

# Overview

- Last Lecture
  - IPv6 Bootcamp
- This Lecture
  - Scripting Techniques
- Next Lecture
  - Linux/Unix file system

# Outline

- Least Privilege Principle
- Unix scripting
- Examples
- Other solutions

# Least Privilege Principle

- System admins should follow this principle
  - No user should be given more privileges than they need to do their job. Likewise, no process or file should be given more privileges than it needs to do its job.
- Examples
  - Setuid programs: don't set unless necessary
  - Run programs under special user id such as www and nobody if possible
  - Some applications such as httpd can change its user id from root to nobody after opening the privileged port number 80.
  - Temporary files shouldn't be in /tmp

# Scripting

- Scripting uses the language/commands of command shell
  - It is easier, a glue, weakly typed, and interpreted
- Cons of scripting
  - I/O is expensive due to process communications
  - Interpretation slower than compiled code
  - Interface inconsistency
  - Parsing could be troublesome
  - Security
    - TOCTTOU (time-of-check to time-of-use) attack
    - `rm /tmp/*/*` (`find /tmp -not-accessed-recently | xargs rm`)

# History of scripting

- Who scripting?
  - Administrators, developers, power users, testers, normal users
- History
  - Job Control Language
  - 1960s Unix pipe
  - 1993 Applescript
  - 2005 Automator
  - 2006 Windows PowerShell
- Available shells in Linux
  - bash, sh, tcsh, csh
  - use **cat /etc/shells** to find out which shell you use.

# Origin of scripting

- Unix philosophy
  - Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.
  - Doug McIlroy, Inventor of the | construct
- Because of this principle, there are many small utility programs in Unix
- Scripting is the glue to integrate them together to achieve more complex functionalities.

# Simple script

- `#!/bin/sh echo "Hello, World!"`
- `$ chmod +x ./hello $ ./hello Hello, World! $ sh ./hello Hello, World!`

# Another example

- `#!/bin/bash`
- `clear`
- `echo "This is information provided by mysystem.sh. Program starts now."`
- `echo "Hello, $USER"`
- `echo`
- `echo "Today's date is `date`, this is week `date +%V`."`
- `echo`
- `echo "These users are currently connected:"`
- `w | cut -d " " -f 1 | grep -v USER | sort -u`
- `echo`
- `echo "This is `uname -s` running on a `uname -m` processor."`
- `echo`
- `echo "This is the uptime information:"`
- `uptime`
- `echo`
- `echo "That's all folks!"`



# #! “Sh-Bang”

- **First** line tells the interpreter
  - #!/bin/sh
  - #!/usr/bin/perl -wnl
  - #!/usr/bin/env python
  - Default is /bin/sh
- **SetUID not** honoured
  - Can't run with the owner's privilege.
- # is also used for comments

# Good scripts

- A sensible name
  - don't clash with existing commands and programs
- No errors
- Perform the intended task
- Have a clear logic
- Efficient, no unnecessary work
- Informative, notifying users about what it is doing
- Reusable
- In summary, it is just like a good program, except the scripts are written in commands.

# Linux BASH basics

- A popular command shell
- Files read by bash
  - /etc/profile, .bash\_profile, .bashrc
  - depending on login, interactive, non-interactive, or use **sh** directly
- Three types of commands
  - built-in, function, executable programs
  - Built-in commands like cd and eval, exit, exec, export,
- debugging a script: **bash -xv script\_file**
- Some self-study required
  - Read **Bash Beginners Guide**

# I/O Channels and Pipe

- `stdin`: standard input from terminal
- `stdout`: standard output to the terminal
- `stderr`: standard error to the terminal
- They are created for each process/command automatically and have file descriptors 0,1,2 respectively
- Commands can be joined with pipe |
  - The output of the first command becomes the input of the second command; uses system calls **pipe()** and **dup2()**.
- Example: find 5 biggest dirs in the current directory
  - `du -xkd 1 | grep -v "^[0-9]*[[:space:]]*\.$" | sort -rn | head -5`

# Command pipeline patterns

- Commands can be joined with pipe |
  - The output of the first command becomes the input of the second command; uses system calls **pipe()** and **dup2()**.
- Source: e.g. **ls**
  - read from file and write to stdout
- Filter: e.g. **sort**
  - read from stdin and write to stdout
- Sink: e.g. **less**
  - read from stdin and write to file
- “Cantrip”: e.g. **rm**
  - do something but return nothing
- Compiler: e.g. **tar**
  - read from file and write to another file

# I/O Redirection

- Standard input/output/error could be redirected to other files
- *command < f1\_in > f2\_out 2> f3\_err*
  - Redirect stdin to f1\_in, stdout to f2\_out, and stderr to f3\_err
  - *command > f1*, overwriting f1
  - *command >> f2*, appending to f1
  - *command 2> f3*, redirect stderr to f3
- Redirect stdout to stderr
  - *echo “Warning to stderr” >&2*
  - *echo “To black hole” 2> /dev/null >&2*

