is: cut [OPTION(S)] [FILE]

Common Options include:

- -d: Specify delimiter (tab is default)
- -c: Specify character position
- -f: Specify field by number
- -s: Suppress lines when no delimiter

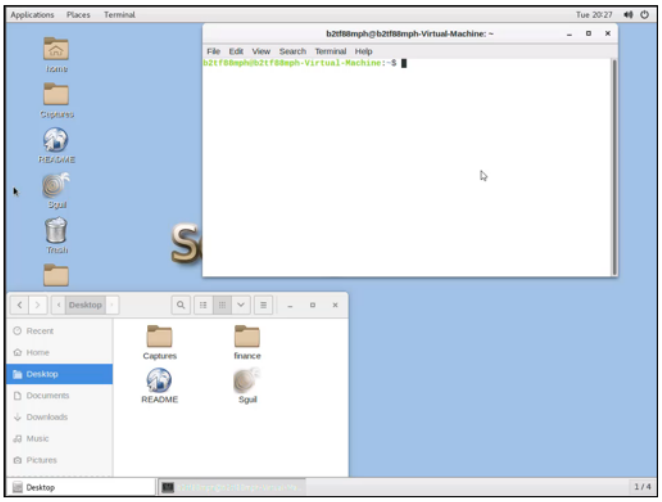
An example using the cut command to extract fields 1 and 7 from /etc/passwd

- # cut –d : -f 1,7 /etc/passwd

Working with Directories

Make and Remove Directories

- The pwd command prints the working directory
  - # pwd
- The cd command, without a pathname, changes to the home directory
  - # cd
- The mkdir command makes a new directory
  - # mkdir Accounting
- The rmdir command removes a directory
  - # rmdir Accounting
- The cd command changes to the directory
  - # cd /finance



FAST Foundational Analyst Security Training

Click the image to play the video

There are a few commands commonly used when working with directories.

The mkdir command makes a new directory, for example
mkdir Accounting

The rmdir command removes a directory, for example
rmdir Accounting

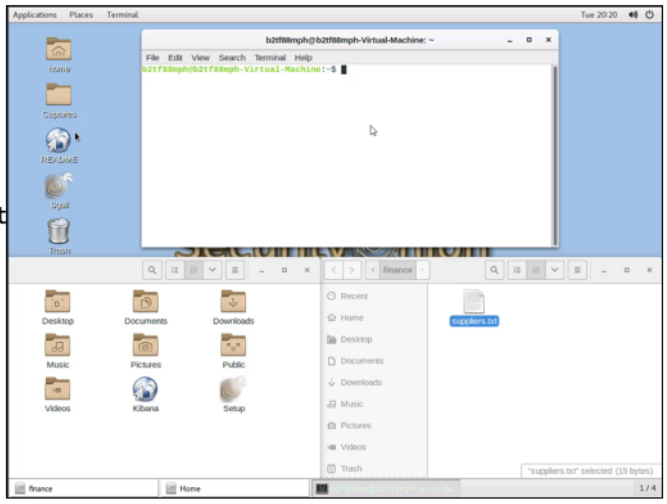
The cd command changes to the directory, for example
cd /finance

The cd command without a pathname changes to your home directory, for example
cd

The pwd command prints the working directory, for example
pwd

Moving Files

- Users have the permissions to create or write files to their home directory
- The ~ character saves files to the home directory
  - # File1 > ~/File1.txt
- The mv command moves files between directories
  - # mv finance/suppliers.txt ~suppliers.txt
- The rm command removes files or directories
  - rm [-i] [-r] File1 File 2 …
    - -i Inquire before removing
    - -r Recursively remove a directory and all its contents and subdirectories
      - # rm file1.txt file2.txt
- The cp command copies files between directories
  - # cp ~/indicators.txt ~/solutions/indicators/



FAST Foundational Analyst Security Training

Click the image to play the video

There are a few commands commonly used when moving files.

Users have the permissions to create or write files to their home directory.

The ~ character saves files to your home directory, for example
File1 > ~/File1.txt

The mv command moves files between directories, for example
mv /finance/suppliers.txt ~/suppliers.txt

The rm command removes files or directories, for example
rm [-i] [-r] File1 File 2 …
-i Inquire before removing
-r Recursively remove a directory and all its contents and subdirectories

An example using the rm command is:
rm file1.txt file2.txt

The cp command copies files to between directories, for example
cp ~/indicators.txt ~/solutions/indicators/

LINUX is Case Sensitive, Sensitive in General!

This is a very important aspect to LINUX…

When the command does not appear to be working, check to make sure that the commands are correct, spacing is correct, and the correct characters were used.

90 percent of the time, it will resolve the issue!

CLI Basic Commands

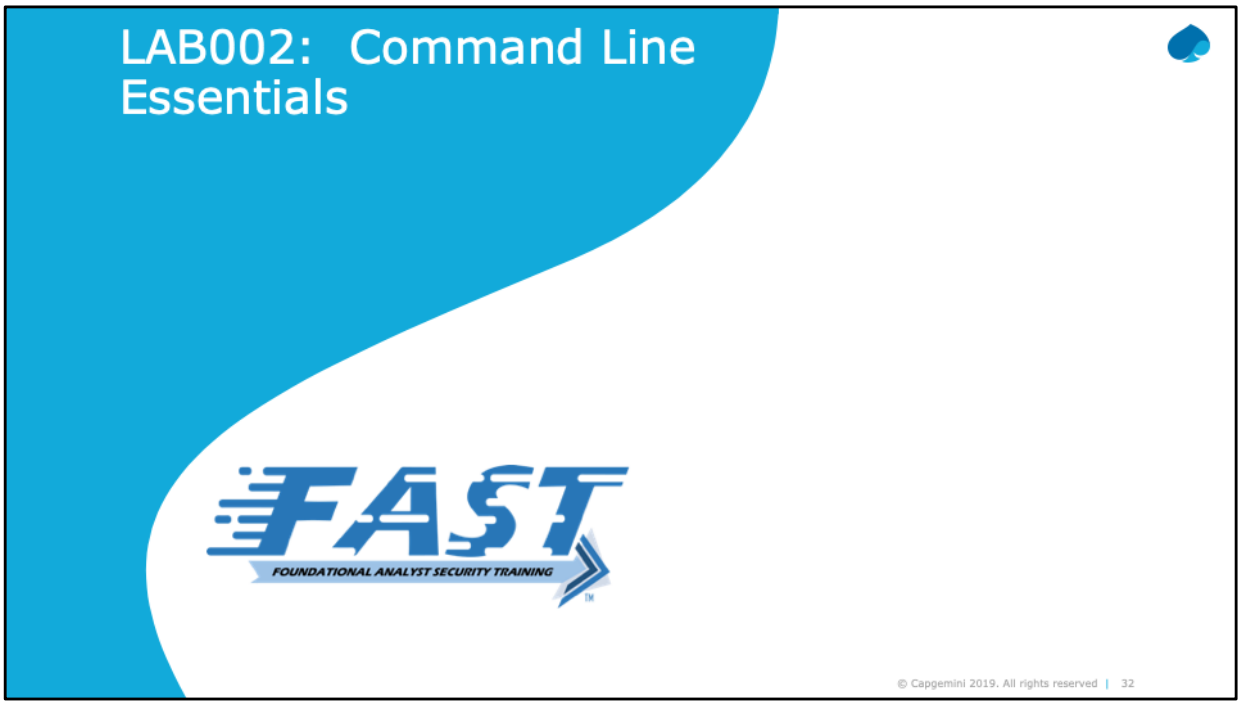
There are many more commands available in Linux. Explore the other commands available; listed here are a few additional commands.

Command	Description
passwd	Change password
paste	Display contents of files side by side
merge	Three-way file merge
join	Join lines of two files with a common field
lpr	Send to printer
touch	Change file timestamps
date	System's time and date information

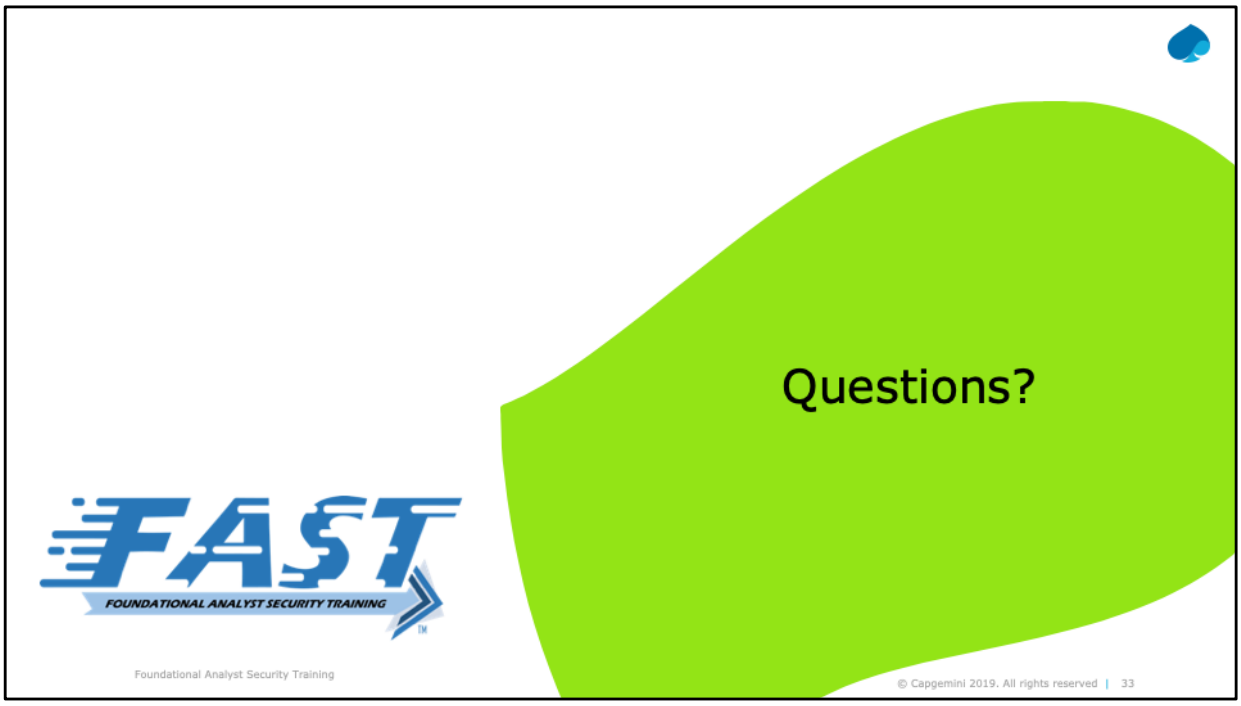
FAST Foundational Analyst Security Training

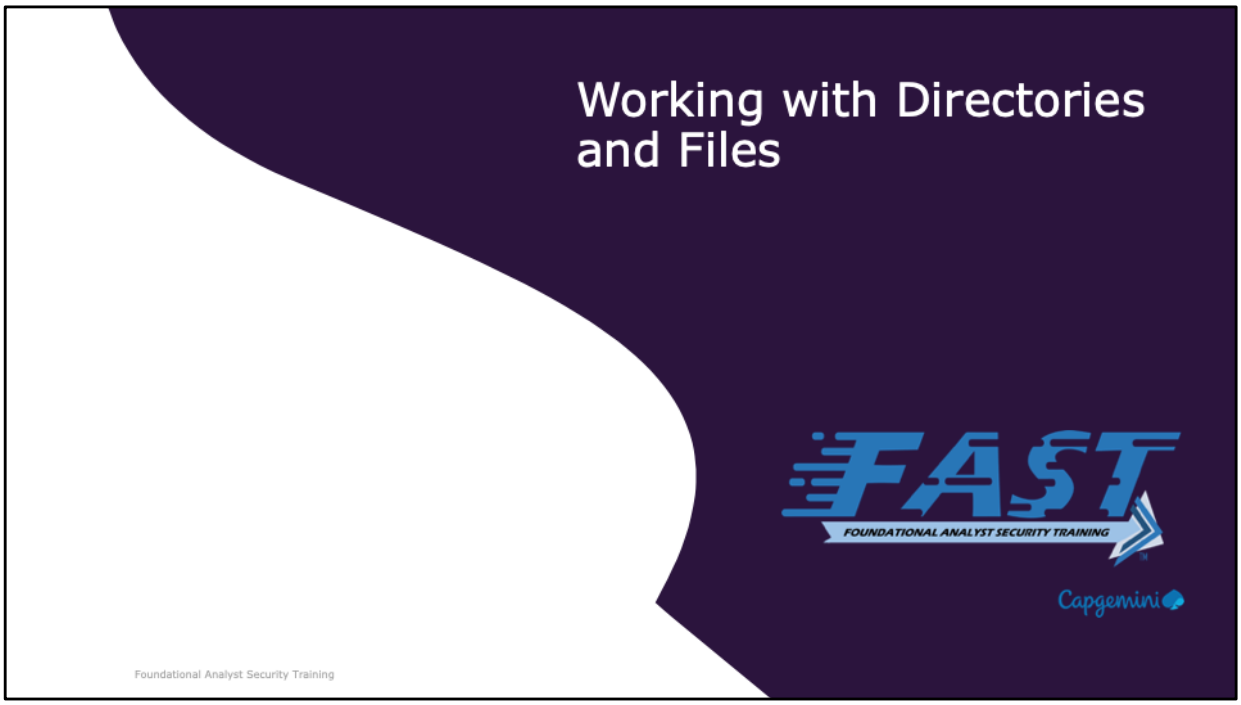
There are many more commands available in Linux. Explore the other commands available; listed below are a few additional commands:

passwd	to change the password
paste	to display contents of files side-by-side
merge	to do a three way file merge
join	to join lines of two files with a common field
lpr	to send to printer
touch	to change file timestamps
date	to display system's time and date information



Content Source:


Questions?
FAST
FOUNDATIONAL ANALYST SECURITY TRAINING
Foundational Analyst Security Training
© Capgemini 2019. All rights reserved | 33



The next section will cover working with directories and files.

