SecureSet Academy
Boot Camp Prep
Intro to Systems Security

Glenn Webb, CISSP, GSEC

Table of Contents

Session 1 - Introduction to Linux

Installing Linux in VirtualBox

During this session we will start with installing VirtualBox on your computer. VirtualBox is virtualization software which lets you run other operating systems on your computer.

Enable Virtualization

Before we start with the Linux installation we must verify that your laptop's Virtualization Extensions are enabled in the BIOS:

1. Reboot your computer
2. When you see the BIOS splash screen look for the key to press to enter the BIOS Setup menu
    a. It will probably be one of F2, F9, F11, F12
    b. Press the key in order to get into the BIOS
3. Find the menu option which allows you to enable Virtualization
4. Save and exit
    a. Typically F10
5. Boot up your laptop

Download VirtualBox

We will download the VirtualBox software from this website:
https://www.virtualbox.org/wiki/Downloads

1. Download the proper VirtualBox package (Windows or OSX) depending on the type of computer you have
2. Now download the VirtualBox Guest Additions
3. Install the VirtualBox software first (as Administrator)
4. Now install the VirtualBox Guest Additions (as Administrator)

Download and install Ubuntu

Note: you need to have a computer with at least 4GB of RAM to effectively run Ubuntu. Less than this will result in a very slow and sluggish system that will be very difficult to work with.

We will download the Ubuntu installation image from:
https://www.ubuntu.com/download

1. Download the latest Ubuntu Desktop installation image
2. Start VirtualBox Manager

3. Click the “New” button to create a new virtual environment
    a. Name: Ubuntu
    b. Type: Linux
    c. Version: Ubuntu (64-bit)
    d. Memory size: 2048 MB
    e. Hard disk: Create a virtual hard disk now
    f. Click “Create” button
    g. File location: Ubuntu
    h. File size: 32 GB
    i. Hard disk file type: VDI (VirtualBox Disk Image)
    j. Storage on physical hard disk: Dynamically allocated
    k. Click “Create” button.
4. In the left panel of the VirtualBox Manager software click on the VM and then click the green “Start” button
5. It will prompt you for a start-up disk
    a. Click the small folder with a green arrow icon on the right of the popup box
    b. Navigate to where you saved the Ubuntu installation ISO and select that for your startup disk
    c. Then click the “Start” button. The installation will begin
6. Click the “Install Ubuntu” button
7. Click the “Download updates…” and “Install third-party software…” check boxes then “Continue”
8. Click “Erase disk…” if necessary then “Install Now”
9. Click “Continue” to write the changes to disk
10. Click on the map to set your location to “Denver” then “Continue”
11. Accept the defaults of “English (US)” keyboard then “Continue”
12. Your name: *enter your first name in lower-case*
13. Your computer’s name: *this will be auto-populated*
14. Pick a username: *this will be auto-populated*
15. Choose a password: enter a password you can remember
16. Confirm your password: re-enter the password
17. Click “Continue” and let the installer run
18. Click “Reboot” when the installation is finished
19. Additionally, press “Enter” if necessary

First Login and Updates

1. When your VM (virtual machine) is rebooted log in with the username and password you created during installation
2. Tap your “Windows” key on your keyboard if your computer is running Windows or the “Command” key if your computer is a MAC

   a. If neither of these options work click on the round white and purple button to bring up the search utility
3. Type **term** in the search window then click on the Terminal icon which is displayed
4. In the terminal window type: `sudo apt-get update` followed by your password
   a. This will run a command which updates the list of available software packages
5. Click on "Devices" in the top-most menu
   a. select "Insert Guest Additions CD Image…"
   b. A popup will ask you if you want to run the software on the Guest Additions CD, click "Yes"
   c. Once again, type in your password then click "Authenticate"
   d. The guest additions will be built and installed
   e. Next press "Return" to close the window
6. Back in the terminal window type `sudo poweroff` to shutdown the VM

Installing Guest Additions

1. Click on the VM in the left panel of the VirtualBox Manager
2. Click the "Settings" button
3. A "Settings" window will be displayed
4. Click on "General" in the left panel then click the "Advanced" tab in the right panel and change these settings:
   a. Shared Clipboard: Bidirectional
   b. Drag'n'Drop: Bidirectional
   c. Then click "OK"
5. Click "Start" to startup your VM
6. Log in when your VM is finished booting

Exploring Your Desktop

1. Press the Settings icon in the top far-right corner of the screen
   a. Then select "About This Computer"
   b. Explore the information contained in the new popup window
   c. How much memory does your VM contain?
   d. What kind of Processor does your VM have?
   e. Read the "Legal Notice". Is there anything to be concerned about?
2. Clicking on the "All Settings" tab gives you many more areas to explore
   a. To return back to the previous screen click on the "Details" icon

Computer Architecture

Your computer is comprised of many components which serve different purposes within the system. We will discuss a few of the more prominent ones.

CPU

What is a CPU? It is the central processing unit (or processor) of a computer. The CPU carries out the instructions of a program by performing control, input/output, arithmetic, and logical operations. It is the brains of your computer.

A CPU implements an instruction set for carrying out computations. An instruction set is the set of operations the CPU can perform. Some examples of instructions are

- Add
- Subtract
- Multiply
- Divide
- Compare
- Jump
- Increment
- Decrement

Some CPU’s like Intel and AMD have compatible implementations of instruction sets

- The x86 instruction set was created by Intel based on the 8086 CPU and implemented by other CPU manufacturers like AMD, Cyrix, and VIA
- The x86_64 instruction set, which is a 64-bit extension of the x86 instruction set, was created by AMD and implemented by other companies like Intel and VIA

Components of a CPU

- Arithmetic Logic Unit (ALU) which performs the arithmetic and logical operations
- Control Unit (CU) which retrieves instructions from memory and decodes and executes them. The CU calls on the ALU when necessary
- Registers are small memory locations in the CPU which can hold data. The registers come in increments of 8 bits. 8 bits is the same as 1 byte. For example the registers in a 32-bit system are 4 bytes in length and the registers in a 64-bit system are 8 bytes in length.

