

Federal University Dutse



CCS 206 - Introduction to Linux for Security and Forensics

Lecture Note 5
By
Muhammad S. Ali



I/O Redirection

- This lesson introduces commands that allows you to redirect the input and output of commands to and from files, as well as connect multiple commands to make powerful command *pipelines*.
- cat – concatenate files
- sort – sort lines of text
- uniq – report or omit repeated lines
- wc – print newline, word, and byte counts for each file
- grep – print lines matching a pattern
- head – output the first part of a file
- tail – output the last part of a file
- tee – read from standard input and write to standard output and files.



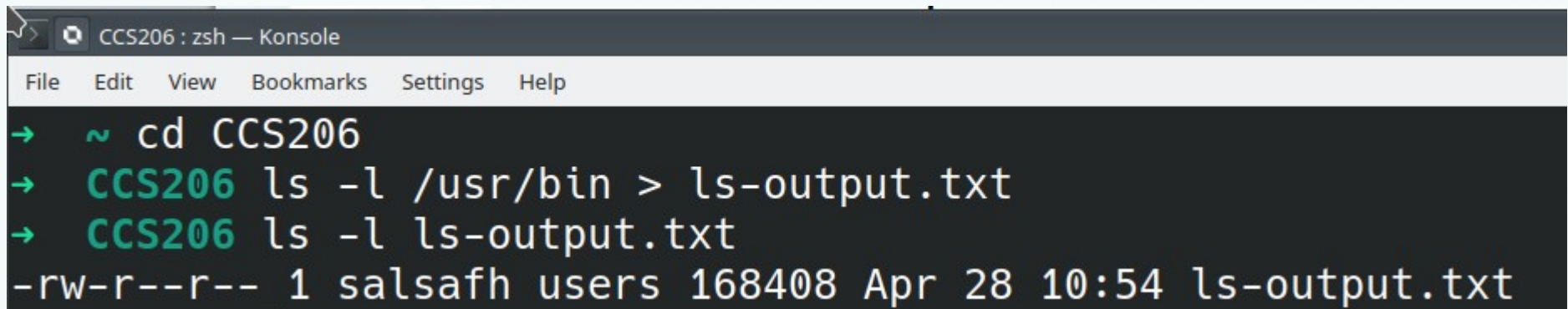
Standard Input, Output, and Error

- Many programs produce output of some kind. This output often consists of two types. First, program's result; that is, the data the program is designed to produce. Second, status and error messages that tell us how the program is getting along.
- Programs such as *ls* send their results to a special file called *standard output* (*stdout*) and their status message to another file called the *standard error* (*stderr*).
- By default, both *stdout* and *stderr* are linked to the screen and not saved into a disk file.
- In addition, many programs take input from a facility called *standard input* (*stdin*), which is by default attached to the keyboard.
- I/O redirection allows us to change where the output goes and where the input comes from. Normally, output goes to the *screen* and input comes from the *keyboard*, but with I/O redirection we change that.



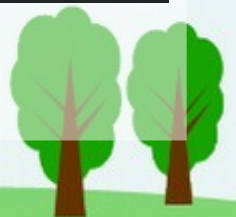
Redirecting Standard Output

- I/O redirection allows us to redefine where standard output goes.
- To redirect standard output to another file instead of the screen, we use the `>` redirection operator followed by the name of the file.
- We often do standard output redirection to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file *ls-output.txt* instead of the screen, see the screenshot below:



```
CCS206 : zsh — Konsole
File Edit View Bookmarks Settings Help
→ ~ cd CCS206
→ CCS206 ls -l /usr/bin > ls-output.txt
→ CCS206 ls -l ls-output.txt
-rw-r--r-- 1 salsafh users 168408 Apr 28 10:54 ls-output.txt
```

The screenshot shows a terminal window titled "CCS206 : zsh — Konsole". It has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows a sequence of commands and their output: first, a prompt followed by `~ cd CCS206`; second, a prompt followed by `CCS206 ls -l /usr/bin > ls-output.txt`; third, a prompt followed by `CCS206 ls -l ls-output.txt`; and finally, the output of the last command: `-rw-r--r-- 1 salsafh users 168408 Apr 28 10:54 ls-output.txt`.