The topics covered in this module include:

- vi/vim
- Regular Expressions
- gawk
- sed

Topic Learning Objectives

Upon completion of this topic, the student should be able to do the following:

- Understand how data is output to streams, Tees, and Pipes in LINUX and how this can be used in file analysis.

I/O Streams

Linux receives and sends output in character streams.

- The three standard I/O streams are as follows:
  1. stdout: Standard command output, which is usually to the screen
  2. stdin: Standard command input, which is usually the keyboard
  3. stderr: Standard error output codes from commands

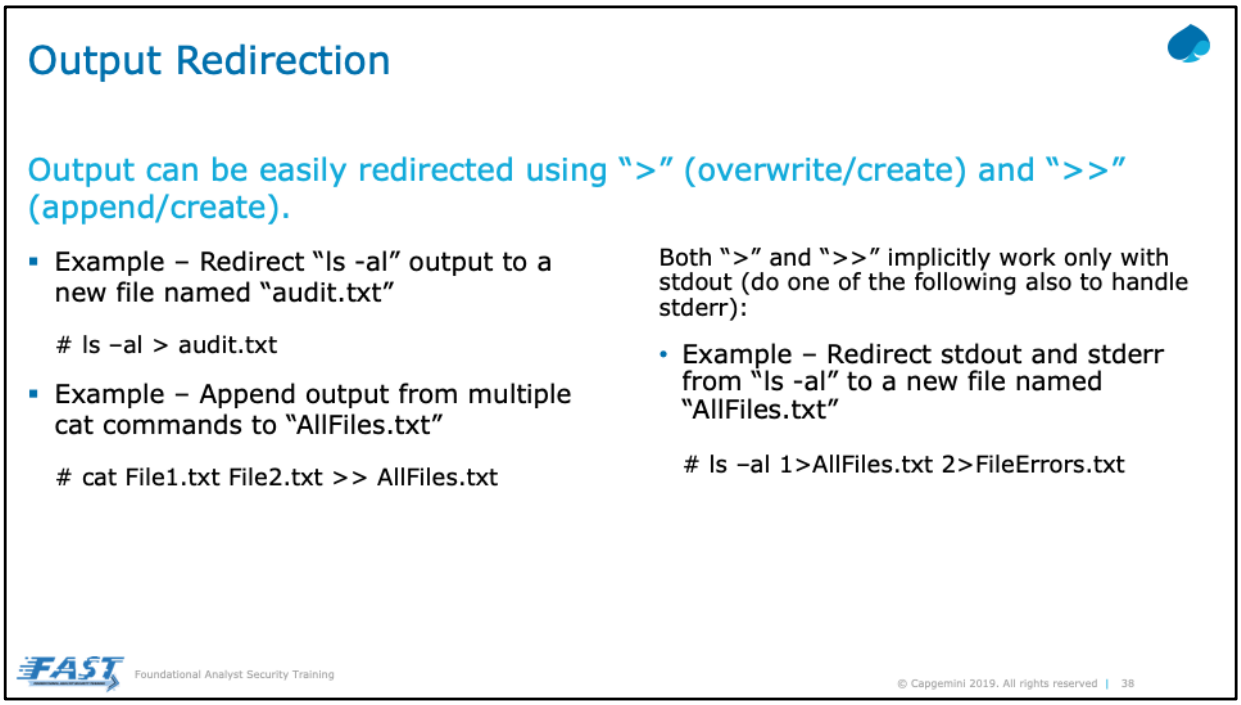
In a Terminal session, stdout and stderr are both output by default; change this with redirection.

FAST Foundational Analyst Security Training

Linux receives and sends output in character streams.

There are three standard I/O streams:
- stdout: Standard command output, which is usually to the screen
- stdin: Standard command input, which is usually the keyboard
- stderr: Standard error output codes from commands

NOTE: In a Terminal session, stdout and stderr are both output by default; you can change this with redirection.



Output can be easily redirected using “>” (overwrite/create) and “>>” (append/create).
An example to redirect “ls -al” output to a new file named “audit.txt” is:
ls –al > audit.txt

An example to append output from multiple cat commands to “AllFiles.txt” is:
cat File1.txt File2.txt >> AllFiles.txt

Both “>” and “>>” implicitly work only with stdout (you must do one of the following to also handle stderr)

An example to redirect stdout and stderr from “ls -al” to a new file named “AllFiles.txt” is:
ls –al 1>AllFiles.txt 2>FileErrors.txt

An example to suppress all output from a command is:
gedit AllFiles.txt 1>/dev/null 2>/dev/null

Basic Pipes

A pipe is a form of redirection to transfer standard output to another destination.

- Combine multiple commands, and redirect the stdout of one command to the stdin of the second.

  Uses "|" to delineate commands

  command1 | command2 | command3 | ...

- Example – Display directory contents in long format, and paginate the results.

  # ls –al | more

- Example – Use ls and find commands to list and print all lines matching a pattern.

  # ls -l | find ./ -type f -name "*.txt" -exec grep "program" {} \;

Foundational Analyst Security Training

A pipe is a form of redirection to transfer standard output to another destination (combine multiple commands and redirect the stdout of one command to the stdin of the second); it uses "|" to delineate commands, such as:
command1 | command2 | command3 | ...

An example to display directory contents in long format and paginate the results is:
ls –al | more

An example to use ls and find commands to list and print all lines matching a pattern is:
ls -l | find ./ -type f -name "*.txt" -exec grep "program" {} \;

Tees

A tee in Linux allows the following actions:

- Save the output from a command to a file.
- Output the results (to the screen or another pipe).

Example

- The filetype outputs are written to file1.txt, which will be displayed in the user's home directory and the terminal.

```
# ls –al | file -f - | tee ~/file1.txt
```

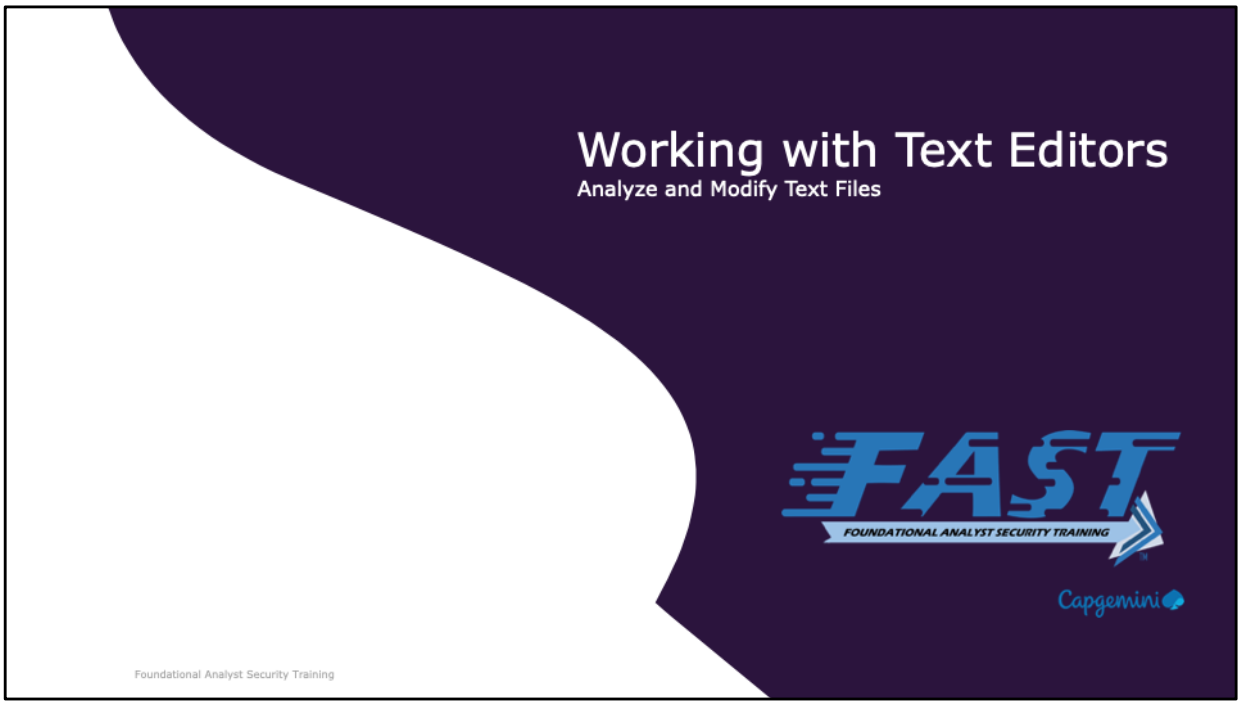
FAST Foundational Analyst Security Training

A tee in Linux allows you to save the output from a command to a file, and output the results (to the screen or another pipe).

An example where the filetype outputs are written to file1.txt, which will be displayed in the user's home directory and the terminal:
ls –al | file -f - | tee ~/file1.txt

Capgemini

Questions?



In this module, we will cover Working with Text Editors.

The topics covered in this module include:

- Basic Scripting Concepts
- Bash Shell
- Other Scripting Languages

Topic Learning Objectives

Upon completion of this topic, the student should be able to do the following:

- Understand what a text editor is and how to use it to create and edit a file in LINUX.
- Understand how to use Global Regular Expression Print (grep), Regex, gawk, xargs, and sed to manipulate data in LINUX.

FAST Foundational Analyst Security Training

 | 44

Upon successful completion of this module, student will be able to:

- Edit text files using vi and vim
- Use regular expressions (regexes) for string matching, parsing, and replacement
- Use gawk for field extraction, sorting, unique, and counts
- Use sed to search and replace

Text Editors

- It is often necessary to use built-in Linux command-line text editors.
  - "vi" provides basic text editing capabilities.
  - "vim" provides additional capabilities.
- What is vi/vim?
  - The default *nix editor is called vi (visual editor).
  - The vi editor has two modes:
    1. Command mode causes action to be taken on the file.
    2. Insert mode inserts text into the file.
  - Pressing the <Esc> key turns off the Insert mode.
  - To enter vi, type vi filename.

If the filename exists, it opens the file; if the filename does not exist, it creates a new file.

You will often need to use built-in Linux command line text editors. "vi" provides basic text editing capabilities, while "vim" provides additional capabilities.

The default editor is called vi, which stands for visual editor. The vi editor has the Command mode, causes action to be taken on the file; and it has the Insert mode, which inserts text into the file.

Pressing the <Esc> key turns off the Insert mode.

To enter vi, type vi followed by the filename.

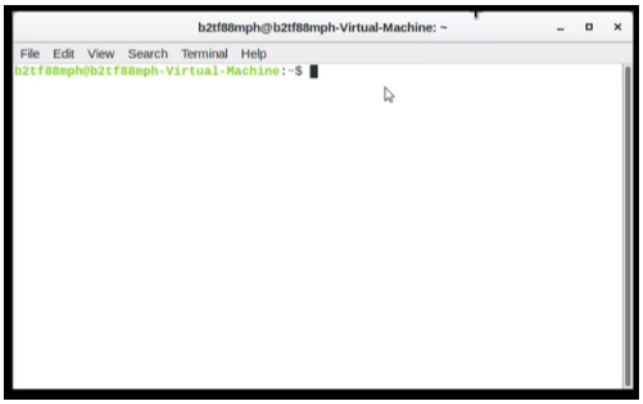
NOTE: If the filename exists, it opens the file; if the filename does not exist, it creates a new file.