What is the difference between 16-bit, 32-bit, and a 64-bit CPU’s?
In the 1960’s through 1970’s 16 bit processors were being developed for use with the earliest computers. In June 1978 Intel release a 16-bit CPU called the 8086 for the fledgling home PC market. The term 16-bit/32-bit/64-bit CPU refers to:

- The ***primary meaning*** refers to the size of the CPU registers. A register on a 16-bit CPU can store 16 bits of data; a register on a 32-bit system can store 32 bits of data; a register on a 64-bit system can store 64 bits of data
    - A bit is the smallest unit of data in a computer. It can hold only 2 values: 0 or 1
- The size of addressable memory
- The range of values which can be stored in 16 bits (binary) is `00000000 00000000` to `11111111 11111111` which equals 0 through 65535.
- The range of values which can be stored in 32 bits (binary) is `00000000 00000000 00000000 00000000` to `11111111 11111111 11111111 11111111` which equals 0 through 4294967295.
- The range of values for a 64-bit address is similar to the two examples above: 64 zeros through 64 ones. The range of values for a 64-bit binary number 0 through 18446744073709551615 (16 exabytes)

The effects of having a 64-bit CPU vs. a 32-bit CPU is that the computer can access **MUCH MORE** memory (RAM) and the data travels down the busses quicker because the data path is wider. 64-bit CPU's are the norm for all computers, cell phones, and tablets.

In the PC market Intel x86_64 processors have the most market share. Windows computers, MACs, and Linux all run on Intel x86_64. The x86_64 designator specifies the CPU instruction set architecture.

Run the following commands to find your processor type:

1. On Windows, in a cmd window type `set` and press enter. Look for the `PROCESSOR_IDENTIFIER` and `NUMBER_OF_PROCESSORS` fields
2. On Mac, in a terminal window type `% sysctl -n machdep.cpu.brand_string` and press enter
3. On Linux, in a terminal window type `cat /proc/cpuinfo` and press enter

Memory

Computer memory called RAM (Random Access Memory) is a form of data storage which your computer uses while it is powered on. RAM is volatile memory which means it loses the values stored in it when the power is removed from the system. The amount of memory accessible by programs running on your computer is dependent on the instruction set. If the computer is a 32-bit computer it has a 32-bit instruction set and the amount of memory available to programs is $2^{32}-1$ = 4294967295 (4 GB of memory). If a computer is a 64-bit computer it actually only allows $2^{48}-1$ = 281474976710655 (256 TB of memory) in the virtual address space.

Disk Drive

A disk drive is a storage mechanism where data is recorded on various types of rotating disks. There are optical disks (CDs and DVDs), disk drives (hard drives), and floppy drives (but they have been obsolete for decades now). Disk drives are non-volatile so they retain the data stored on them after power is removed from the system.

Thumb Drive

Also known as a USB flash drive is another data storage device which uses non-volatile flash memory. Thumb drives are often used to transport files between systems because of their ease of use and compatibility with almost all computers.

Keyboard and Mouse

These are the typical and most often used input devices for a computer. Laptops have keyboards with a mouse device integrated into them.

Operating Systems

Definition of an Operating System

An operating system is the base software which manages the hardware components and software resources of a computer system. All programs which users interact with require an operating system to run upon.

History of UNIX

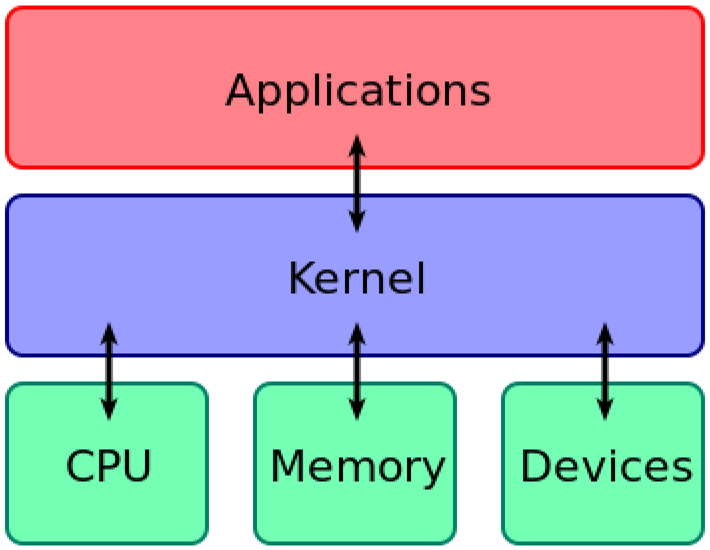
- UNIX was developed in the 1970’s by AT&T Labs and the source code was leased out to companies and universities.
- Solaris is a UNIX implementation developed by Sun Microsystems (now Oracle Corp.) based on the UNIX specification
- HP/UX is a UNIX implementation developed by Hewlett-Packard Corporation based on the UNIX specification
- FreeBSD (the core of Apple's’ OSX operating system) was developed by the University of California at Berkeley back in the 1970’s based upon the Unix source code leased from AT&T labs. FreeBSD does not contain any AT&T source code, the code from AT&T was rewritten and made open-source and thus FreeBSD is a free distribution like Linux
- Minix was a small UNIX-like operating system written by a Computer Science professor in the Netherlands named Andrew Tanenbaum. A book about the AT&T UNIX source code (version 6) had been written by John Lions (University of New South Wales) which university professors around the world had been using to teach their students about UNIX. When AT&T found out that their UNIX source code was being used to teach university students they changed the license agreement in version 7 to prohibit the use of the source code for teaching purposes. Because of this restriction Tanenbaum wrote a completely new operating system called Minix based on his knowledge of the AT&T source code for use in his teaching and for his students to learn from. Minix is still around and is free and open source.

- Linux was written by Linus Torvalds (a college student at the University of Helsinki). Linus gained inspiration for the Linux project by studying the source code of Minix. Linus wrote the Linux kernel from scratch. Linux does not use or borrow any of the AT&T source code.
- https://en.wikipedia.org/wiki/Unix#/media/File:Unix_history-simple.svg

Other Operating Systems

- Android is the most used operating system on mobile devices in the world. It is based on the Linux kernel. The Android operating system is open source and has been released by Google under an open source license.
- iOS is the an operating system developed by Apple for use on their devices. iOS is the second most popular operating system for mobile devices.
- Windows is a closed source (proprietary) operating system from the Microsoft Corporation. There have been many versions of Windows over the years. Windows 7, 8.1, and 10 are all very prominent today.

