ProLUG – Scripting System Checks

Required Materials

Rocky 9 or equivalent

Root or sudo command access

EXERCISES (Warmup and quick review)

1. vi /etc/passwd
   Put a # in front of all your local users you created in a lab a few weeks back
   Review how to use vi, if you have a problem getting out or saving your file or use vimtutor
   https://www.tutorialspoint.com/unix/unix-vi-editor.htm
2. rpm –qa | grep –i gcc
3. dnf whatprovides gcc
4. dnf search gcc
   Check out all the options of different compilers
5. dnf install gcc
   Look at what is going to be installed.
6. rpm -qi gcc
   Look at this package to see a little about what gcc does.
7. Repeat steps 2-6 for the software package strace

A brief look at compilers and compiling code

So we did all this just to show you a quick rundown of how compiled code works on your system. We are going to be doing scripting today, which is not compiled, but rather interpreted code according to a type of interpreter so we'll just breeze through this part. You can come back and play with this at your leisure, I will provide links for more study.

1. Let's write a C program

mkdir c_practice
cd c_practice

vi a.c
Add this to the file:

```
#include <stdio.h>

main()
{
```

```
printf("My first compiled program \n");

return 0;
}
```

2. Let's use gcc to compile that program

   gcc a.c

   This will create a file for you called a.out
   If there is an error, does it still work?

   Alternatively, and more correctly, use this:

   gcc –o firstprogram a.c

   Which will create an executable file called firstprogram

   ls –salh

   Will show you all your files. Note how big those compiled programs are.

3. Execute your programs
   ./a.out
   ./firstprogram

   Both of these should do the exact same thing, as they are the same code.
4. Watch what your system is doing when you call it via strace
   strace ./a.out
   strace ./firstprogram

   There's a lot of interesting things going on in that page that you'll see. It's outside the scope of this lab, but you can start to see what libraries your program is using and where memory is mapped to. People will use a simple bash shell execution and exit inside a simple c program like this to figure out where those are mapped in memory on a system to work to escalate priviliges.

Depending on your level of interest, here are some other things to play with in this space now that you have a compiler on your system. Feel free to play with them, or avoid this is programming isn't for you:

http://www.linfo.org/create_c1.html

http://www.cprogramming.com/tutorial/c-tutorial.html

LAB

Log into your Rocky server and become root.

Scripting

All that pre-lab work to tell you that we're not doing any of that today, with gcc. In fact, we're not doing any compiling of any type of programs. We're just using code, fed into the system via some syntax, and making it execute our steps in order. It's like reading lines from a play, in order, or reading music, or maybe even reading a script…

There are a lot of resources to learn how to script. There are a lot of extra things I think you should study if you're interested in getting better at it, as it really is something you have to practice daily to get better. To get started, I think you only need to know 3 things:

1. How to get input and where to put it
2. How to test/evaluate things
3. How to loop

All a system ever does is Input, Process, Output and that's a decent place to start but Those 3 things up there are the keys to starting scripting, because we can jump in head first and worry about some of the other stuff as it comes up.

1. **Getting input**

Let's use examples from our "Operate Running Systems" lab to see what it looks like to gather system information and store it in locations we call variables. Variables in scripting can be thought of as named boxes where we put things we want to look at or compare later. We can, by and large, stuff anything we want into these boxes.
Try this:

```
echo $name          #no output
uname
name=`uname`
echo $name

echo $kernel          #no output
uname -r
kernel=`uname –r`
echo $kernel

echo $PATH
```

There will be output because this is one of those special variables that make up your environment variables. You can see them with printenv | more. These ones should not be changed, but you can change them if you feel the need to. You can reset them by re-logging into your shell, if you overwrite any of them.

So we start to see that we can package things in locations called variables and then variables are called by their name preceded by a $.

Try this to get some numerical values in your system that we can then check later in our conditional tests.

```
cat /proc/cpuinfo
```

Not very good as a count

```
cat /proc/cpuinfo | grep –i proc
```

Lets me know how many processors there are on the system, but is less useful to test against

```
cat /proc/cpuinfo | grep –i proc | wc –l
```

gives me a number that I can use later

```
numProc=` cat /proc/cpuinfo | grep –i proc | wc –l`
```

Store in a variable

```
echo $numProc
```

```
free –m
```

All the output, good for us, but not good for the system

```
free –m | grep –i mem
```

Cut down to not see headers or swap, but still not very readable

```
free –m | grep –i mem | awk ‘{print $2}’
```

Only see the column we need for memory

```
memSize=` free –m | grep –i mem | awk ‘{print $2}’`
```

Store in a variable

```
echo $memSize
```

One of the other, very important types of input you can read are looking at the output variable from a command to see if it found what you expected. I think of this like my “what I expect to see” variable because this is a multitool that is useful in almost every case where you would normally have to look for something in a system.