Text Editors (cont.)

Opening, Closing, and Saving

- To enter vi, type vi filename.
- NOTE: If the filename exists, it opens the file; if the filename does not exist, it creates a new file.
- To quit, type a ":" which moves the cursor to the bottom of the screen.
- Enter the desired command, followed by "Enter":

:w	quit vi, write to disk
:w filename	quit vi, writing to filename
:wq!	quit vi, write to disk, override read only
q!	quit vi, do not save changes

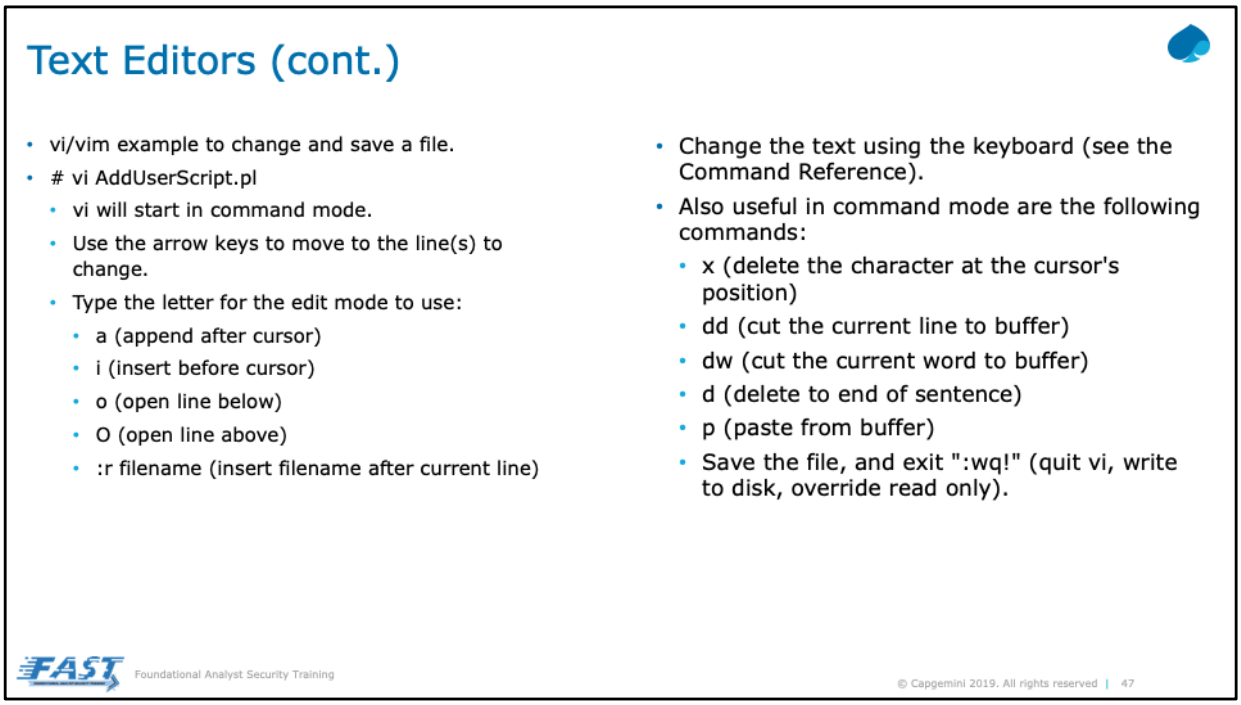


FAST Foundational Analyst Security Training

As previously mentioned, to enter vi, type vi followed by the filename.

To quit, type a ":" which moves the cursor to the bottom of the screen. Then, enter the desired command followed by "Enter":

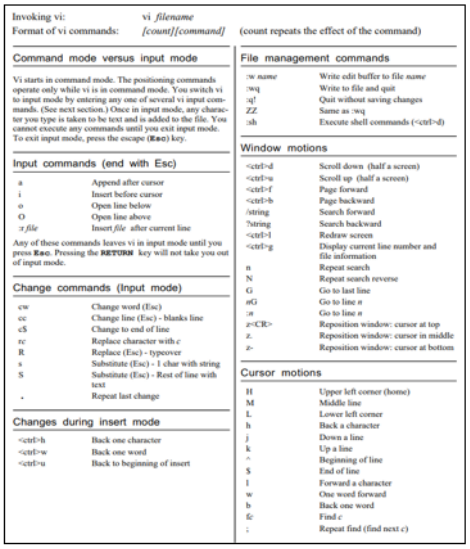
- :w quit vi, write to disk
- :w filename quit vi, writing to filename
- :wq! quit vi, write to disk, override read only
- :q! quit vi, do not save changes



The following example sequence would change and save a file:

- Enter vi followed by the filename, such as AddUserScript.pl.
- vi will start in command mode
- Then you can use the arrow keys to move to the line(s) you want to change.
- Type the letter for the edit mode you want to use, such as:
    - a (append after cursor)
    - i (insert before cursor)
    - o (open line below)
    - O (open line above)
    - :r filename (insert filename after current line)
- Change the text as needed, using the keyboard (see the Command Reference on the next slide – or many that exist online).
- Also useful in command mode are the commands:
    - x (delete the character at the cursor's position)
    - dd (cut the current line to buffer)
    - dw (cut the current word to buffer)
    - d) (delete to end of sentence)
    - p (paste from buffer)
- Save the file and exit ":wq!" (quit vi, write to disk, override read only).

vi/vim Command Reference



Invoking vi: vi *filename*
Format of vi commands: *[count][command]* (count repeats the effect of the command)

Command mode versus input mode

Vi starts in command mode. The positioning commands operate only while vi is in command mode. You switch vi to input mode by entering any one of several vi input commands. (See next section.) Once in input mode, any character you type is taken to be text and is added to the file. You cannot execute any commands until you exit input mode. To exit input mode, press the escape (**Esc**) key.

Input commands (end with Esc)

a	Append after cursor
i	Insert before cursor
o	Open line below
O	Open line above
:r *file*	Insert *file* after current line

Any of these commands leaves vi in input mode until you press **Esc**. Pressing the **RETURN** key will not take you out of input mode.

Change commands (Input mode)

cw	Change word (Esc)
cc	Change line (Esc) - blanks line
c$	Change to end of line
rc	Replace character with *c*
R	Replace (Esc) - typeover
s	Substitute (Esc) - 1 char with string
S	Substitute (Esc) - Rest of line with text
.	Repeat last change

Changes during insert mode

<ctrl>h	Back one character
<ctrl>w	Back one word
<ctrl>u	Back to beginning of insert

File management commands

:w *name*	Write edit buffer to file *name*
:wq	Write to file and quit
:q!	Quit without saving changes
ZZ	Same as :wq
:sh	Execute shell commands (<ctrl>d)

Window motions

<ctrl>d	Scroll down (half a screen)
<ctrl>u	Scroll up (half a screen)
<ctrl>f	Page forward
<ctrl>b	Page backward
/string	Search forward
?string	Search backward
<ctrl>l	Redraw screen
<ctrl>g	Display current line number and file information
n	Repeat search
N	Repeat search reverse
G	Go to last line
*n*G	Go to line *n*
:*n*	Go to line *n*
z<CR>	Reposition window: cursor at top
z.	Reposition window: cursor in middle
z-	Reposition window: cursor at bottom

Cursor motions

H	Upper left corner (home)
M	Middle line
L	Lower left corner
h	Back a character
j	Down a line
k	Up a line
^	Beginning of line
$	End of line
l	Forward a character
w	One word forward
b	Back one word
f*c*	Find *c*
;	Repeat find (find next *c*)

Deletion commands

:dd or :*n*dd	Delete *n* lines to general buffer
dw	Delete word to general buffer
d*n*w	Delete *n* words
d)	Delete to end of sentence
db	Delete previous word
D	Delete to end of line
x	Delete character

Recovering deletions

p	Put general buffer after cursor
P	Put general buffer before cursor

Undo commands

u	Undo last change
U	Undo all changes on line

Rearrangement commands

yy or Y	Yank (copy) line to general buffer
"*z*nyy	Yank 6 lines to buffer *z*
yw	Yank word to general buffer
"*a*9dd	Delete 9 lines to buffer *a*
"*A*9dd	Delete 9 lines; Append to buffer *a*
"*a*p	Put text from buffer *a* after cursor
p	Put general buffer after cursor
P	Put general buffer before cursor
J	Join lines

Parameters

:set list	Show invisible characters
:set nolist	Don't show invisible characters
:set number	Show line numbers
:set nonumber	Don't show line numbers
:set autoindent	Indent after carriage return
:set noautoindent	Turn off autoindent
:set showmatch	Show matching sets of parentheses as they are typed
:set noshowmatch	Turn off showmatch
:set showmode	Display mode on last line of screen
:set noshowmode	Turn off showmode
:set all	Show values of all possible parameters

Move text from file *old* to file *new*

vi *old*	
"a10yy	yank 10 lines to buffer *a*
:w	write work buffer
:e *new*	edit new file
"ap	put text from *a* after cursor
:30,60w *new*	Write lines 30 to 60 in file *new*

Regular expressions (search strings)

^	Matches beginning of line
$	Matches end of line
.	Matches any single character
*	Matches any previous character
.*	Matches any character

Search and replace commands

Syntax:

```
:[address]s/old_text/new_text/
```

Address components:

.	Current line
n	Line number *n*
.+*m*	Current line plus *m* lines
$	Last line
/string/	A line that contains "string"
%	Entire file
[addr1],[addr2]	Specifies a range

Examples:

The following example replaces only the **first** occurrence of Banana with Kumquat in each of 11 lines starting with the current line (.) and continuing for the 10 that follow (.+10).

```
:.,.+10s/Banana/Kumquat
```

The following example replaces **every** occurrence (caused by the `g` at the end of the command) of `apple` with `pear`.

```
:%s/apple/pear/g
```

The following example removes the last character from every line in the file. Use it if every line in the file ends with `^M` as the result of a file transfer. Execute it when the cursor is on the first line of the file.

```
:%s/.$//
```

This cheat sheet will be available in the laboratory environment, when needed.

This slide depicts common commands. Likewise, if you search online, you will see many command reference that you can view and/or download.

Xargs

- The "xargs" command is used to execute an additional Linux command on every line from standard input.
  - This allows performing operations on a list via command line execution.
  - Almost always receives information from a pipe.
  - The Linux command to run on each line is placed after "xargs."

Example…

"ls -l" output creates a list of files in the current directory.

- # ls –l
- document1.txt
- important_file.doc
- config.sys

From this output, use piping and "xargs" to perform the Linux command "rm" on each file passed to STDIN from the pipe to delete the files.

- # ls –l | xargs rm

Text Files – Regular Expressions (Regexes)

- Used for string matching, parsing, and replacement
- Especially helpful to match a known pattern (such as dates, IP addresses, Uniform Resource Locators [URLs], etc.)
- Regular grep supports basic Regex (use egrep or grep –e for extended support).

FAST Foundational Analyst Security Training

Regular expressions (regexes) are used for string matching, parsing, and replacement, which is very common when analyzing files, such as log files.

They are especially helpful to match a known pattern (such as dates, IP addresses, URLs, etc).

Regular grep supports basic regex; use egrep or grep –e for extended support.

Text Files – Regex Quantifiers

Specify characters to match.

- Search for each instance of the word "audit" (-i for not case sensitive).

  # grep -i "audit" file1

- Search for every line that does not contain the word "audit.‟ "-v" will invert the match (find those that do not match).

  # grep -v "audit" file1

- Search for the number of matches ("-n" for the number, or count, of matches).

  # grep -n "audit" file1

- "^" to match at the beginning of the line

  # grep "^audit" file1

- "$" to match at the end of the line

  # grep "audit$" file1

FAST Foundational Analyst Security Training

This slide shows some examples using the grep command to match characters in a file.

To search for each instance of the word "audit" (-i for not case sensitive).
grep -i "audit" file1

To search for every line that does not contain the word "audit“. "-v" will invert the match (find those that do not match)
grep -v "audit" file1

To search for the number of matches. "-n" for the number, or count, of matches.
grep -n "audit" file1

Use "^" to match at the beginning of the line.
grep "^audit" file1

Use "$" to match at the end of the line.
grep "audit$" Ffile1

Text Files – Regex Quantifiers (cont.)

Specify how many instances of a character, group, or character class to match.

- Only apply to the character, group, or character class that immediately precedes it.
- The quantifiers only apply to the character "c" in the following examples:
  - "*": matches character before it zero or more times; match "abcxyz," "abxyz," "abccxyz"

    # egrep "abc*xyz" file1
  - "?": matches zero or one time; matches "abcxyz," "abxyz" but does not match "abccxyz"

    # egrep "abc?xyz" file1
  - "+": matches one or more; matches "abcxyz," "abccxyz" but does not match "abxyz"

    # egrep "abc+xyz" file1

FAST Foundational Analyst Security Training

 | 52

As previously mentioned, egrep provides extended capabilities.