The Linux Kernel



https://en.wikipedia.org/wiki/Kernel_(operating_system)#/media/File:Kernel_Layout.svg

A kernel is the core program of an operating system which manages the hardware of the computer and provides interfaces for applications running on the system. The kernel is the first thing loaded on system startup and takes over the remaining tasks while the system is booting up. All operating systems have a kernel; operating systems like Windows, OSX, and Linux have differing types of kernels.

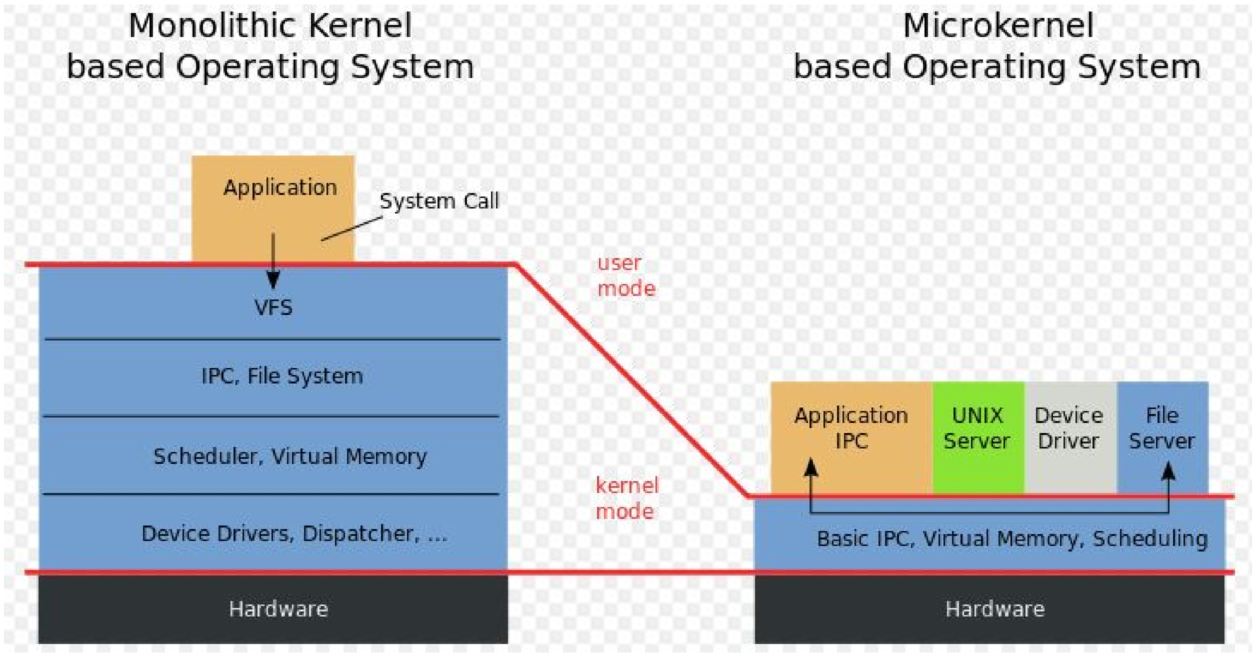
Computers are made to process data and preserving data integrity is the kernel's highest priority. Having a computer system that does not protect data integrity is a useless system. What good is using a system when you are not guaranteed that the data in the system will be protected?
Because of the importance of this task, the kernel manages access to the data on a computer; regular users are not allowed unmanaged access to computer data. The kernel runs in a protected area of memory called *kernel space*. This area of memory is protected from *user space* programs. Remember, the kernel is used to manage system resources like CPU, memory, disk drives, but its number one priority is preserving data integrity. Have you ever seen a "Stop Error" screen (aka BSOD - blue screen of death) on Windows or a kernel panic on Linux?

> A kernel panic also known as computer death or PC death, is a safety measure taken by an operating system's kernel upon detecting an internal fatal error in which it either is unable to safely recover from or cannot have the system continue to run without having a much higher risk of major data loss.
>
> https://en.wikipedia.org/wiki/Kernel_panic

The Linux kernel was developed by Linus Torvalds in 1991. As the kernel quickly matured (with the help from other developers) many pieces of software were integrated into Linux from the GNU project. The GNU project is a Unix-like operating system started by Richard Stallman (from MIT) in 1983. The GNU project had many operating system utilities like shells, core utilities, compilers, libraries, etc) already developed but the GNU kernel was not complete. The completed utilities of GNU were combined with the Linux kernel to form a completely free and open source operating system called GNU/Linux.
Linus took advantage of tools which were available to him while developing the Linux kernel. For example, Linus used the Minix operating system for developing the Linux kernel. He took the source code for the GNU gcc compiler and compiled it on Minix using the Minix cc C compiler. Then he used the gcc compiler to develop the Linux kernel.

Monolithic and Micro Kernels

The Linux kernel is *monolithic* meaning that the entire kernel (kernel, resources, and device drivers) runs in the privileged memory space and the kernel runs in one process address space. OSX and Windows NT use a *microkernel*. A microkernel is a kernel which offers only a few services and it runs in privileged memory space. The other services and device drivers run as services in unprivileged/user memory space. This makes the microkernel (theoretically) more secure than monolithic kernels since a buggy device driver cannot crash the entire kernel. Microkernels are generally a few percent slower than monolithic kernels.


Monolithic Kernel based Operating System
Microkernel based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user mode
kernel mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware

https://en.wikipedia.org/wiki/Microkernel#/media/File:OS-structure.svg

Processes and Process ID's

In Linux a process is a running instance of a computer program. Running processes are owned by the user who launches the process and the processes are assigned a process identifier (PID). The ps command lists currently running processes. In a terminal window run

```
ps -ef | less
```

This command lists the programs currently running on your Linux system. The first column shows the owner of the process (user ID - UID), the second column shows the PID, the third column shows the Parent PID (PPID), and the last column shows the name of the running process.

At boot-up the Linux kernel (PID 0) starts up a process called init (PID 1) and a thread daemon (kthreadd PID 2). kthreadd is used to start up certain kernel threads running kernel code which are managed as processes. init is responsible for launching critical system services. Run the following commands to see the process trees of kthreadd and init.

```
pstree 1 | less
pstree 2 | less
```

New processes are created when a parent process "forks" a new process. There are two important commands used in forking (duplicating) a process; they are `fork` and `exec`. For example if you are in a terminal window and you type `echo $$` and press enter it will show you the current PID of the terminal window you are running in. Now type `xclock` and press enter; you will see a new application which displays a clock.