# Environment variables and files

- Environment variable
  - A variable with name and value used by shells and processes
  - Use `printenv` or `env` to find them
  - They can be set by
    - Globally, `/etc/profile`, `/etc/bash.bashrc`
    - Per user, `~/.bash_profile`, `~/.bashrc`, `~/.profile`
- `/etc/profile`, `~/.bash_profile`, `~/.bash_logout`
  - Used by login shells
- `/etc/bash.bashrc`, `~/.bashrc`
  - used by interactive, non-login shells
- Shell scripts use non login shell, non interactive shell
- For details [https://wiki.archlinux.org/index.php/environment\\_variables](https://wiki.archlinux.org/index.php/environment_variables)

# Variables in BASH

- *varname=value*
  - Assignment, no spaces around ‘=’
- *\$varname* for deference
- Global and local variables
  - Environment variables are global variables.
  - Variables by default are global after assignment
  - Local variables defined with keyword “local”
- Variables can be seen by subshell/child processes
  - *export PATH=\$HOME/bin:\$PATH*
- Beware white-space in string values
  - *Varname=“foo bar”*, using “” if there is white space



# Interpolation

- A built-in command in a string can be executed and the execution output will replace the location of the original command.
  - *'non-interpolated string'*
  - *`command`*
  - *"interp. string \$varname `command`"*
  - `foo=`command `command``
  - `foo=$(command $(command))` (Bash specific)
- Example
  - `echo -e "This is output from ls:\n`ls`:"`

# Conditions — if

- **if** `[_]$_# -lt 2` ]; **then** *if-less-than-two-arguments* **elif** `[_]\( [_ "$1" = 'foo' )_ -a _ \`  
`\( [_ -r _/etc/foorc _ )_ ]`; **then** *if-arg1-is-foo-*  
*and-foorc-is-readable* **else** *if-otherwise* **fi**
- **if** `!_ grep -q ...`; **then** *if-grep-did-not-*  
*find* **fi**
- Note: **man 1 test** to find more about if conditions

# Conditionals — case

```
case "$fo_proc" in    'fop')  
    command;;        'xep')  
    command1; commandN;;    *)  
    default-command >&2           exit  
1;; esac
```

# Loops — for

- **for** *i in foo bar baz* **do** echo \$i **done**
- (( ... ; ... ; ... )) is a Bash-ism **for** ((i=128; i<160; i++)); **do** printf “ip%03d\tA\t192.168.1.%d\n” \$i \$i **done**

# Loops — while

**ls | while read *filename* do**      *do stuff with*  
**“*\$filename*” done**

**while true do**    *infinite loop body*    **done**

# Arithmetic

- `expr 2 \* 8` 16
- `echo $((2 * 8))`      *Bash-ism* 16
- `echo 'scale=2; 1/3' | bc` .33
- `echo 'ibase=10; obase=2; 192' |  
bc` 11000000

# Sed and Awk

- Read a book!
- Regular expressions!
- Takes a while to learn
- A few recipes are useful



# List all system commands

- `find /bin /usr/bin /sbin /usr/sbin \ -type f  
-perm /111 | \ xargs -L1 basename | \  
xargs -L1 whatis | grep '([18])'`



# Applescript example

- Is 10% of disk available? [https://developer.apple.com/library/mac/documentation/applescript/conceptual/applescriptlanguide/conceptual/ASLR\\_lexical\\_conventions.html#//apple\\_ref/doc/uid/TP40000983-CH214-SW1](https://developer.apple.com/library/mac/documentation/applescript/conceptual/applescriptlanguide/conceptual/ASLR_lexical_conventions.html#//apple_ref/doc/uid/TP40000983-CH214-SW1)

```
tell application "Finder"      set the percent_free to  $\frac{\text{((the free space of the startup disk) / (the capacity of the startup disk)) * 100}}{\text{div 1}}$  end tell if the percent_free is less than 10 then      tell application (path to frontmost application as text)      display dialog "The startup disk has only " & the percent_free & " percent of its capacity available." & return & return & "Should this script continue?" with icon 1      end tell end if
```

# PowerShell examples

- This example is from *Monad Manifesto*
- What is filling up my application logs?
  - **Get-EventLog application|Group source|Select –first 5|Format-Table counter Property =====**  
===== 1,269 crypt32 1,234 MsiInstaller  
1,062 Ci 280 Userenv 278 SceCli

# Summary

- What is the least privilege principle?
- List a few pros and cons of shell scripting compared with other programming languages like C/C++.

# References

- *The Art of Unix Programming* Eric S. Raymond
- *The Unix Hater's Handbook* Simson Garfinkel, Daniel Weise, and Steven Strassmann
- ***Monad Manifesto*** Jeffrey P. Snover
- *Scripting: Higher Level Programming for the 21st Century* John K. Ousterhout (father of Tcl)
- Bash Guide for Beginners Machtelt Garrels
- [Reference] bash(1)