You can specify how many instances of a character, group, or character class to match. It only apply to the character, group, or character class that immediately precedes it.

The quantifiers only apply to the character "c" in the following examples.

The "*" is commonly used to match character before it zero or more times. **# egrep "abc*xyz" file1** will match "abcxyz", "abxyz", "abccxyzg"

The "?" is commonly used to match zero or one time. **# egrep "abc?xyz" file1** will match "abcxyz", "abxyz", but does not match "abccxyz"

The "+" is commonly used to match one or more. **# egrep "abc+xyz" file1** will match "abcxyz", "abccxyz", but does not match "abxyz"

Text Files – Regex Character Classes

Use square brackets ("[ ]") to group characters.

- Match any of characters of "1," "2," and "3"

  # grep "[123]" file1

- Match "hood" or "food"

  # grep "[hf]ood" file1

- Match any characters between "a" and "z"

  # grep "[a-z]" file1

- Match lines that begin with a capital letter

  # grep "^[A-Z]" file1

FAST Foundational Analyst Security Training

You can also use square brackets ("[]") to group characters, as depicted in these examples.

To match any of characters of "1", "2", and "3"
grep "[123]" file1

To match "good" or "hood"
grep "[hf]ood" file1

To match any characters between "a" and "z"
grep "[a-z]" file1

To match lines that begin with a capital letter
grep "^[A-Z]" file1

Text Files – Regex Character Classes (cont.)

Use ^ inside the brackets to invert the results.

- Match anything other than characters "1," "2," or "3"

```
# grep "[^123]" file1
```

Use curly braces ("{ }") to specify the number of occurrences of the preceding character.

- Match at least three occurrences of a digit character and no more than six occurrences.

```
# egrep "[0-9]{3,6}" file1
```

This applies to extended grep and Perl-Compatible Regular Expression (PCRE) only.

FAST Foundational Analyst Security Training

Use ^ inside the brackets to invert the results; for example, to match anything other than characters "1", "2", or "3":
grep "[^123]" file1

Use curly braces ("{}") to specify the number of occurrences of the preceding character. Note: This applies to extended grep and PCRE only. To match at least 3 occurrences of a digit character, and no more than 6 occurrences:
egrep "[0-9]{3,6}" file1

Text Files – Regex Logic (Or)

Use the pipe character ("|") inside the quotes for the "OR" condition.

- Match "good" or "well" in the file1.
- # egrep "good|well" file1
- Match "good" or "gold" in file1.
- # grep "go(od|ld)" file1

Use multiple grep statements, separated by pipes, to search for two or more patterns (AND condition). Match "pattern1" in file1; and then, from that result, match "pattern2":

- # grep "pattern1" file1 | grep "pattern2"

Use the pipe character ("|") inside the quotes for the "OR" condition. For example, to match "good" or "well" in file1:
egrep "good|well" file1

To match "good" or "gold" in file1:
grep "go(od|ld)" file1

Use multiple grep statements separated by pipes to search for two or more patterns, applying the AND condition. Match "pattern1" in file1 then from that result match "pattern2":
grep "pattern1" file1 | grep "pattern2"

Text Files – Regex Logic (Not)

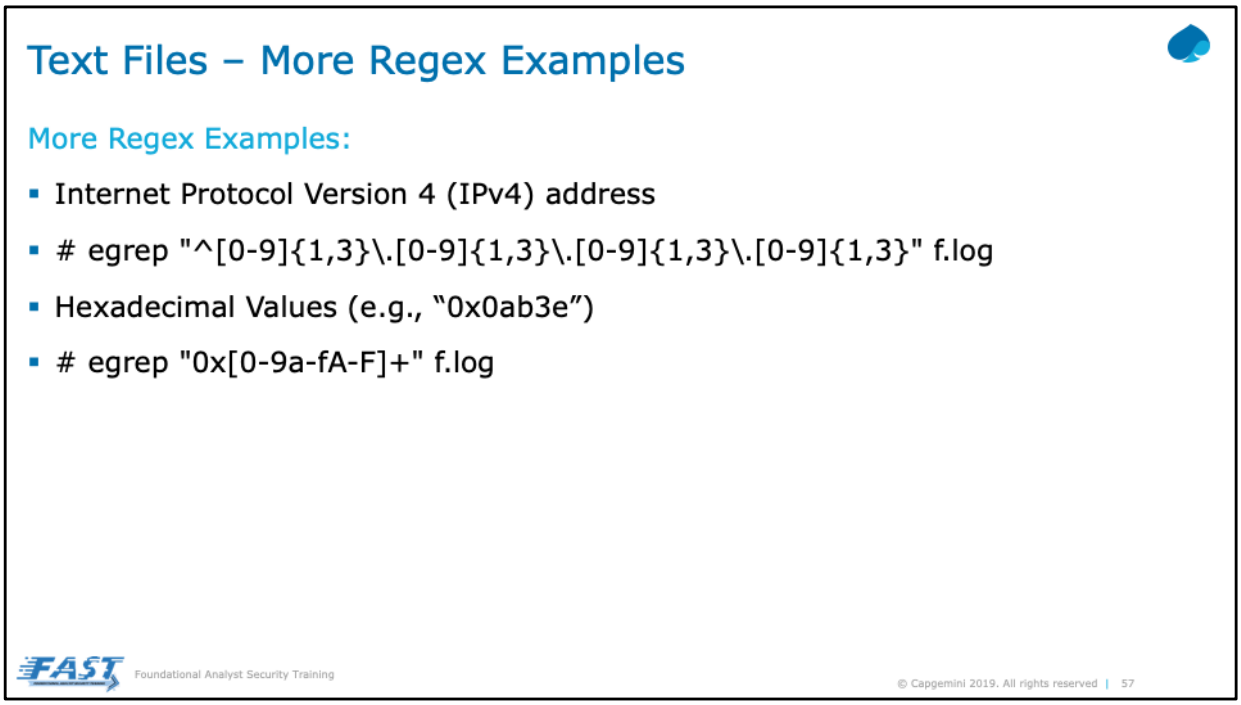
Escaping Meta-Characters

- Meta-Characters have special meaning; thus, to search for those, such as a literal period or a literal opening bracket, "escape" these characters to ignore their special meaning.
- "\" before the character escapes the special meaning of the character.
- Search for lines that begin with a capital letter and end with a period:
- # grep "^[A-Z].*\.$" file1

FAST Foundational Analyst Security Training

 | 56

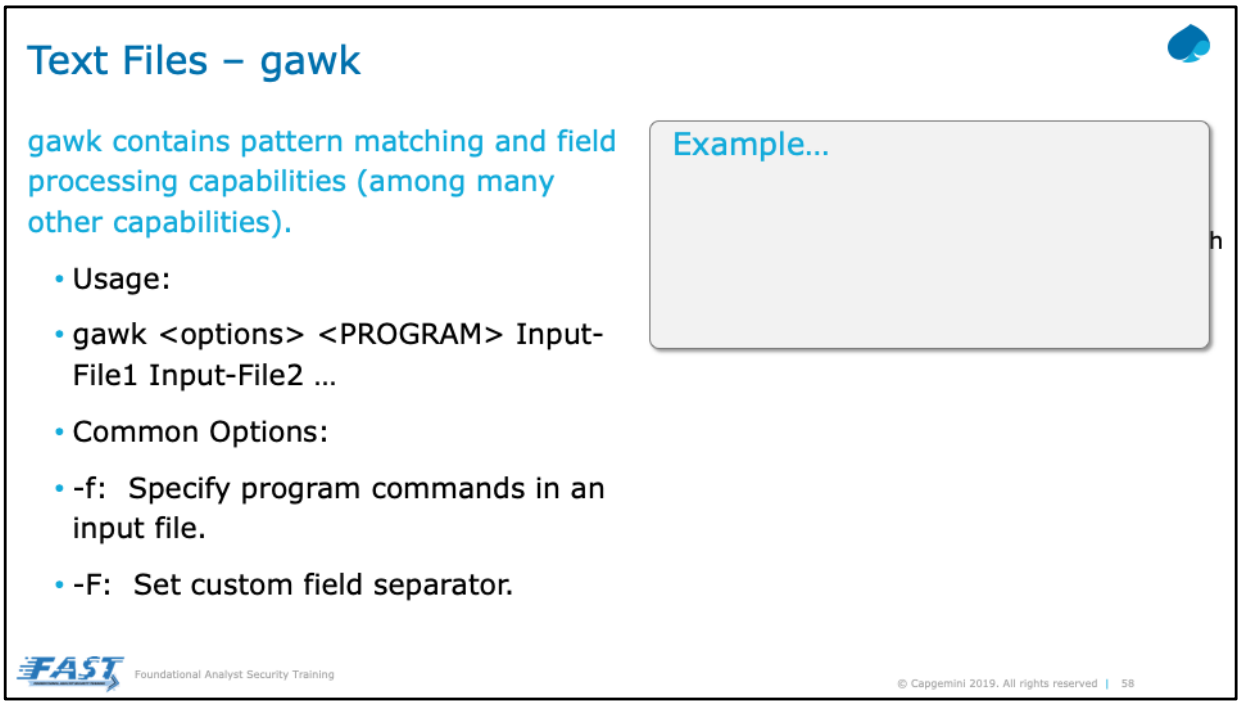
There are meta characters that have special meaning; thus, to search for those, such as a literal period or a literal opening bracket, you need to "escape" these characters to ignore their special meaning. Using the "\" before the character escapes the special meaning of the character. For example, to search for lines that begins with a capital letter and ends with a period:
grep "^[A-Z].*\.$" File1



A couple other egrep examples to search for IP addresses or hexadecimal values are as follows:

An example to search IPv4 addresses is:
egrep "^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" f.log

An example to search Hexadecimal Values, such as “0x0ab3e”, is:
egrep "0x[0-9a-fA-F]+" f.log



gawk contains pattern matching and field processing capabilities (among many other capabilities). The syntax for using gawk is:
gawk <options> <PROGRAM> Input-File1 Input-File2 …

Common options include:
- -f: Specify program commands in an input file
- -F: Set custom field separator

An example using gawk to find all lines in file1 that start with a string between 2011 and 2019 is:
- # gawk '/^/201[1-9]/' file1

Text Files – gawk (cont.)

Field Extraction and Fields

- Print the fifth field in every line.

  # gawk '{print $5}' /var/log/file1

- Change the field delimiter to a colon + space, and print the fifth field.

  # gawk –F ': ' '{print $5}' /var/log/file1

- Print the first and second field separated by a forward slash.

  # gawk '{print $1 "/" $2}' /var/log/file1

- Print the number of fields in each line.

  # gawk '{print NF}' /var/log/file1

Gawk examples for Field Extraction and working with Fields is depicted on this slide.

To print the fifth field in every line:
gawk '{print $5}' /var/log/file1

To change the field delimiter to a colon + space, and print the fifth field:
gawk –F ': ' '{print $5}' /var/log/file1

To print the first and second field separated by a forward slash:
gawk '{print $1 "/" $2}' /var/log/file1

To print the number of fields in each line:
gawk '{print NF}' /var/log/file1

Text Files – gawk (cont.)

Field Extraction and Fields

- Print every line that has at least five fields.

  # gawk 'NF>=5{print}' /var/log/file1

- Print out the last field ($NF) in each matching line.

  # gawk '{print $NF}' /var/log/file1

- Print the second-to-last field in every line.