In a separate terminal window type

```
pstree -a PID from the echo $$ you just ran
```

This will show you that `bash` is the parent of `xclock`. When you run the `xclock` command the `bash` process calls `fork()` which creates a new process which is an exact duplicate of the current `bash` process. Next, the `exec(xclock)` command is called; this command reads the `xclock` program from disk and replaces the contents of the new `bash` process with `xclock` program and begins running the new `xclock` process.
This process is important to remember! To start a new process Linuxs `fork()`'s then `exec()`'s.

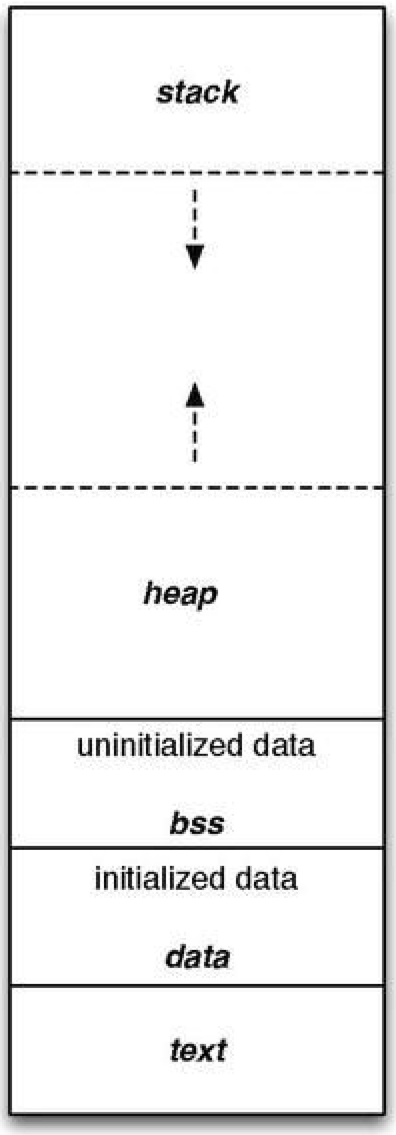
Daemons

A daemon is a long-running process which executes in the background and is not interactively controlled by a user. Daemon process names often end with a 'd' to separate them from other processes. Daemons are usually started at boot up and they remain running until the system is shut down. A few examples of daemons are rsyslogd (which handles system log messages), sshd (which handles incoming ssh login requests), and httpd (which handles incoming requests for Web pages).

Virtual Memory

When processes are started in Linux they have a process address space allocated to them but they do not directly access physical memory. Linux is a virtual memory operating system meaning that memory presented to system processes have a virtual address which is mapped to a physical memory address. These mappings between virtual address and physical address are maintained in kernel memory mapping tables.
Theoretically a process on a 64-bit OS has a virtual memory address range from 0 to $2^{64} - 1$.

Two raised to the 64th power is an incredibly huge number, approximately 18 Quintillion or 16 Exabytes. There are no computers with 16 Exabytes of RAM. Processes on 64-bit operating systems use a notation which theoretically allows that much memory but it is never actually realized. Currently Intel and AMD 64-bit processors only use the lower 48 bits of the 64-bit address space for process memory. This is still a huge number, 256 Terabytes for process address space. Since current computers do not have physical memory this large virtual memory mapping is used. As a process runs, any memory which is needed is dynamically allocated and the virtual-to-physical mapping is done during runtime. Each process has its own virtual address space and thus they are protected from the actions of other processes. The kernel maintains the integrity of each process's memory by allocating a dedicated address space to each process and processes are only allowed access to their own virtual memory.



The **stack** is the memory portion of the program which contains automatically created variables and variables which belong to a function invocation. The stack is typically allocated at the end of the memory address space and as things are added to the stack it grows downward. The stack is a last-in-first-out construct
The **heap** area is placed below the stack with a large unallocated area between them. The heap contains memory allocated with functions like malloc, calloc, and realloc. Memory is released from the heap by using the free() function.
The **bss** (block started by symbol) area contains global and statically allocated variables which are uninitialized (set to zero by default)
The **data** area contains global or static variables which have a predefined value or can be modified. These would be variable defined outside the scope of functions or declared inside a function with the static keyword.
The **text** area contains the executable instructions of the program. This area is usually fixed size and read-only.

https://en.wikipedia.org/wiki/File:Program_memory_layout.pdf

Filesystems

The term filesystem means two things in the context of Linux. The first usage describes the methods and data structures that an operating system uses to keep track of files on a disk or a disk partition; that is, the way the files are organized on the disk.
The second usage describes an instance of the filesystem (first meaning) on a disk or disk partition. One might say they have two filesystems on their disk drive (if it is divided into two partitions and each partition contains a filesystem).
Most applications on Linux computers interact with the filesystem; they do not interact directly with the raw sectors of a disk drive. One exception would be the Linux program `mkfs` (make filesystem) which creates the filesystem on the raw disk device.
`mkfs` initializes the raw disk and creates the bookkeeping data structures on the disk which are necessary for the filesystem to work. The bookkeeping data structures written to a disk include:

- Superblock - contains information about the filesystem as a whole, like the size
- Inode - contains all the information about a file except its name. The inode contains the number of the data blocks used by the file.
- Data block - the actual file data is stored in data blocks
- Directory block - filenames are stored in a directory along with the number of the inode corresponding to the file
- Indirection block - pointed to by an inode when the inode does not have enough space to hold all the numbers of the data blocks comprising a file

Linux systems do not have the concept of drives (c:\ or d:\) like Windows. Instead it has a single hierarchical directory structure designated by a forward slash `/` and known as the root partition. Everything in the Linux filesystem starts from this root partition and expands into subdirectories. A filesystem on a disk or disk partition is not accessible until it is mounted by the kernel. The kernel first mounts the root partition at bootup and other partitions are mounted on mount points contained on the root filesystem. When a Linux system shuts down gracefully (by using init 0, poweroff, shutdown) it writes any in-memory cached filesystem data to the physical disk before it unmounts the partition. One of the implications of this method is that you should always gracefully shutdown your Linux system. You risk corrupting your filesystem if you just power off your system.

http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/foreward.html

Session 1 Homework

Use Google to look up the Usage Share of Operating Systems on the internet. Find the percentage breakdown of operating systems for:

- Desktop and laptop computers
- Mobile devices
- Public servers on the internet

Bring your research results to the next class period and we will share and compare the answers we got to see if we can come up with a consensus for these figures.