What we’re going to do it look at the $? (exit code) of a program. We’re going to be able to see if something was or wasn’t found in the system. I’ll show the log we use in the next section For now, just look at the output

```
ps -ef | grep -i httpd | grep -v grep
echo $?
```

```
1                       #nothing found with search so variable of 1 (not good "0")
ps -ef | grep -i httpd
root     5514 17748  0 08:46 pts/0    00:00:00 grep --color=auto -i httpd
echo $?
0                       #something found so normal 0 exit code on the search


rpm -qa | grep -i superprogram
echo $?
1            #Super program didn't exist so grep exited with something other than 0
rpm -qa | grep -i gcc
libgcc-4.8.5-11.el7.x86_64
gcc-4.8.5-11.el7.x86_64
echo $?
0            #gcc is found so we have a value of 0. This is something we can then add to logic later.
```

$? Only holds the value of the last command executed so the storage of that value into a variable must be IMMEDIATE. You can't check and then store it, or you're checking the next command's exit code, even it's own. Store this immediately after execution, as follows.

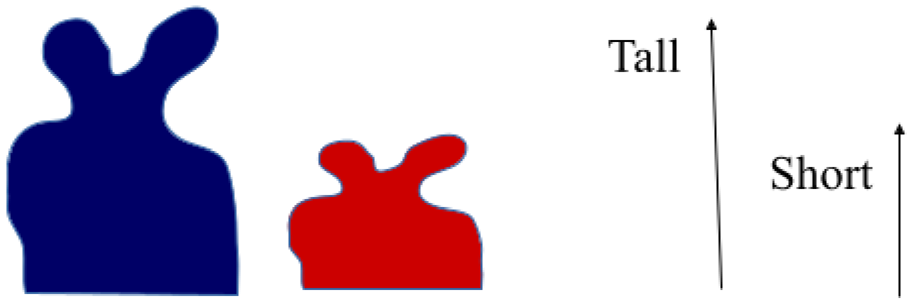
```
rpm -qa | grep -i superprogram
superCheck=`echo $?`
rpm -qa | grep -i gcc
          libgcc-4.8.5-11.el7.x86_64
          gcc-4.8.5-11.el7.x86_64
gccCheck=`echo $?`

echo $superCheck
echo $gccCheck
```

Go back over your old documents and see if there's anything interesting you want to try to stuff into a variable. See if you can do it and echo the value out to the screen.

2. Testing and Evaluating Things

Basics of logic and truth tests

I commonly say that "All engineering is the test for truth." I do not say that to wax philosophical, but rather to say that we test input from somewhere and trust what the input is. We then test that against what we expect and either come to a conclusion of "true" or "anything else". Testing for what something is, is infinitely easier than testing for what something is not, as logically there are infinite numbers of things which are not what I expect. For example:



The Red bunny is tall. We look at our examples and see that this is not true, so the statement evaluates to false.

The Blue bunny is short. We look at our examples and see that this is not true, so the statement evaluates to false.

The idea of And and Or

And is a restricting test and Or is an inclusive test. I will prove that here shortly. Anding is the idea of checking both sides for truth and only evaluating to true if both sides are true. Oring is the idea that either side can be true and the statement evaluates to true. This makes or a more inclusive test. For example:

And Examples

The right bunny is Red and Tall. This evaluates to true for the Red test but False for the Tall test. That means this test evaluates to false.

The left bunny is Blue and Tall. This evaluates to true for the Blue and then also evaluates to true for the Tall test. This statement evaluates to true.

Or Examples

The right bunny is Red or Tall. This evaluates to true for the Red test, but False for the Tall test. That means this evaluates to True.

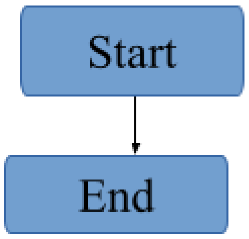
The left bunny is Red and Short. This evaluates to false for Red and False for the Tall test. This means that this or conditional statement evaluates to false.