  # gawk '{print $(NF-1)}' /var/log/file1

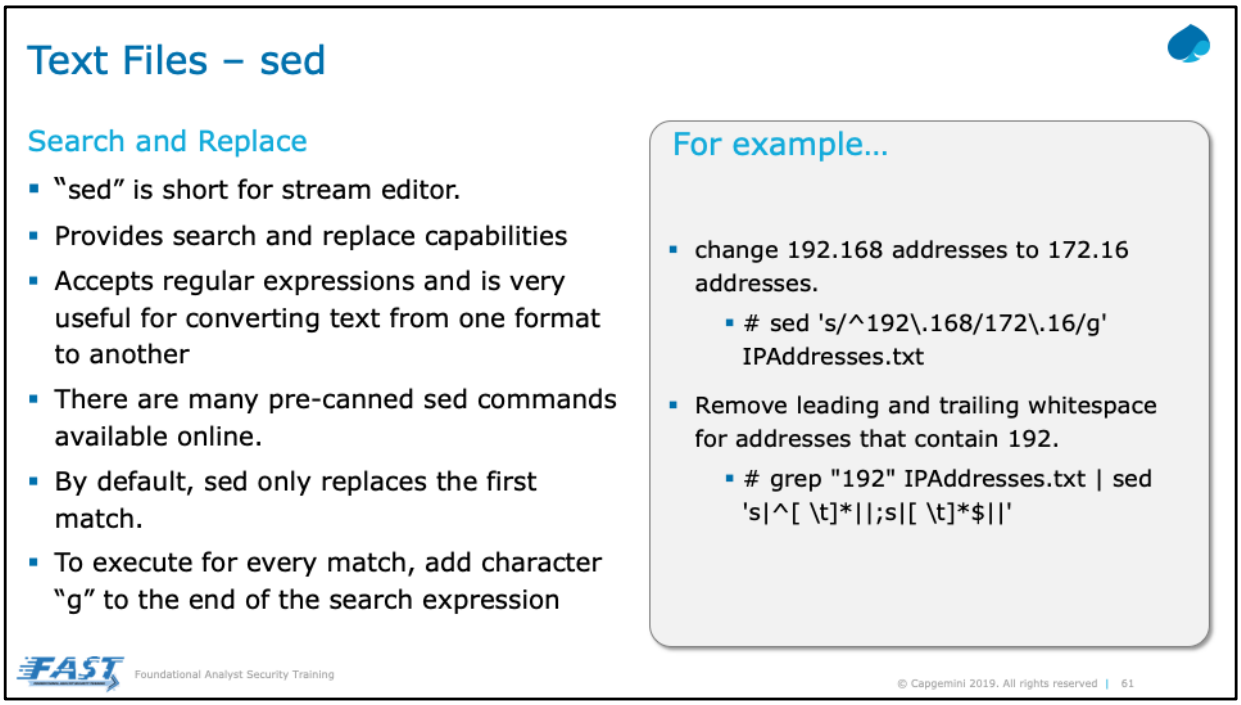
FAST Foundational Analyst Security Training

More examples of Field Extraction and working with Fields include:

To print every line that has at least 5 fields:
gawk 'NF>=5{print}' /var/log/file1

To print out the last field ($NF) in each matching line:
gawk '{print $NF}' /var/log/file1

To print the second-to-last field in every line:
gawk '{print $(NF-1)}' /var/log/file1



Sed is a commonly used command for searching and replacing.

"sed" is short for "stream editor, and it provides search and replace capabilities.

Sed accepts regular expressions and is very useful for converting text from one format to another. Additionally there are many pre-canned sed commands available online.

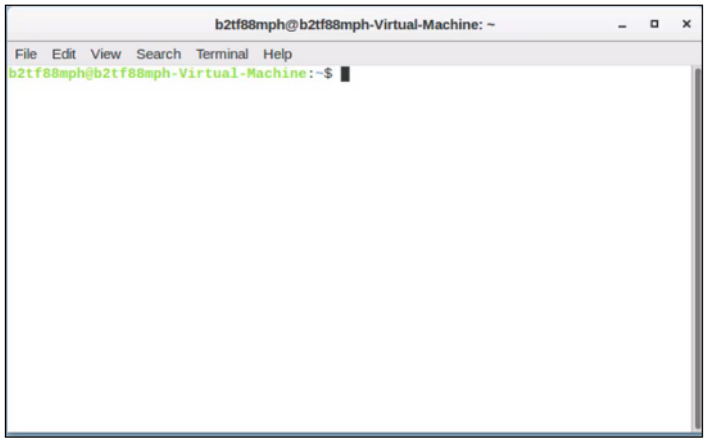
By default, sed only replaces the first match. To execute for every match, add the character "g" to the end of the search expression.

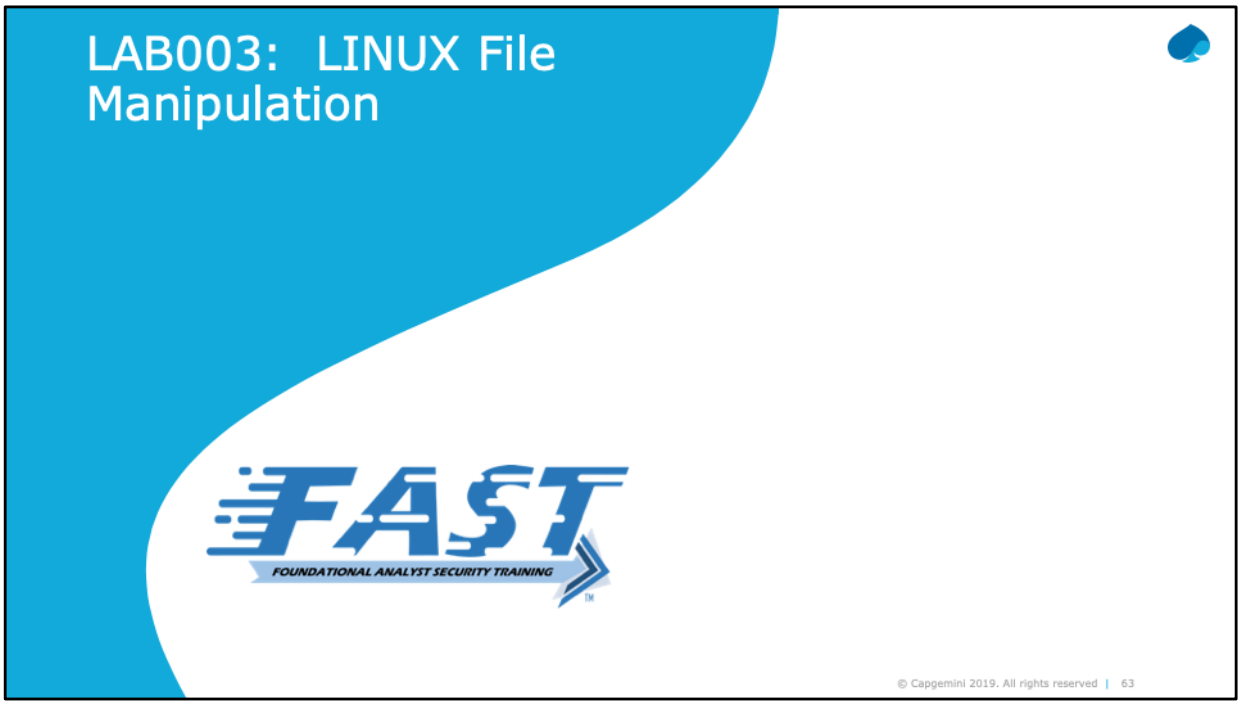
For example, to change 192.168 addresses to 172.16 addresses:
sed 's/^192\.168/172\.16/g' IPAddresses.txt

To remove leading and trailing whitespace for addresses that contain 192:
grep "192" IPAddresses.txt | sed 's|^[\t]*||;s|[\t]*$||'

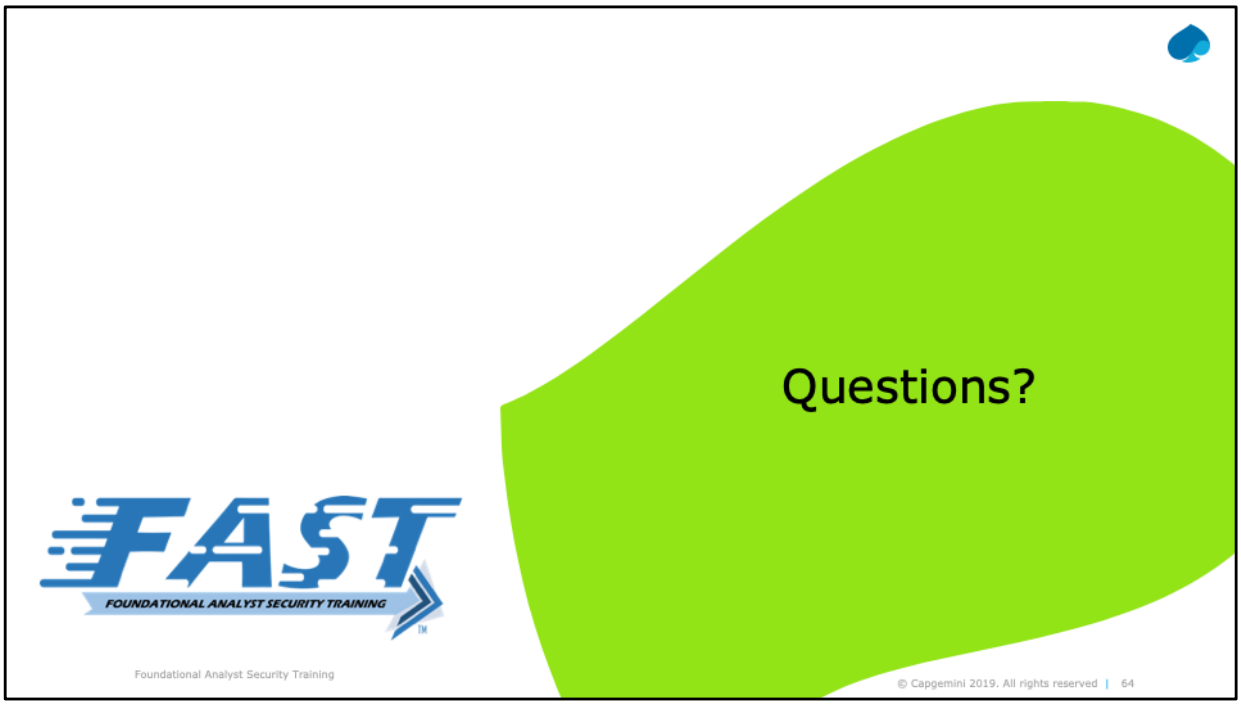
Text Files – grep

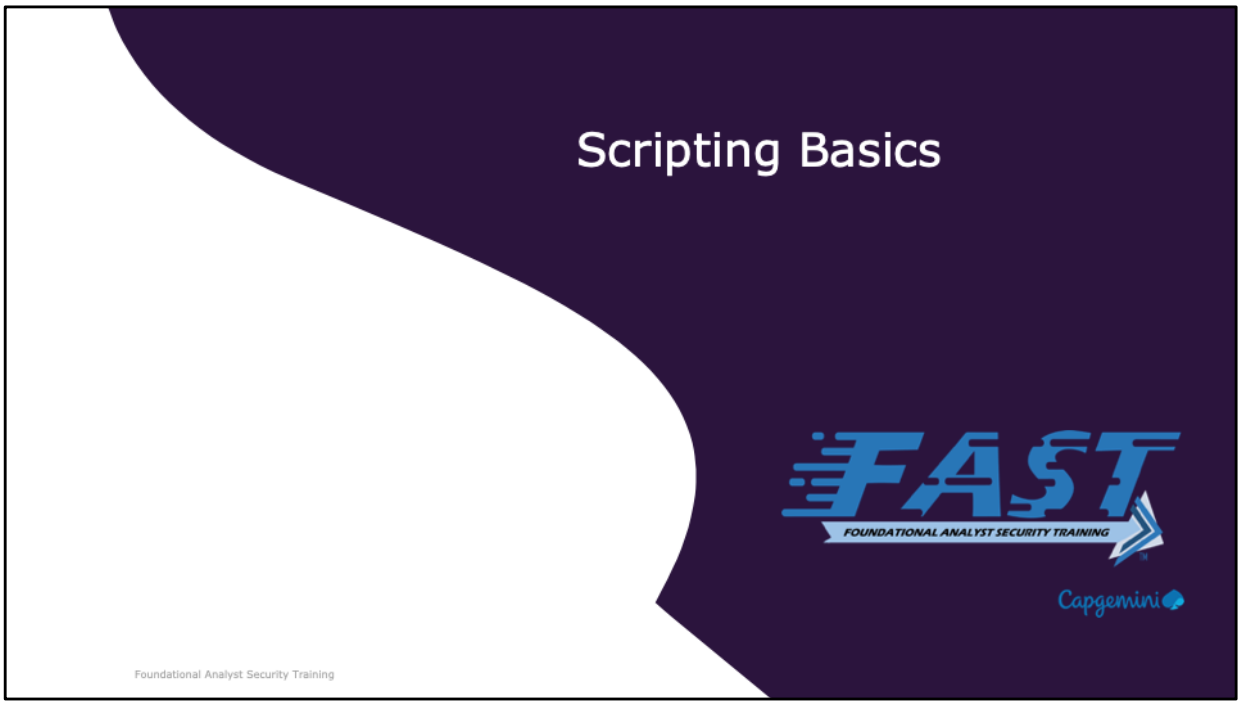
- grep is a command-line utility for searching plaintext data sets for lines that match a regular expression.
- Its name comes from the ed command g/re/p, which has the same effect: doing a global search with the regular expression and printing all matching lines.
- grep was originally developed for the Unix operating system but later available for all Unix-like systems.