References:

https://en.wikipedia.org
http://www.webopedia.com/quick_ref/computer-architecture-study-guide.html
http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf
http://www.tldp.org/LDP/abs/abs-guide.pdf

Session 2 - Shells and Programming Languages

Shells

The language of modern computers is the binary numbering system (1's and 0's). All the processing that computers do and the data they work with is in binary. Linux shells are applications which allow humans (who natively do not speak binary) with computers (who natively do speak binary). A shell is one of the tools we have which allow humans to tell the computer what they want it to do. A shell takes human readable commands and translates them into commands the kernel can understand and process.
When you open a terminal window on Linux, or when you remotely login to a Linux machine using SSH a shell gets started. From this shell you can type commands and interact with the computer.
There are several shells available on Linux. To see which shells you have on your Linux system type `cat /etc/shells` in a terminal windows and then press enter. We will concentrate on `bash`, the most popular shell on Linux.

Basic Commands

Command	Result
`pwd`	Print working directory. The directory you are currently in
`cd` *`optional path`*	Change directory to your home directory or the directory given after command
`ls` *`optional path`* Other options: `ls -l, ls -la, ls -lrt`	List the current directory contents or the contents of the directory given after command
`cat filename`	Print the contents of the given file
`date`	Prints the system time
`mv old_filename new_filename`	Renames a file
`cp original_file new_file`	Creates a copy of a file with a new name
`rm filename`	Deletes a file
`echo "some words"`	Prints a string to standard output
`mkdir some_dir, rmdir some_dir`	Creates or removes a directory

`touch`	Creates an empty file (or update last change time to current for existing files)
`df -h`	Show the amount of storage on a filesystem
`grep`	A command line search utility
`man command_name`	Prints the manual for the command given
`whoami`	Prints the username
`printenv`	Prints the environment variables in your shell

File Matching metacharacters

Symbol	Result
`*`	Matches any number of characters
`?`	Matches exactly one character
`[...]`	Matches any one of the characters in brackets which can be a comma separated list or a hyphen-separated range

In-class activity 1

In a terminal type

`cd ~` *this command puts you in your home directory*

```
mkdir wordtest
cd wordtest
touch rat cat bat splat drat ration duration rats
ls ?at
ls *at
ls ??at
ls *ration
ls *at*
ls *[pt]*
echo "good morning" > greeting
echo "good evening" >> greeting
cat greeting
cd ..
mv wordtest testwords
rm -r testwords
sudo cat /etc/passwd
```

Find your account in the list and you will see your default shell at the end of the line.

The following are very good resources for learning Bash
http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html
http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

Bash Environment Variables, Pipes, and Redirection

The Bash shell environment has variables which customize and enhance the operation of the shell. Environment variables are typically set in the files described next.

When you log into a system using a Desktop Environment and then open a terminal window you are creating a **non-login** shell. It is a non-login shell because you have already logged into the system. Non-login shells run the `~/.bashrc` file at startup. The tilde `~` is a shorthand way of expressing the path to your home directory.

When you open a terminal window you have a set of environment variables pre-set for you. These come from the `~/.bashrc` file. Modifying this file is one way you can customize your environment.
Example of a `~/.bashrc` file

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
PATH=$PATH:$HOME/bin

export PATH

# The PS1 variable is how you change your prompt
PS1='$PWD> '

# If I was running an Oracle database I would need something like
this
ORACLE_SID=testdb
export PS1 ORACLE_SID
```

When you remotely connect into a system you are creating a **login shell** and the `~/.bash_profile` file gets run at startup. Modifying this file is another way you can customize your environment. Example of a `~/.bash_profile` file

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
```

In-class activity 2

`sudo apt-get install vim`
Work through the `vipractice.txt` file. The instructor will step you through this introduction to the `vi` editor.

In-class activity 3

In a terminal window run the following commands:
```
sudo apt-get install ssh
vi ~/.bash_profile
```
add this line to the file
```
export bash_profile_variable="this was set by the .bash_profile file"
```
save the file and exit
```
sudo apt-get install ssh
Do you want to continue? [Y/n] y
ssh localhost
```
provide your password
```
printenv | grep bash_profile_variable
```
What you've done is log into your own machine on a loopback interface. And shown that when you log in remotely the `~/.bash_profile` file is read and interpreted.

Different ways to make the computer do what you want it to

1. You could write programs in a very low level language like assembly. For example to make the computer say "Hello World" you could do the following:
   Start a terminal window
   type: `vi helloworld.s`

```
.section .data
     hello:
          .ascii "Hello world!\n"    # output string

.section .text
     .globl _start
     _start:
     movl $4,%eax          # system call ID: sys_write
     movl $1,%ebx          # file descriptor for standard output
     movl $hello,%ecx           # string address
     movl $13,%edx         # length of the string
     int $0x80             # generate interrupt
     movl $1,%eax          # system call ID: sys_exit
     movl $0,%ebx          # exit code 0 success
     int $0x80             # generate interrupt
```

   assemble the program: `as -o helloworld.o helloworld.s`
   link the program: `ld -o helloworld helloworld.o`
   run the program: `./helloworld`

2. You could write programs in a higher level compiled language like C.
   Start a terminal window
   type: `vi helloworld.c`

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

   compile the program: `gcc -o helloworld helloworld.c`
   run the program: `./helloworld`

3. You could write programs in another higher level intermediate compiled language like Java.
   Start a terminal window
   type: `vi helloworld.java`

```
public class HelloWorld {
    public static void main(String[] args) {
        // Prints "Hello, World" to the terminal window.
        System.out.println("Hello, World");
    }
}
```

compile the program: `javac helloworld.java`
run the program: `java ./helloworld`

4. you could write programs in an interpreter like Bash
   Start a terminal window
   type: `printf "Hello World!\n"`

5. you could write programs for another interpreter like Python
   Start a terminal window
   type: `python`
   type: `print "Hello World!\n"`

Interpreted and Compiled Languages

In the examples above we saw:

Compiled into machine code languages - assembly and C

When code written in a language is compiled, its syntax is transformed into an executable form (machine code, 1's and 0's) before running. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages.