Redirecting Standard Output

- When nonexistent directory is used, we received an error message, which is displayed on the *stdout* instead of the file. That is because, the program sends its output to the standard out.
- If the redirection of the directory that does not exist was sent to an existing file, the file is truncated. That is because, when we redirect output with `>` redirection operator, the destination file is always rewritten from the beginning.
- Using the redirection operator with no command preceding it will truncate an existing file or create a new empty file.
- We can append redirected output to a file instead of overwriting the file from the beginning using the `>>` redirection operator.



Redirecting Standard Output

- Using the >> operator will result in the output being appended to the file. If the file does not already exist, it is created just though the > operator had been used. See the examples below:

```
salsafh@localhost:~> ls -l /usr/bin >> lsoutput.txt
```

- Repeating the above command several times will result in an output file that is very large.
- Try the following and observe the output file length:

```
salsafh@localhost:~> ls -l /usr/bin >> ls-output.txt
```

```
salsafh@localhost:~> ls -l /usr/bin >> ls-output.txt
```

```
salsafh@localhost:~> ls -l /usr/bin >> ls-output.txt
```



Redirecting Standard Error

- Redirecting *standard error* lacks the ease of using a dedicated redirection operator. To redirect standard error we must refer to its file descriptor. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard *input*, *output*, and *error*, the shell references them internally as file descriptors 0, 1, and 2, respectively.
- The shell provides a notation for redirecting files using the file descriptor number.
- Since standard error is the same as file descriptor 2, we can redirect standard error with the following notation:

```
salsafh@localhost:~> ls -l /usr/bin 2> ls-error.txt
```

- The file descriptor is placed immediately before the redirection operator to perform the redirection of standard error to the file ls-error.txt.



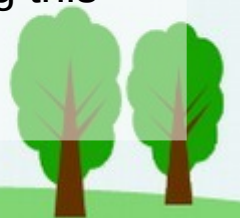
Redirecting stdout and stderr to one file

- There are cases in which we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, here is the traditional way, which works with old versions of the shell:

```
salsafh@localhost:~> ls -l /bin/usr > ls-output.txt 2>&1
```

- With this method, we perform two redirections. First we redirect standard output file ls-output.txt, and then we redirect file descriptor 2 (stderr) to a file descriptor 1 (stdout) using the notation 2>&1.
- The order of the redirection is significant. The redirection of the stderr must always occur after stdout or it doesn't work. In the example above, > ls-output.txt 2>&1 redirects stderr to the file ls-output.txt, but if the order is changed to 2>&1 > ls-output.txt, stderr is directed to the screen.
- Recent versions of bash provide a second, more streamlined method for performing this combined redirection:

```
salsafh@localhost:~> ls -l /bin/usr &> ls-output.txt
```



Disposing Unwanted Output

- Sometimes we do not want output from a command – we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called `/dev/null`. This file is a system device called a bit bucket, which accepts input and does nothing with it.
- To suppress error messages from a command, we can write the following command:

```
salsafh@localhost:~> ls -l /bin/usr 2> /dev/null
```



Redirecting Standard Input

cat – Concatenate Files

- The *cat* command reads one or more files and copies them to standard output. Since the *cat* accept more than one file as argument, it can also be used to join files together.

cat [file...]

- Suppose we downloaded a large file that has been split into multiple chunks and we want join them back together. If the files were named:

movie.mpeg.001 movie.mpeg.002 --- movie.mpeg.099 we could rejoin them with the following command:

[salsafh@localhost](#):~> cat movie.mpeg.0* > movie.mpeg

- If *cat* is not given any arguments, it reads from the standard input (keyboard) and waits for us to type something.



Redirecting Standard Input

cat – Concatenate Files

```
salsafh@localhost:~> cat
```

The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog.

- Hold down CTRL key and press D to tell cat that it has reached *end-of-file* (EOF) on standard input.
- In the absence of filename arguments, *cat* copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Suppose we wanted to create a file called `lazy_dog.txt` containing the text in the above example. We would do this:

```
salsafh@localhost~> cat > lazy_dog.txt
```

The quick brown fox jumps over the lazy dog.