Content Source:


Questions?
FAST
FOUNDATIONAL ANALYST SECURITY TRAINING
Foundational Analyst Security Training
© Capgemini 2019. All rights reserved | 64



In this topic, we will cover basic scripting concepts.

The topics covered in this module include:

- Basic Scripting Concepts
- Bash Shell
- Perl
- Other Scripting Languages

Topic Learning Objectives

Upon completion of this topic, the student should be able to do the following:

- Understand the difference between compiled and non-compiled scripting languages.
- Identify some common programming and scripting languages.
- Understand how Python is used to code scripts to perform repetitive tasks.
- Understand how Python interacts and manipulates information using strings.

Scripting vs. Programming

Scripting Languages

- Programming language that support scripts or programs written for a special run time environment.
- Often used for the automation of repetitive tasks such as log analysis or searching.
- Operate at a very high level, and do not have to be compiled like a programming language.

FAST Foundational Analyst Security Training

A **scripting** or **script language** is a programming language that supports **scripts** — programs written for a special run-time environment that automates the execution of tasks[1] that could alternatively be executed one-by-one by a human operator.

Scripting languages are often interpreted (rather than compiled). Primitives are usually the elementary tasks or API calls, and the language allows them to be combined into more complex programs.

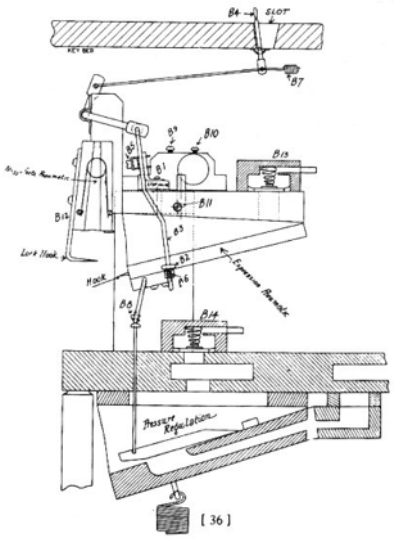
Environments that can be automated through scripting include software applications, web pages within a web browser, usage of the shells of operating systems (OS), embedded systems, as well as numerous games.

A scripting language can be viewed as a domain-specific language for a particular environment; in the case of scripting an application, it is also known as an **extension language**. Scripting languages are also sometimes referred to as very high-level programming languages, as they operate at a high level of abstraction, or as **control languages**, particularly for job control languages on mainframes.

Scripting vs. Programming (cont.)

Programming Language

- A programming language is made up of a set of instructions that produce an output.
- These types of languages are used in computer programming to create instructions for computers.
- Modern-day programming languages are known as compiled languages that have to be compiled before they can be run.



FAST Foundational Analyst Security Training

A **programming language** is a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.
Most programming languages consist of instructions for computers.

There are programmable machines that use a set of specific instructions, rather than general programming languages. Early ones preceded the invention of the digital computer, the first probably being the automatic flute player described in the 9th century by the brothers Musa in Baghdad, during the Islamic Golden Age.[1]

Since the early 1800s, programs have been used to direct the behavior of machines such as Jacquard looms, music boxes and player pianos.[2] The programs for these machines (such as a player piano's scrolls) did not produce different behavior in response to different inputs or conditions.

From 1832 until 1932, the firm produced mechanical musical instruments of the highest quality. The firm's founder, Michael Welte (1807-1880), and his company were prominent in the technical development and construction of orchestrions from 1850, until the early 20th century.

Scripting vs. Programming (cont.)

Programming Language

One of the earliest forms of programming was used by the Welte-Mignon automatic piano-playing systems, which used a prerecorded mechanical representation of the music to reproduce a song.

FAST Foundational Analyst Security Training

A **programming language** is a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.
Most programming languages consist of instructions for computers.

There are programmable machines that use a set of specific instructions, rather than general programming languages. Early ones preceded the invention of the digital computer, the first probably being the automatic flute player described in the 9th century by the brothers Musa in Baghdad, during the Islamic Golden Age.[1]

Since the early 1800s, programs have been used to direct the behavior of machines such as Jacquard looms, music boxes and player pianos.[2] The programs for these machines (such as a player piano's scrolls) did not produce different behavior in response to different inputs or conditions.

From 1832 until 1932, the firm produced mechanical musical instruments of the highest quality. The firm's founder, Michael Welte (1807-1880), and his company were prominent in the technical development and construction of orchestrions from 1850, until the early 20th century.

Scripting: Process

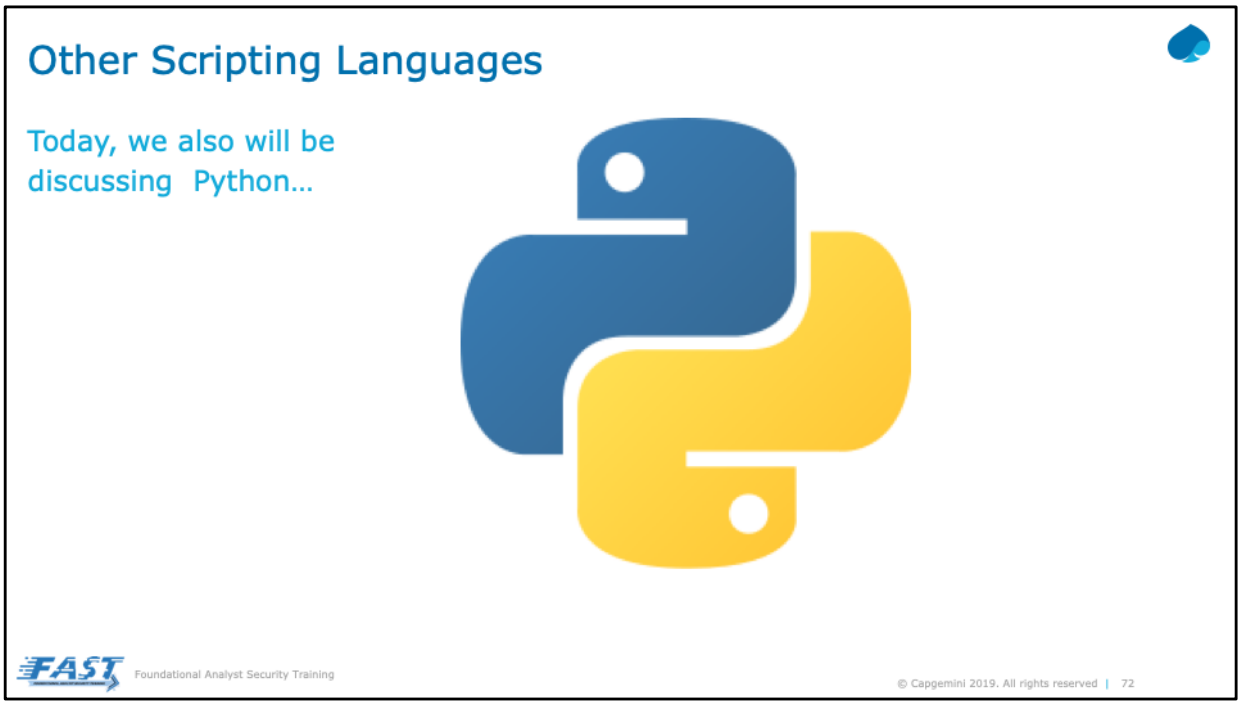
- Scripting languages are tools to reach a goal.
- They all have similarities and differences, but whichever language helps to reach that goal is what should be used.
- Do not get hung up on a specific language or its syntax.

Approach: Start with a roadmap of what you are trying to do.

For example…

Take a proxy server log file, filter for lines with source address "222.216.0.45," and then display the results ranked by the amount of times the destination addresses appear (from most active destination address to least active).

Step 1	Read in the log file.
Step 2	Filter out all entries with "222.216.0.45" in the source address field.
Step 3	Group by destination IP.
Step 4	Display in reverse order according to size of the group, along with the count.



As we have learned A **scripting** or **script language** is a programming language that supports **scripts** — programs written for a special run-time environment that automates the execution of tasks[1] that could alternatively be executed one-by-one by a human operator. Scripting languages are often interpreted (rather than compiled). Primitives are usually the elementary tasks or API calls, and the language allows them to be combined into more complex programs. Environments that can be automated through scripting include software applications, web pages within a web browser, usage of the shells of operating systems (OS), embedded systems, as well as numerous games.

VBScript ("*Microsoft Visual Basic Scripting Edition*") is an Active Scripting language developed by Microsoft that is modeled on Visual Basic.

Lasso is an application server and server management interface used to develop internet applications and is a general-purpose, high-level programming language.

JavaScript (/ˈdʒɑːvəˌskrɪpt/),[8] often abbreviated as **JS**, is a high-level, interpreted programming language that conforms to the ECMAScript specification. It is a programming language that is characterized

as dynamic, weakly typed, prototype-based and multi-paradigm.

Prototype-based programming is a style of object-oriented programming in which behavior reuse (known as inheritance) is performed via a process of reusing existing objects via delegation that serve as prototypes. This model can also be known as *prototypal*, *prototype-oriented, classless*, or *instance-based* programming. Delegation is the language feature that supports prototype-based programming.

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.
Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

Python – Overview

- Python is a programming language created by Guido van Rossum that was released in 1991.
- Python is an interpreted, high-level programming language used for "utility" or general-purpose coding.
- Python focuses on code readability.
- Python is open sourced and is available for most Windows and NIX platforms.

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

FAST Foundational Analyst Security Training

Python is a programming language created by Guido van Rossum that was released in 1991.
Python is an interpreted, high-level programming language used for 'utility' or general purpose coding.
Python focuses on code readability

Trivia….

Van Rossum's long influence on Python is reflected in the title given to him by the Python community: *Benevolent Dictator For Life* (BDFL) – a post from which he gave himself permanent vacation on July 12, 2019.

Python – Overview (cont.)

Python is one of the most popular programming languages and is used by everyone from the Library of Congress to Google and NASA.

Python – Overview (cont.)

```
1 #!/usr/bin/env python
2 print "Hello World!"
3
4
```

- Python is currently in version 2.7 release and is included on the laboratory machines, when needed.
- Python supports Object Oriented Programming (OOP), as well as Structured Programming.
- Python is what is referred to as an interpreted programming language.

FAST Foundational Analyst Security Training

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as *attributes),* and code, in the form of procedures (often known as *methods*). A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.[1][2] OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.

An **interpreted language** is a type of programming language for which most of its implementations execute instructions directly and freely, without previously compiling a program into machine-language instructions.

The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines, and then into another language (often machine code).

Python Philosophy…

Beautiful is better than ugly
Explicit is better than implicit
Simple is better than complex
Complex is better than complicated
Readability counts

Python – Syntax

Indentation

- Python groups code blocks using indentation and colons.
- This is different from many other programming languages, which use braces or curly brackets {}.
- In this example, the "else" actually belongs to the second "if" statement.

```
1   /*Bogus C code*/
2   if (FUN)
3         if (SUN)
4               baz (foo, bar);
5   else
6         qux();
7
8
8
10
11
```

FAST Foundational Analyst Security Training

 | 76

Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

Indentation

- Now, see the same statement written in Python.
- It appears much less confusing.
- Clearly see that the "else" statement belongs to the second "if" statement.

```
1   #!/usr/bin/env python
2   if FUN:
3         if SUN:
4               baz (foo, bar)
5         else:
6               qux()
7
8
8
10
11
```

FAST Foundational Analyst Security Training

 | 77

Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

Comments

- Comments are lines that exist in computer programs that are ignored by compilers and interpreters.
- Including comments in programs makes code more readable for humans, as it provides some information or explanation about what each part of a program is doing.
- Like many other languages, Python uses the pound sign (#) to denote a comment.
- Comment much, comment often, comment completely!

```
1   #!/usr/bin/env python
2   # Print 'Hello World!' to
3   console
4   print "Hello World!"
5
6
7
8
8
10
11
```


Another unique aspect to python is that any string not assigned to a variable is considered a comment

Python – Syntax (cont.)

