LESSON NOTES

Intro to Linux

Scripting, Containers, and Automation

3.1.3 Script Utilities and Variables

Lesson Overview:

Students will:

- Understand how to use common script utilities to automate tasks

Guiding Question: How can common script utilities be used to automate tasks?

Suggested Grade Levels: 9 - 12

Technology Needed: None

CompTIA Linux+ XK0-005 Objective:

3.1 - Given a scenario, create simple shell scripts to automate common tasks

- Common script utilities
  - awk
  - sed
  - find
  - xargs
  - grep
  - egrep
  - tee
  - wc
  - cut
  - tr
  - head
  - tail
- Environment variables
  - $PATH
  - $SHELL
  - $?
- Relative and absolute paths

This content is based upon work supported by the US Department of Homeland Security's Cybersecurity & Infrastructure Security Agency under the Cybersecurity Education Training and Assistance Program (CETAP).

Script Utilities and Variables

Common Script Utilities

awk excels at scanning files for specific patterns and extracting information based on those patterns. Users can define patterns using regular expressions and specify actions to be taken when a match is found. It is particularly powerful for processing structured text data, like CSV files.

sed, the stream editor, is designed for efficiently processing and transforming text streams. It operates on a line-by-line basis, making it effective for batch processing and editing. sed allows for search and replace, insertion, deletion, and other text transformations, making it a versatile tool in scripting.

The **find** command is used for searching files and directories in a directory hierarchy. Users can specify various criteria such as file type, size, modification time, etc. It is a powerful tool for locating files that meet specific conditions and then performing actions on those files.

xargs takes input from a pipe or a file and converts it into arguments for a specified command. Useful for situations where a command doesn't directly accept input from standard input. Enhances the flexibility of command-line operations by allowing dynamic argument generation.

grep is a widely used tool for searching patterns within text files. It displays lines containing the specified pattern, making it invaluable for log analysis, code searches, and more. Grep supports regular expressions for advanced pattern matching.

egrep (or **grep -E**) is an extended version of grep that supports extended regular expressions. It provides additional features for complex pattern matching, offering a broader range of search capabilities.

tee reads from standard input and writes to both standard output and files simultaneously. Useful for redirecting output to multiple destinations, allowing users to see real-time output while saving it to a file.

Word Count (**wc**) is a utility that counts lines, words, and characters in a file or input stream. It provides essential statistics about the content of a file, aiding in data analysis and processing.

cut is used to extract specific sections (fields) from each line of a file.

It's particularly handy for working with delimited data, such as CSV files, where users can specify the delimiter and the desired fields.

tr translates or deletes characters in a stream of data. It's useful for character-level transformations, such as converting lowercase to uppercase, deleting specified characters, or replacing characters.

head displays the first few lines of a file, providing a quick preview of the beginning of the file. Useful for examining the structure or initial content of a file without opening the entire document.

tail displays the last few lines of a file, making it a valuable tool for monitoring log files in real-time. It's

also useful for extracting the end portion of a file, especially in situations where new data is continually appended.

Environment Variables and Paths

`$PATH` specifies a colon-separated list of directories where the system looks for executable programs. When a command is entered, the system searches these directories in order to find the executable associated with the command. Users can customize the **`$PATH`** variable to include additional directories, allowing them to run custom scripts or executables without specifying the full path.

`$SHELL` points to the user's default shell, which is the command-line interpreter or program that provides the command-line interface. It determines the behavior of the command line, including syntax, scripting language, and available features. Changing the value of **`$SHELL`** can alter the user's default command-line environment.

`$?` represents the exit status of the last executed command. After a command is run, the **`$?`** variable holds the exit code, where a value of 0 typically indicates successful execution, and non-zero values indicate errors or specific exit statuses. It's commonly used in scripts for error handling and conditional execution based on the success or failure of previous commands.

Relative paths describe the location of a file or directory in relation to the current working directory. This is useful for navigating within the file system without specifying the entire path. Examples **`include ./file`** (referring to a file in the current directory) or **`../parent_directory/file`** (referring to a file in the parent directory).

Absolute paths specify the complete path from the root directory to a file or directory. They provide an unambiguous reference to the location of a file or directory, regardless of the current working directory. Examples include **`/home/user/file or /var/log/syslog`**, where the leading slash indicates the path starts from the root directory.