Google Truth Tables to see the engineering diagrams that we commonly use for testing truth in complex statements. We will not draw them out in this lab. I say this because there are really good

examples and this is a well known/solved/understood concept in the engineering world. I'm not going to re-invent those drawings https://en.wikipedia.org/wiki/Truth_table

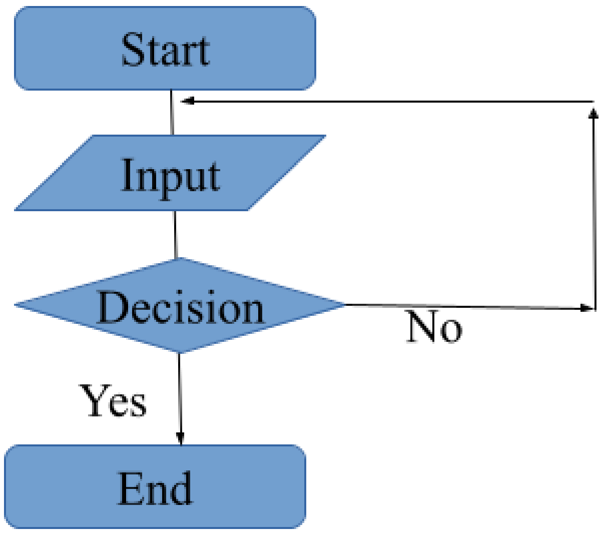
Flow in a program

All programs start at the top and run to the bottom. Data never flow back towards the start, unless on a separate path from a decision which always returns to the original path.



When we start thinking about how to lay something out and logically work through it, the idea of a formalized flow chart can help us get a handle on what we're seeing.

Some common symbols you'll see as we go through drawing out our logic. This example creates a loop in the program until some decision evaluates to yes (true).



3 Types of Decisions

There are 3 primary types of decisions you'll run into with scripting, they are:

1. Single alternative
2. Dual alternative
3. Multiple alternative

Single Alternative (if/then)

Single alternatives either occur or they do not. They only branch from the primary path if the condition occurs. They either can or cannot occur, depending on the condition, but compared to alternative paths where a decision must occur if these do not evaluate to true, they are simply passed over.

Evaluate these from earlier and look at the difference.

```
if [ $superCheck -eq "0" ]; then echo "super exists"; fi

if [ $gccCheck -eq "0" ]; then echo "gcc exists"; fi
```

You'll note that only one of them caused any output to come to the screen, the other simply ran and the condition never had to execute.

Dual alternative (if/then/else)

Dual alternatives forces the code to split. A decision must be made. These are logically if, then, else. We test for a truth, and then, if that condition does not exist we execute the alternative. If you're a parent or if you ever had a parent, this is the dreaded "or else". One of two things is going to happen here, the path splits.

```
if [ $superCheck -eq "0" ]; then echo "super exists"; else echo "super does not exist"; fi
super does not exist
if [ $gccCheck -eq "0" ]; then echo "gcc exists"; else echo "gcc does not exist"; fi
gcc exists
```

Multiple Alternative (if/then/elif/…/else or Case)

Multiple alternatives provide a branch for any numbers of ways that a program can go. They can be structured as if, then, else if (elif in bash), else. They can also be framed in the case statement, which can select any number of cases (like doors) that can be selected from. There should always be a default (else) value for case statements, that is to say, if one of the many conditions don't exist there is something that happens anyways (*) in case statements.

```
superCheck=4
```

```
if [ $superCheck -eq "0" ]; then echo "super exists"; elif [ $superCheck -gt "1" ]; then echo "something is really wrong"; else echo "super does not exist"; fi

gccCheck=5
if [ $gccCheck -eq "0" ]; then echo "gcc exists"; elif [ $gccCheck -gt "1" ]; then echo "something is really wrong"; else echo "gcc does not exist"; fi
```

Set those variables to the conditions of 0, 1, and "anything else" to see what happens.

Think about why greater than 1 does not hit the condition of 1. Might it be easier to think of as greater than or equal to 2? Here's a list of things you can test against.
http://tldp.org/LDP/abs/html/tests.html

Also a huge concept we don't have a lot of time to cover is found here: File test operators http://tldp.org/LDP/abs/html/fto.html, do files exist and can you do operating system level things with them?

We didn't get to go into case, but they are pretty straight forward with the following examples: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_03.html

We didn't get to explore these much earlier, but to test AND and OR functionality use this.