- To see the results, we can use *cat* to copy the file to standard output again:

```
salsafh@localhost:~> cat lazy_dog.txt
```



Pipelines

- The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called *pipelines*. Using the pipe operator `|` (vertical bar), the standard output of one command can be *piped* into the standard input of another.

command1 | command2

- In the following example, we send the long list of the `/usr/bin` directory to `less` – which displays the page by page output of the command to the standard output:

`salsafh@localhost:~> ls -l /usr/bin | less`

- This extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.



Filters

- Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters*.
- Filters take input, change it somehow, and then output it. Suppose we want make a combine list of all the executable programs in */bin* and */usr/bin*, put them in sorted order, and then view the list, we can do as follows:
[salsafh@localhost](#):~> `ls /bin /usr/bin | sort | less`
- Since we specified two directories (*/bin* and */usr/bin*), the output of *ls* would have consisted of two sorted lists, one for each directory. By including *sort* in the pipeline, we change the data to produce a single, sorted list.



uniq – Report or Omit Repeated Lines

- The *uniq* command is often used in conjunction with *sort*. *uniq* accepts a sorted list of data from either standard input a single filename argument and by default, removes any duplicates from the list.
- The following example remove duplicates from both */bin* and */usr/bin* directories:

```
salsafh@localhost:~> ls /bin /usr/bin | sort | uniq | less
```

- In the above example, we use *uniq* to remove any duplicates from the output of the *sort* command. If we want to see the list of duplicates instead, we add the *-d* option to *uniq* as follows:

```
salsafh@localhost: ~> ls /bin usrbin | sort | uniq -d | less
```



wc – Print Line, Word, and Byte Counts

- The `wc` (word count) command is used to display the number of *lines*, *words*, and *bytes* contained in files. For example:

```
salsafh@localhost:~> wc ls-output.txt
```

- In this case it prints out three numbers: lines, words, and bytes contained in `ls-output.txt`. If `wc` is executed without command-line arguments, it accepts standard input. The `-l` option limits its output to only reports lines. To see the number of items we have in our sorted list, we can do this:

```
salsafh@localhost: ~> ls /bin usrbin | sort | uniq | wc -l
```



grep – Print Lines Matching a Pattern

- *grep* is a powerful program used to find text patterns within files, like this:

grep pattern [file...]

- When *grep* encounters a pattern in the file, it prints out the lines containing it. The patterns that *grep* can match can be very complex. We will consider only simple text matches in this lesson.
- Suppose we want find all the files in our list of programs that have the word *zip* in the name. Such a search might give us an idea of which programs on our system have something to do with file compression. We would do this:
salsafh@localhost: ~> ls /bin *usrbin* | sort | uniq | grep zip
- -i option causes *grep* to ignore case and -v tells *grep* to print only lines that do not match the specified pattern.



head/tail – Print First/Last Part of Files

- Sometimes you may not want display all the output from a command. You may want only the first few lines or the last few lines. The *head* command prints the first 10 lines of a file, and the *tail* command prints the last 10 lines. By default, both commands print 10 lines of text, but this can adjusted with the -n option.

[salsafh@localhost](#): ~> head -n 5 ls-output.txt

[salsafh@localhost](#): ~> tail -n 5 ls-output.txt

- *tail* has an option (-f) that allows you to view files in real time. This is useful for watching the progress of log files as they are being written.

[salsafh@localhost](#): ~> sudo tail -f /var/log/messages

- Using the -f option, *tail* continues to monitor the file and when new lines are appended, they immediately appear on the display. This continues until you type CTRL+C.



tee – Print First/Last Part of Files

- Linux provides a command called *tee* which creates a “T” fitting on our pipe. The *tee* program reads a standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline’s contents at an intermediate stage of processing.
- The following example capture the entire directory listing to the file *ls.txt* before *grep* filters the pipeline’s content.

salsafh@localhost: ~> `ls /usr/bin | tee ls.txt | grep zip`

- By default, *tee* overwrites the specified file if it already exists and not empty, to append to the file, use the *-a* option in the *tee* command as follows:

salsafh@localhost: ~> `ls /usr/bin | tee -a ls.txt | grep zip`