Strings and Substrings

- Strings are amongst the most popular types in Python.
- Create them simply by enclosing characters in quotes.
- Python treats single quotes the same as double quotes.
- Creating strings is as simple as assigning a value to a variable.

```
1   #!/usr/bin/env python
2
3   var1 = 'Back to the Future!'
4   var2 = "Flux_Capacitor"
5
6
7
8
8
10
11
```

FAST Foundational Analyst Security Training

 | 79

Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

Strings and Substrings

- Python does not support a character type; these are treated as strings of length one, thus also considered a substring.
- To access substrings, or components of a string, use the square brackets for slicing along with the index or indices to obtain a substring.
- In this example, get an output of different characters from the variables.

```
1   #!/usr/bin/env python
2
3   var1 = 'Back to the Future!'
4   var2 = "Flux_Capacitor"
5
6   print "var1[0]: ", var1[0]
7   print "var2[1:5]: ", var2[1:5]
8
8
10
11
```

FAST Foundational Analyst Security Training

 | 80

Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

Strings and Substrings

- The output will look like this:
  var1[0]:

```
1   #!/usr/bin/env python
2
3   var1[0]:  B
4   var2[1:5]:  lux_C
5
6
7
8
8
10
11
```


Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

Escape Characters

- Python also uses Escape Characters or non-printable characters that can be represented by backslash notation.
- This table contains a few of the possible escape characters to use in Python; a little online research will yield a more complete list.

Backslash Notation	Hex Character	Description
\a	0x07	Bell or Alert
\s	0x20	Space
\v	0x0b	Vertical Tab
\n	0x0a	Newline
\r	0x0d	Carriage Rtn
\b	0x08	Backspace
\nnn	Octal notation where n is the range 0-7	

FAST Foundational Analyst Security Training

Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

String Special Operators

- String special operators are used to manipulate variables in Python.
- The table contains a few of the string special operators that can be used in Python; once again, a more exhaustive list is available online.

Operator	Description
+	Concatenation: Adds values on either side of the operator
*	Repetition: Creates new strings, concatenating multiple copies of the same string
[ ]	Slice: Gives the Characters from the given index
[x:y]	Range Slice: Gives the Characters from the given range
%	Format: Performs string formatting

Ashley can you offer some insight on this for the instructor notes?

We will talk about sting formatting operators shortly

Python – Syntax (cont.)

String Special Operators – +

Operator	Description
+	Concatenation: Adds values on either side of the operator
*	Repetition: Creates new strings, concatenating multiple copies of the same string
[]	Slice: Gives the Character from the given index
[:]	Range Slice: Gives the Characters from the given range

```
1   #!/usr/bin/env python
2
3   var1 = 'hello'
4   var2 = 'python'
5
6   print var1 + var2
7   #this will be your output
8
8   hellopython
10
11
```

FAST Foundational Analyst Security Training

Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

String Special Operators – *

Operator	Description
+	Concatenation: adds values on either side of the operator
*	Repetition: Creates new strings, concatenating multiple copies of the same string
[]	Slice: Gives the Character from the given index
[:]	Range Slice: Gives the Characters from the given range

```
1   #!/usr/bin/env python
2
3   var1 = 'hello'
4   var2 = 'python'
5
6   print var2*
7   #this will be your output
8
8   pythonpython
10
11
```


Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

String Special Operators – [ ]

Operator	Description
+	Concatenation: adds values on either side of the operator
*	Repetition: Creates new strings, concatenating multiple copies of the same string
[ ]	Slice: Gives the Characters from the given index
[:]	Range Slice: Gives the Characters from the given range

```
1   #!/usr/bin/env python
2
3   var1 = 'hello'
4   var2 = 'python'
5
6   print var2[3]
7   #this will be your output
8
8   h
10
11
```


Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

String Special Operators – [x:y]

Operator	Description
+	Concatenation: adds values on either side of the operator
*	Repetition: Creates new strings, concatenating multiple copies of the same string
[]	Slice: Gives the Character from the given index
[x:y]	Range Slice: Gives the Characters from the given range

```
1  #!/usr/bin/env python
2
3  var1 = 'hello'
4  var2 = 'python'
5
6  print var1[2:4]
7  #this will be your output
8
8  llo
10
11
```


Ashley can you offer some insight on this for the instructor notes?

Python – Syntax (cont.)

String Formatting Operator

- One of Python's most interesting features is the string format operator "%"
- This operator is unique to strings and makes up for the lack of having functions from the C programming languages "printf()" functions.
- Combining the "%" operator with different conversions, reformat data used in the string.
- For example, the "%s" string formatting operator will perform a string conversion via "str()" prior to formatting.
- The string formatting operator "%d" will output a signed decimal integer.

```
1   #!/usr/bin/env python
2
3   Print "My name is %s and my
4   height is %d ft!" % ('Marty',
5   6)
6
7   My name is Marty and my height
8   is 6 ft!
8
10
11
```


Ashley can you offer some insight on this for the instructor notes?

Signed and unsigned integers deal with how computer calculate the binary representations of the integers into binary formatting. It is beyond the scope of this class, but feel free to do your own research.

Python – Summary

- Python is a great flexible tool for automating complex or repetitive tasks using simple scripts that you can write yourself.
- An example of where Python might be an effective tool is the repetitive analysis of Windows log files from hundreds of host computers.

```
1  #!/usr/bin/env python
2
3
4
5
6
7
8
8
10
11
```

FAST Foundational Analyst Security Training

Ashley can you offer some insight on this for the instructor notes?

Signed and unsigned integers deal with how computer calculate the binary representations of the integers into binary formatting. It is beyond the scope of this class, but feel free to do your own research.

Python – Example

```
#!/usr/bin/env python
""" This simple script looks at all files starting with ac and
treats it as an standard web log file. Reports on all domains that
have visited a course description page"""

import os
import re
from hname import *

shorttable = {}
ctable = {}
lookfor = re.compile("GET\s/course/([a-z]+)")

for filename in os.listdir("."):
   if not filename.startswith("ac"):
      continue  # eliminate files that are not logfiles
   for line in open(filename).xreadlines():
      parts = line.split(" ")
# How much of the name / IP address do we report?
      host = hname(parts[0])
      dpsumm = host.getShort()
# Count the host usage and log any course files
      shorttable[dpsumm] = 1 + shorttable.get(dpsumm,0)
      gotten = lookfor.findall(line)
      if (not gotten): continue
      ctable[dpsumm] = ctable.get(dpsumm,"") + " " + gotten[0]

def byhits(one,two):
   global shorttable
   return shorttable[two].__cmp__(shorttable[one])

visitors = ctable.keys()
visitors.sort(byhits)

for browser in visitors:
   print browser,ctable[browser],shorttable[browser]
```

This simple script looks at all files starting with ac and treats it as an standard web log file.

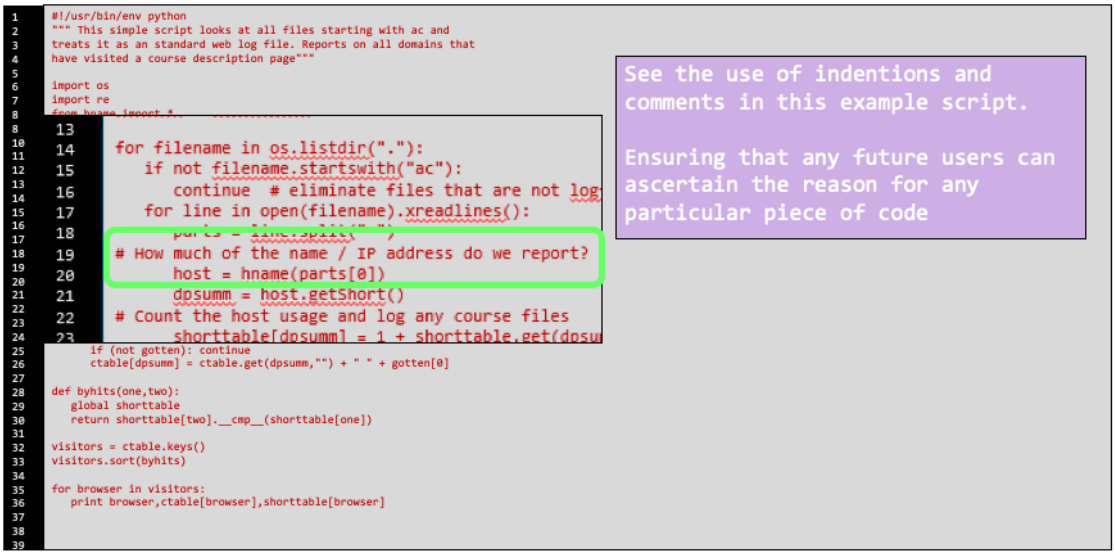
Reports on all domains that have visited a course description page

Python – Example (cont.)

```
#!/usr/bin/env python
""" This simple script looks at all files starting with ac and
treats it as an standard web log file. Reports on all domains that
have visited a course description page"""

import os
import re
from hname import *

for filename in os.listdir("."):
    if not filename.startswith("ac"):
        continue  # eliminate files that are not log
    for line in open(filename).xreadlines():
        parts = line.split(" ")
# How much of the name / IP address do we report?
        host = hname(parts[0])
        dpsumm = host.getShort()
# Count the host usage and log any course files
        shorttable[dpsumm] = 1 + shorttable.get(dpsu
    if (not gotten): continue
    ctable[dpsumm] = ctable.get(dpsumm,"") + " " + gotten[0]

def byhits(one,two):
   global shorttable
   return shorttable[two].__cmp__(shorttable[one])

visitors = ctable.keys()
visitors.sort(byhits)

for browser in visitors:
   print browser,ctable[browser],shorttable[browser]
```

See the use of indentions.

Python – Example (cont.)



```
#!/usr/bin/env python
""" This simple script looks at all files starting with ac and
treats it as an standard web log file. Reports on all domains that
have visited a course description page"""

import os
import re
from hname import *

for filename in os.listdir("."):
    if not filename.startswith("ac"):
        continue  # eliminate files that are not log
    for line in open(filename).xreadlines():
        parts = line.split(" ")
# How much of the name / IP address do we report?
        host = hname(parts[0])
        dpsumm = host.getShort()
# Count the host usage and log any course files
        shorttable[dpsumm] = 1 + shorttable.get(dpsu
    if (not gotten): continue
    ctable[dpsumm] = ctable.get(dpsumm,"") + " " + gotten[0]

def byhits(one,two):
    global shorttable
    return shorttable[two].__cmp__(shorttable[one])

visitors = ctable.keys()
visitors.sort(byhits)

for browser in visitors:
    print browser,ctable[browser],shorttable[browser]
```

See the use of indentions and comments in this example script.

Ensuring that any future users can ascertain the reason for any particular piece of code

Scripting in Bash

Bash is a "sh" – compatible command language interpreter that executes commands read from standard input or from a file.

This allows advanced users to automate routine repetitive tasks that do not require the advanced functionality of a structured programming language.

Bash Logic

File Redirection

> file	create (overwrite) file
>> file	append to file
< file	read from file
a | b	Pipe 'a' as input to 'b'

Common Constructs

$ while read f > do > echo "Line is $f" > done < file	read text file line by line note: "$" prompt becomes ">"
$ grep foo myfile afoo foo foobar	find lines in myfile containing the text "foo"
$ cut -d: -f5 /etc/passwd Steve Parker	get 5th field delimited by colon
$ cmd1 || cmd2	run cmd1; if fails, run cmd2
$ cmd1 && cmd2	run cmd1; if it works, run cmd2
case $foo in a) echo "foo is A" ;; b) echo "foo is B" ;; *) echo "foo is not A or B" ;; esac	act upon the value of a variable note that ";;" is required at the end of each section
myvar=`ls`	get output of ls into variable

doubleit() {
expr $1 * 2
}
doubleit 3 # returns 6
	function declaration and syntax for calling it

Test Operators

```
if [ "$x" -lt "$y" ]; then
  # do something
fi
```

Numeric Tests

lt	less than
gt	greater than
eq	equal to
ne	not equal
ge	greater or equal
le	less or equal

File Tests

nt	newer than
d	is a directory
f	is a file
x	executable
r	readable
w	writeable

String Tests

=	equal to
z	zero length
n	not zero length

Logical Tests

&&	logical AND
||	logical OR
!	logical NOT

Arguments

$0	program name
$1	1st argument
$2	2nd argument
…	…
$#	no. of arguments
$*	all arguments

Variable Substitution

${V:-default}	$V, or "default" if unset
${V:=default}	$V (set to "default" if unset)
${V:?err}	$V, or "err" if unset

Conditional Execution

cmd1 || cmd2	run cmd1; if fails, run cmd2
cmd1 && cmd2	run cmd1; if ok, run cmd2

Files

mv /src /dest	move /src into /dest
ls a*	list files beginning with "a"
ls *a	list files ending with "a"
ls -ltr	list oldest first, newest last
ls -lSr	list smallest first, biggest last
ls -a	list all files, including hidden
find /src -print \ | cpio -pudvm	copy /src into current directory, preserving links, special devices, etc.