AND condition

```
if [ $gccCheck -eq "0" -a $superCheck -eq "1" ]; then echo "We can install someprogram"; else echo "We can't install someprogram"; fi
We can't install someprogram
```

OR condition

```
if [ $gccCheck -eq "0" -o $superCheck -eq "1" ]; then echo "We can install someprogram"; else echo "We can't install someprogram"; fi
We can't install someprogram
```

Looping around

As with everything today, this is simply a primer and there are hundreds to thousands of examples online a simple google away. There are only two types of loops; counting loops and conditional loops. At the most basic level, we use counting loops when we (or the system) know how many times we want to go through something. Some examples of this are actual hard counts, lists, or lengths of files typically by line. While loops will continue until a condition exists or stops existing. The difference is until that condition occurs there's no reasonable way of knowing how many times the loop may have to occur.

For loops

Counting is iteration.

We can count numbers

```
for i in 1 2 3 4 5; do echo "the value now is $i"; done
```

We can count items

```
for dessert in pie cake icecream sugar soda; do echo "this is my favorite $dessert"; done
```

But, it’s impractical to count for ourselves sometimes so we let the system do it for us.

```
seq 100
seq 4 100
seq 6 2 100
man seq
```

What did each of those do? Let’s put them in a loop we can use

Maybe we want to count our 1000 servers and connect to them by name.

```
for i in `seq 1000`; do echo "Connecting to server p01awl$i"; done
```

Maybe we need to create a list of all our servers and put it in a list

```
for i in `seq 1000`; do echo "p01awl$i" >> serverfile; done
```

Maybe someone else gave us a list of servers and we need to read from that list to connect and do work.

```
for server in `cat serverfile`; do echo "connecting to server $server"; done
```

So, while those are even just a limited set of cases those are all times when, at the start, we know how many times we’re going to go through the loop. Counting or For loops always have a set number of times they’re going to run. That can change, but when it starts the end number of runs is known.

While loops

While loops are going to test conditions and loop while that condition evaluates to true. We can invert that, as we can with all logic, but I find that testing for truth is always easiest.

It is important to remember that “CRTL + C” will break you out of loops, as that will come handy here.

Administrators often find themselves looking at data and needing to refresh that data. One of the simplest loops is an infinite loop that always tests the condition of true (which always evaluates to true) and then goes around again. This is especially useful when watching systems for capacity issues during daemon or program startups.

```
while true; do date; free -m; uptime; sleep 2; done
```

This will run until you break it with CTRL + C. This will loop over the date, free –m, uptime, and sleep 2 commands until the condition evaluates to false, which it will never do.

Let’s run something where we actually have a counter and see what that output is

```
counter=0
while [[ $counter -lt 100 ]]; do echo "counter is at $counter"; (( counter++ )); done
```

What numbers were counted through?

If you immediately run this again, what happens? Why didn’t anything happen?

Reset counter to 0

```
counter=0
```

Re-run the above loop. Why did it work this time?

Reset the counter and run it again. Try moving the counter to before the output. Can you make it count from 1 to 100? Can you make it count from 3 to 251? Are there challenges to getting that to work properly?

What if we wanted something to happen for every MB of free RAM on our system? Could we do that?

```
memFree=`free -m | grep -i mem | awk '{print $2}'`
counter=0
while [[ $counter -lt $memFree ]]; do echo "counter is at $counter"; (( counter++ )); done
```

Pulling it all together

The main thing we haven’t covered is what to actually do with these things we’ve done. We can put them into a file and then execute them sequentially until the file is done executing. To do that we need to know the interpreter (bash is default) and then what we want to do.

```
touch scriptfile.sh
chmod 755 scriptfile.sh                    #let’s just make it executable now and save trouble later
vi scriptfile.sh
```

add the following lines:

```
#!/bin/bash

echo "checking system requirements"

rpm -qa | grep -i gcc > /dev/null
gccCheck=$?

rpm -qa | grep -i superprogram > /dev/null
superCheck=$?

if [ $gccCheck -eq "0" -a $superCheck -eq "1" ]
  then echo "We can install someprogram"
else
  echo "We can't install someprogram"
fi
```

Execute this with the command

```
./scriptfile.sh
```

and you’ll have your first completed script.

run the command

```
strace ./scriptfile.sh
```

to see what is happening with your system when it interprets this script.

Conclusion

There are a lot of ways to use these tools. There are a lot of things you can do and include with scripts. This is just meant to teach you the basics and give you some confidence that you can go out there and figure out the rest. You can develop things that solve your own problems or automate your own tasks.