Compiled into intermediate representations languages - Java

When code written in an intermediate representation language is compiled, its syntax is transformed into byte code (similar to assembly). The byte code must then be interpreted or further compiled to execute it. This is done by a Virtual Machine.

Interpreted languages - Bash and Python

When code written in a language is interpreted, its syntax is read and then executed directly, with no compilation stage. A program called an *interpreter* reads each program statement, following the program flow, then decides what to do, and does it.
https://en.wikipedia.org/wiki/High-level_programming_language

So why not just write everything in an interpreted language? ***Speed and Efficiency***
Compiled programs run much faster than interpreted programs and they use less memory.
What language do you think the Linux kernel is written in?

Session 2 Homework

Try variations on all the the commands located under Bash Basics, File Commands, and Directory Commands in the Bash Cheat Sheet. Save the outputs to a file to turn in to show what you tried. Make note of which programs are not installed and any other observations you have. Turn in your homework by emailing it to your instructor.

Session 3 - Linux Security and Files

Users, groups, and passwords

System accounts are stored in `/etc/passwd`; system groups are stored in `/etc/group`; account passwords are stored in `/etc/shadow`. Originally the passwords were stored in the second field in `/etc/passwd` but as hackers began brute force cracking the encrypted passwords in `/etc/passwd` it became evident that the passwords needed to be moved to a restricted file. The file `/etc/shadow` now contains the passwords and is restricted to root only.

Run the following commands to see the permissions on those files:

```
ls -l /etc/passwd
ls -l /etc/shadow
ls -l /etc/group
```

Types of users on Linux Systems

There are two types of users on Linux systems:

- system users (UID from 0 to 999). These are accounts for administrative tasks and also accounts which are associated with system services. Some system user accounts have higher privileges than other users. For example, the root account has full system privileges
- regular users (UID from 1000 to 60000). These accounts are for regular users of your system. These accounts have limited privileges.

Groups

User accounts are also assigned to *groups*. Groups are a way of organizing associated users into logical divisions. For example, a server at a large company may have an accounting group, a human resources group, an engineering group, and a management group. Your group membership on the server will depend on which department you work in.

User accounts on the system are listed in `/etc/passwd` and groups on the system are listed in `/etc/group`. Passwords for accounts are stored in `/etc/shadow`. Every user on the system is assigned to at least one group and some users are assigned to multiple groups.

Run the following commands:

```
cat /etc/passwd
```

Verify that system users have a UserID between 0 and 999
Verify that regular users have a UserID between 1000 and 60,000

```
cat /etc/group
```

Verify that system users are assigned to groups

You can view your own UID (userid) and GID (groupid) by typing in a terminal window

```
id -a
```

Permission Bits

After you have been introduced to Linux and the command line you will surely notice that files and directories have permission settings and ownership settings. This section will explain the Linux ownership and permissions model.

One of the most important features built into a multi-user system like Linux is the protection of data from unauthorized users. Linux has very robust data protection features built in and these are directly related to any discussion on the **CIA** aspects of cyber security.

File Permissions and Ownership

When listing file attributes with the `ls -l` command you will see 10 characters at the beginning of each output line with the format similar to `-rwxr-x-r-x`, `drw-rw-r--`, or `lrwxrwxrwx`. The first character will be

- A dash `-` to signify a file
- A `d` to signify a directory
- An `l` (lowercase L) to signify a symbolic link
- A `b` to signify a block device
- A `c` to signify a character device
- An `s` to signify a socket
- A `p` to signify a named pipe

The remaining nine characters are broken up into groups of three characters (`rwx`) called octets

- Each group of three characters signify the read, write, and execute permission for the file owner, group assigned to the file, and all others (respectively)
- If a dash appears inside one of these groups instead of the letter, it means the permission for that associated letter is turned off

Other important details about File Permissions

- You may see an `s` where you would normally see an `x` in the user or group bits like so, `-rwsr-xr-x`, or `-rwxr-sr-x`, or both `-rwsrwsr-x`. When the `s` appears it means any user can launch the command but the ownership of the running command is assigned to the application user/group instead of the user launching the command. This is referred to as SetUID (when the `s` appears in the user octet) and SetGID (when the `s` appears in the group octet)
- If a `t` appears at the end of a directory listing it means the sticky bit is set on that directory. For example `drwxrwxr-t`. The owner of a directory can set the sticky bit on a directory where other users can create files in order to prevent them from deleting files which do not belong to them. Some Linux distributions have the sticky bit set on the `/tmp` directory because all users can write files there but they are not allowed to delete files which they do not own.
- If you see a plus sign (+) or a period (.) at the end of the permission bits it means that the file has extended SELinux (Security Enhanced Linux) settings or ACL (access control list) settings.

Examples:

```
ls -l /usr/bin/date
-rwxr-xr-x. 1 root root 62200 Nov  5  2016 /usr/bin/date
```

This example shows that owner root has read-write-execute permissions on the date command, the root group has read-execute permissions on the file, and all other users have read-execute permissions on the file.

```
cd ~
ls -l /home/mary/
-rw-rw-r--.  1 mary mary   65 Jul  1 20:43 poem.txt
```

This example shows that owner mary has read-write permissions on the poem file, the mary group has read-write permissions, and all other users have read permission.

```
ls -l my_first_poem
lrwxrwxrwx. 1 mary mary 10 Jul  1 20:50 my_first_poem -> ./poem.txt
```

This example shows that mary has a symbolic link pointing to her poem file. Note: permissions on symbolic links are not taken into account by the file system. Modifying the permissions on a symbolic link will change the permissions on the file pointed to by the symbolic link. Although it appears that any user can modify the link (because of w in other grouping) only the owner of the link and root user may modify the symbolic link.

