Ben Williams

CPRE 308

Lab 3

The purpose of this lab was to extend our knowledge of creating new processes and having those processes execute other programs. We also learned about how to write a simple shell script (which was not as simple as it sounds) based around the Bourne Again Shell (BASH) and how to implement some of its built in functions in a C program.

The output of the hello_world script is as follows:

Hello World

I can tell that the echo command is used for printing out text to the console, and that the console understands that it is a bash file from the reference in the first line: #! /bin/bash (also acceptable is #! /bin/sh).

An interpreter is a program that reads in text in a certain format and then "interprets" what that text should mean as a program, then usually decides what to do with C or assembly scripting.

Interpreters can also pass over comments, in our own case being the character #. Mid line comments are not always supported, and not recommended for maximum portability.

Using the **type** command, we can tell the type of different commands. If they are built in, they may work differently for different interpreters, while hashed (executable) commands are expected to behave similarly with all interpreters.

cd Built In

ls Aliased Command

To see environment variables in the Linux shell, the command **printenv** is commonly used. To see the more commonly used variables only, pipe in **less.**

To create a new environment variable, we use the **export** command. It can create, overwrite, or append to existing environment variables using the convention **export VAR_NAME=value**.

To print out a single environment variable to the screen, **echo $VAR_NAME** is used. It will simply print the value of the variable to the shell.

These environment variables were exploited using the Shellshock Bug, otherwise known as Bashdoor. The core concept was that Bash could be forced to unintentionally execute commands when they were concatenated to the end of function definitions stored in environment variable values. Nobody really thought about the fact that environment variables were something you had to make sure were secure, but patches were quickly dispatched to fix the issue.

The task we were given was to implement a simple script interpreter called the Cyclone Advanced Shell or CASH for short. It can interpret 4 different commands (cd, pwd, export, and echo) as well as any program specified by the user or can be found in one of the directories specified in $PATH.

To start things off, I checked for correctness of input to the CASH (make sure there were exactly 2 arguments.) After this, I attempted to open the input file using fopen() in read only mode. I made sure that the file descriptor wasn't pointing to a null pointer or something had gone wrong.

I then created an 80 character wide buffer to read in each line of the input. I created a simple while loop that would continue to parse the input shell script until a line referenced a null pointer (we were done reading the file.)

On the first go around, I checked if the line matched the literal string "#! /bin/bash". If so, I set my control bit (first) to 0, indicating that we were no longer on the first line and lines no longer needed to be compared to the literal string.

After this, I checked if the line was blank space or a comment by comparing line[0] to ascii values, and if so, I passed over the line using a continue.

After this, I parsed the input line into a command buffer (the first word) and an argument buffer (everything after the first string.) Using a if/else-if chain and strncmp(), my favorite string parsing function, I could determine which command the user was intending to use.

Before I got into this if/else-if chain, I checked whether or not the user ended the line with an ambersand (&.) If so, they wished to fork off a new process executing a command of their own choosing, so using my previous knowledge of fork and exec(), I created a child process, and forced my parent to wait on the child until it was finished. I then printed out the requested information in the requested format to the console to inform the user that the program had finished running.

After this, I checked whether the user had selected **echo, cd, pwd, export,** or an executable of their own choosing. Depending on which item they selected, I had a different series of C statements to do what they wanted.

Export was definitely the trickiest of the custom implementations simply due to the massive amount of string parsing that needed to be done. I had to find the pointer to the first =, the first $, and the first :, if needed. After that, I parsed to find the environment variable name and value, which were then either created/set to a new value, or appended on to the end of the existing value.

cd and **pwd** were probably the two easiest to implement, since C has functions that literally do what they require. All you have to do is parse the input and print out the return value of either **getcwd** or call **chdir.**

Echo was a little tricky since I had to determine whether or not to print the simple user string or return the value of an environment variable.

All in all, it was a very fun lab and I learned a lot about how the Linux shell works. I also got much better at parsing strings and using pointers, which I feel is a very valuable skill. It was a little challenging after having to implement so many new things, but that's what makes the end product so rewarding.