Preset Variables

$SHELL	what shell am I running?
$RANDOM	provides random numbers
$$	PID of current process
$?	return code from last cmd
$!	PID of last background cmd

Generally Useful Commands

file /etc/hosts	determine file type
basename /bin/ls	strip directory name (ls)
dirname /bin/ls	get directory name (/bin)
ifconfig -a	show all network adapters
netstat -r	show routers
netstat -a	show open ports
date +%Y%m%d	Year, Month, Day
date +%H%M	Hours, Minutes
wc -l	count number of lines
pwd	present working directory

Bash Shell Scripting: Loops

- Loops in shell scripts allow for the creation of repeating processes that do not terminate until a specific condition is met.
- Example: Loop through the specified user directory, and determine if each line is a file or a directory.

```
#!/bin/bash
# search through all files in a directory
for file in /home/b2tf88mph/Documents
do
  if [ -d "$filename" ]
  then
    echo "$file is a directory"
  elif [ -f "$filename' ]
  then
    echo "$file is a file"
  fi
Done
```


Bash Shell Scripting: Other Loop Types

Until

```
count = 1
until ["$*" = ""]
    do
        echo "Count = $count"
        shift
        count = `expr $count + 1`
    done
```

Do-while

```
while  ["$*" != ""]
    do
        echo "Argument value is: $1"
        shift
    done
```

For

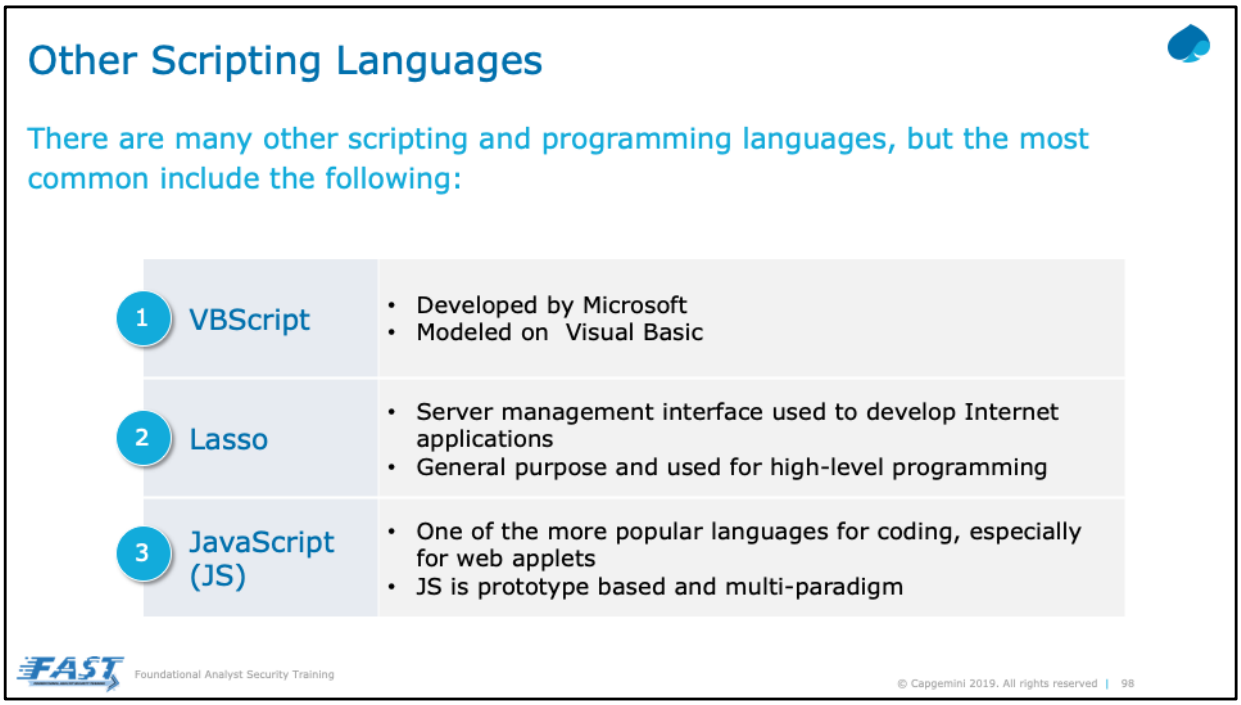
```
# expects array as input ($@)
for item in "$@"
    do
        echo "Argument value is: $item"
    done
```


Bash Shell Scripting: In Practical Application

Example: List all IP addresses that have visited a website via access.log.

```
#!/bin/bash
OUT=/tmp/spam.ip.$$
HTTPDLOG="/var/log/apache2/entry.log"
if [ -f $HTTPDLOG ];
then
      awk '{print}' $HTTPSDLOG >$OUT
      awk '{ print $1}' $OUT | sort -n | uniq -c | sort -n
else
      echo "$HTTPDLOG not found. Make sure domain exists and setup correctly."
fi
/bin/rm -f $OUT
```

Try running this against dmz_web.log, and see what IPs show up!



As we have learned A **scripting** or **script language** is a programming language that supports **scripts** — programs written for a special run-time environment that automates the execution of tasks[1] that could alternatively be executed one-by-one by a human operator. Scripting languages are often interpreted (rather than compiled). Primitives are usually the elementary tasks or API calls, and the language allows them to be combined into more complex programs. Environments that can be automated through scripting include software applications, web pages within a web browser, usage of the shells of operating systems (OS), embedded systems, as well as numerous games.

VBScript ("*Microsoft Visual Basic Scripting Edition*") is an Active Scripting language developed by Microsoft that is modeled on Visual Basic.

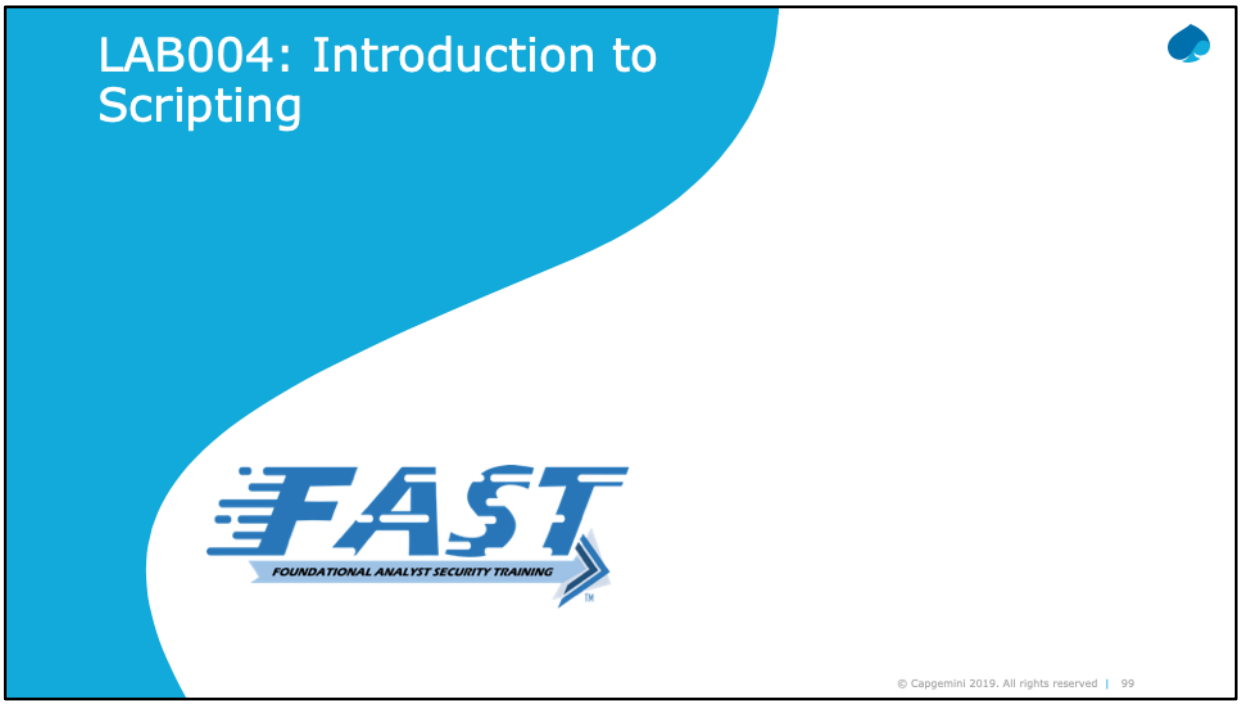
Lasso is an application server and server management interface used to develop internet applications and is a general-purpose, high-level programming language.

JavaScript (/ˈdʒɑːvəˌskrɪpt/),[8] often abbreviated as **JS**, is a high-level, interpreted programming language that conforms to the ECMAScript specification. It is a programming language that is characterized

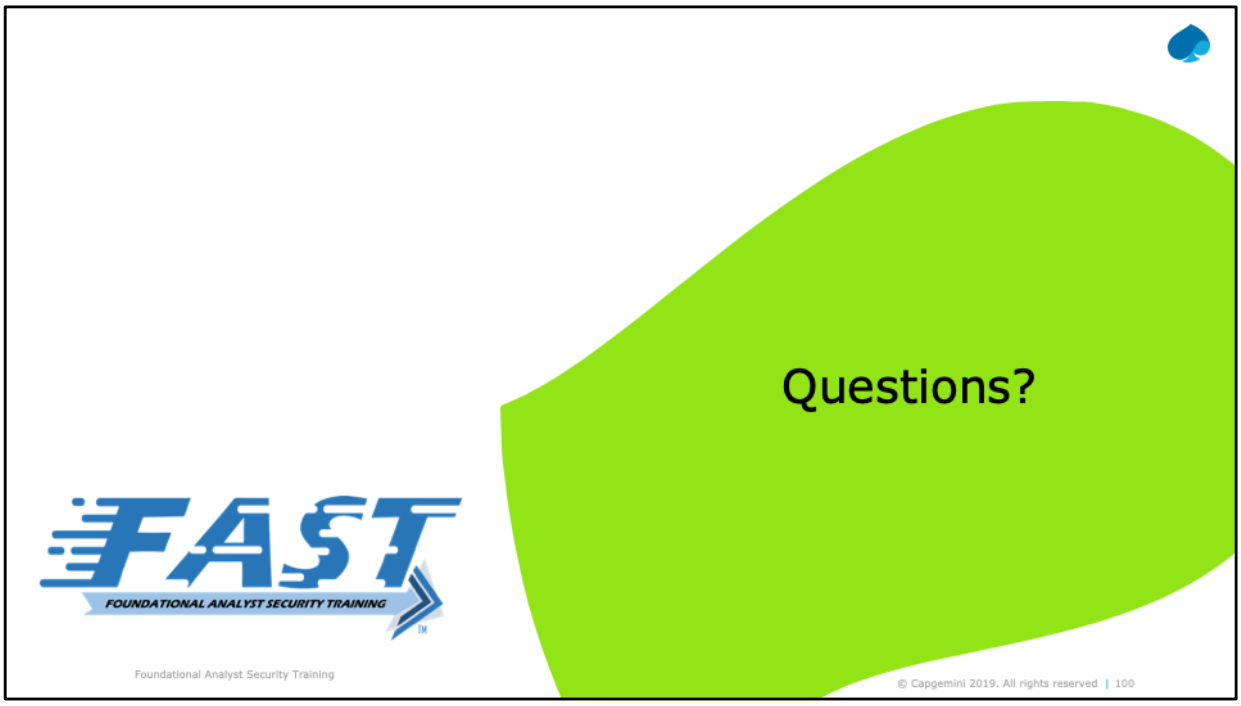
as dynamic, weakly typed, prototype-based and multi-paradigm.

Prototype-based programming is a style of object-oriented programming in which behavior reuse (known as inheritance) is performed via a process of reusing existing objects via delegation that serve as prototypes. This model can also be known as *prototypal*, *prototype-oriented, classless*, or *instance-based* programming. Delegation is the language feature that supports prototype-based programming.

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.
Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.



Content Source:


Questions?
FAST
FOUNDATIONAL ANALYST SECURITY TRAINING
Foundational Analyst Security Training
© Capgemini 2019. All rights reserved | 100


FAST
FOUNDATIONAL ANALYST SECURITY TRAINING
People matter, results count.
This presentation contains information that may be privileged or confidential and is the property of the CapGemini Group.
Copyright © 2019 CapGemini. All rights reserved.
About CapGemini
A global leader in consulting, technology services and digital transformation, CapGemini is at the forefront of innovation to address the entire breadth of clients' opportunities in the evolving world of cloud, digital and platforms. Building on its strong 50-year heritage and deep industry-specific expertise, CapGemini enables organizations to realize their business ambitions through an array of services from strategy to operations. CapGemini is driven by the conviction that the business value of technology comes from and through people. It is a multicultural company of 200,000 team members in over 40 countries. The Group reported 2017 global revenues of EUR 12.8 billion.
Learn more about us at
www.capgemini.com