Because files and directories are different from each other, the read-write-execute permissions for them mean different things. The table below highlights each meaning

Permission	File	Directory
Read	View the contents of the file	View the files and subdirectories it contains
Write	Change the file contents, rename the file, or delete the file	Add or remove files and subdirectories to the directory
Execute	Run the file as a program	Change to the directory with the `cd` command, execute a program in the directory, view file details of files in the directory

Changing Permissions with `chmod` command

The command `chmod` (which means "change mode") is used to change the way a file or directory can be accessed. There are two ways to use this command, using number or letters, and we will review both methods.
If you own a file or are the root user you can change the permissions on a file or directory any way you please. Recall that the nine permission characters are broken into groups of three. Each character has a numeric value as follows, `r = 4, w = 2, x = 1, -(dash) = 0`. Each octet of 3 characters (`rwx`) can have numeric values from 0 to 7 (a total of 8 possible values). This is why the group of 3 characters is called an *octet*. So the octet `rwx` would have the numeric value of 7 (4 + 2 + 1). This octet `rw-` would have the numeric value of 6 (4 + 2 + 0) and so on.
Here are some examples of using the `chmod` command with numeric values:

Original file

```
ls -l my_script
-rw-rw-r--.  1 mary mary   65 Jul  1 20:43 my_script
```

`chmod 777 my_script`, results in `-rwxrwxrwx`
`chmod 755 my_script`, results in `-rwxr-xr-x`
`chmod 644 my_script`, results in `-rw-r--r--`
`chmod 000 my_script`, results in `----------`
`chmod -R 755 my_apps_dir`, results in all files and directories below, and including the `my_apps_dir` being recursively changed to `rwxr-xr-x`.

The other method of using the chmod command involves using letters and plus (`+`) and minus (`-`) signs. You can change the read, write, and execute bits for the (`u`) user or owner of the file, (`g`) the group assigned to the file, (`o`) other users, and (`a`) all users.
Here are some examples of using the `chmod` command with letters:

Original file

```
ls -l my_script
-rw-rw-r--.  1 mary mary   65 Jul  1 20:43 my_script
```

Command	Results in
`chmod a-w my_script`	`-r--r--r--`
`chmod a+x my_script`	`-rwxrwxr-x`
`chmod u+x my_script`	`-rwxrw-r--`
`chmod o+w my_script`	`-rw-rw-rw-`
`chmod ug+x my_script`	`-rwxrwxr--`

Setting the SetUID, SetGID, and sticky bit
A numeric mode consists of one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Omitted digits are assumed to be leading zeros. The first digit selects the SetUID (4) and SetGID (2) and sticky bit (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

For example to set the SetUID and SetGID permissions on a file
`chmod 6775 my_script` would yield `-rwsrwsr-x`

To set the sticky bit on a directory
`chmod 1777 /home/mary/my_temp_dir` would yield `drwxrwxrwt`

When a regular user creates a file it is given a default permission of `-rw-rw-r--` and a newly created directory is given a default permission of `rwxrwxr-x`. Linux systems have a setting called umask which you can see by typing `umask` in a terminal window. It is typically set to `0002`. A non-executable regular file with wide-open permissions would be `666` or `rw-rw-rw-` and a directory with wide-open permissions would be `777` or `rwxrwxrwx`. If you ignore the first zero of the umask value, you would have `002`. Subtracting the umask value from a wide-open setting for a file would be

```
 666
-002
------
 664  ⇒ rw-rw-r--
```

Subtracting the umask value from a wide-open setting for a directory would be

```
 777
-002
------
 775 ⇒ rwxrwxr-x
```

This example shows the results of creating new files or directories with a umask value of `022`

```
 666
-022
------
 644  ⇒ rw-r--r--

 777
-022
------
 755 ⇒ rwxr-xr-x
```

Changing file ownership with the `chown` command

The root user can change the ownership of any files on the system. Regular users cannot change the file ownership of the files they have created. The following examples (which are run as root user) show how the root user can use this command.

Original file

```
ls -l my_script
-rw-rw-r--.  1 mary mary   65 Jul  1 20:43 my_script
mv /home/mary/my_script ~/roots_script
chown root:root ~/roots_script
```

Original directory

```
ls -ld /media/myusb
drwxr-xr-x. 2 root root 6 Jul  2 15:57 /media/myusb
chown -R mary:mary /media/myusb
```

Session 3 Homework

Try variations on all of the commands located under Input/Output Redirectors and Process Handling in the cheat sheet. Remember to put a command in front of the redirection symbol. For example: `ls -la > files_in_my_dir`

Save the outputs to a file called homework3.txt to turn in to show what you tried. Attach this file in an email to your instructor with "Homework 3" in the subject line.

Also make note of which programs are not installed.

REMEMBER: if you are having difficulty getting a command to work read the man page.

Session 4 - Shell Scripting

Bash Scripting

What is a shell script? In the simplest terms a shell script is a text file which contains a sequence of shells commands to be run by the computer. The commands in a shell script are the same kind of commands you would type in while interacting with the shell. In our case we will be learning to write Bash scripts.
Why write shell scripts? There are several reasons to write shell scripts:

- they allow you to create tools which you can use over and over
- they are useful for automation of repetitive tasks
- they are useful in rapid prototyping of program designs which will be rewritten later in a more suitable programming language
- your Linux system uses shell scripts extensively while it is booting so being able to write and modify shell scripts will be useful if you wish to become proficient at system administration

Simple Hello World shell script

```
vi simple1.sh
```

```
printf "Hello World\n"
```

```
bash ./simple1.sh
```

Modified Hello World script

```
vi simple2.sh
```

```
#!/bin/bash

# Author: John Doe
# Purpose: This script will greet the user
# Version: 1.0

read -p "Please enter your name: " myname
printf "Hello %s, pleased to meet you.\n" $myname
```

```
chmod +x simple2.sh
./simple2.sh
```

Simple2 script explanation

1. The first line in the script is the “interpreter directive” and effectively tells the system what shell (or other interpreter) to use when this script is run. The first two characters `#!` (sometimes called 'shebang') are a *magic number*. Magic numbers are a series of a few characters at the beginning of a file which the operating system uses to determine what type of file it is. There are many different magic numbers corresponding to the many file types on a Linux system.
2. The lines beginning with a pound sign `#` are comments. The comment which starts after the `#` are ignored by the shell. The comment extends to the end of the line.
3. The read command prints a prompt for the user using the -p parameter and stores the input value into a variable called myname. You should give your parameters meaningful names. Parameter can be uppercase, lowercase, mixed, and contain numbers, but they may not start with a number. Some good examples are: firstname, lastname, name1, filename2, currenttime, userinput, first_name, last_name, file_name, curr_time, user_input, etc…
4. The printf command prints a string with a format specifier (`%s`) to standard output (stdout). When the string is printed the format specifier is replaced with the value stored in the second parameter.

sysinfo script

```
vi sysinfo.sh
```

```
#!/bin/bash

# Author: John Doe
# Purpose: This script will print out information about the system
# Version: 1.0

clear
printf "Hello, $USER\n"
printf "\n"
printf "Today's date is $(date), this is week $(date +"%V").\n"
printf "\n"
printf "These users are currently connected:\n$(who)\n"
printf "\n"
printf "This is $(sed -n '/PRETTY_NAME/s/.*=//p' /etc/os-release) "
printf "running on an $(uname -m) processor.\n"
printf "\n"
printf "This is the uptime information:\n"
uptime
printf "\n"
printf "This is the physical RAM information:\n"
cat /proc/meminfo | grep MemTotal
printf "\n"
exit 0
```

```
chmod +x sysinfo.sh
./sysinfo.sh
```

sysinfo script explanation

1. the $USER variable is an environment variable. Environment variables are available for use within shell scripts.
2. Commands between $(...) run in a child process and their output is inserted into the string.
3. sed is a stream editor we will only mention briefly
4. `/proc/meminfo` is a virtual file which contains system memory information
5. `exit 0` is an exit status returned to the calling environment indicating success

In-class activity 1

Work through the vipractice.txt file provided by your instructor. The instructor will guide the class together through this exercise.

In-class activity 2

Using `simple2.sh` as an example, write your own Bash script named `greeting.sh` which

- prompts the user to enter their first name
- then prompts the user to enter their last name
- prints a greeting similar to "Hello John Doe, pleased to meet you."

In-class activity 3

Follow along as the instructor steps through a slightly more complex script and ways that you can debug a script.

Variables

Variables in a Bash script are used to hold or contain values. Let us carefully distinguish between the *name* of a variable and its *value*. If **variable1** is the name of a variable, then **$variable1** is a reference to its *value*, the data item it contains.

Example of variables

```
joe@joe-VirtualBox:~$ my_name=Joe
joe@joe-VirtualBox:~$ echo my_name
my_name
joe@joe-VirtualBox:~$ echo $my_name
Joe
joe@joe-VirtualBox:~$ printf "%s\n" my_name
my_name
joe@joe-VirtualBox:~$ printf "%s\n" $my_name
Joe
```

http://tldp.org/LDP/abs/html/varsubn.html

Flow Control Structures

if .. elif .. else

```
if [ condition ]; then
  commands
elif [ condition ]; then
  commands
else
  commands
fi
```

Example of if .. elif .. else structure

```
read -p "Please enter a number between 1 and 100: " number
if [ $number -lt 0 ]; then
  printf "The number you entered is too low.\n"
elif [ $number -gt 100 ]; then
  printf "The number you entered is too high.\n"
else
  printf "You entered a good number: %d\n" $number
fi
```

http://tldp.org/LDP/abs/html/testconstructs.html

case statements

```
case $variable in
  "value1")
    commands
    ;;
  "value2")
    commands
    ;;
  "value3")
    commands
    ;;
  *)
    commands
    ;;
esac
```

Example of a case statement

```
read -p "Please enter the kind of pie you want: " pie_choice
case $pie_choice in
  "Apple")
    printf "Your apple pie is coming right up.\n"
    ;;
  "Peach")
    printf "Your peach pie is coming right up.\n"
    ;;
  "Cherry")
    printf "Your cherry pie is coming right up.\n"
    ;;
  *)
    printf "Sorry, we don't have %s, how about chocolate?\n" $pie_choice
    ;;
esac
```

http://tldp.org/LDP/abs/html/testbranch.html

for .. do loops

```
for variable in $somelist
do
  commands
done
```

Example for .. do loop

```
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
  if [ "$planet" = "Pluto" ];
  then
    printf "Sorry, Pluto is no longer a planet.\n"
  else
    printf "%s is a planet.\n" $planet
  fi
done
```

Another type of for .. do loop

For the example below LCV means loop control variable

```
for (( initialize LCV; test LCV; update LCV ))
do
  commands
done
```

Another example of a for .. do loop

```
j=0
for (( i=0; i<10; i++ ))
do
  j += i
  printf "%d\n" $j
done
```

while .. do

```
while [ evaluation ]
do
  commands
done
```

Example while .. do loop

```
user_input="n"
while [ "$user_input" = "n" ]
do
  read -p "Are you done yet? [y/n] " user_input
done
```

until ... do

```
until [ evaluation ]
do
  commands
done
```

Example until .. do loop

```
user_input="n"
until [ $user_input = "y" ]
do
  read -p "Are you done yet? [y/n] " user_input
done
```

http://tldp.org/LDP/abs/html/loops1.html

Functions

Like other programming languages, Bash has functions, though in a somewhat limited implementation. A function is a subroutine; a code block that implements a set of operations, a "black box" that performs a specified task. Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

Simple Function (no parameters)

```
function_name(){
  command...
}
```

Example of function without parameters

```
greeting(){
  read -n "Please enter your name: " name
  printf "Hello %s\n" $name
}
```

Function with parameters

```
function_name(){
  commands using $1, $2, $3 etc...
}
```

Example of function with parameters

```
print_fibonacci(){
  f1=0
  f2=2

  for (( i=0; i<=$1; i++ ))
  do
    printf "%d " $f1
    fn=$((f1+f2))
    f1=$f2
    f2=$fn
  done
}
```

http://tldp.org/LDP/abs/html/functions.html

Secure Shell (SSH)

Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network. The best known example application is for remote login to computer systems by users.

SSH provides a secure channel over an unsecured network in a client-server architecture, connecting an SSH client application with an SSH server. Common applications include remote command-line login and remote command execution, but any network service can be secured with SSH.

The most visible application of the protocol is for access to shell accounts on Unix-like operating systems

.

SSH was designed as a replacement for telnet and for unsecured remote shell protocols such as the Berkeley rlogin, rsh, and rexec protocols. Those protocols send information, notably passwords, in plaintext, rendering them susceptible to interception and disclosure using packet analysis. The encryption used by SSH is intended to provide confidentiality and integrity of data over an unsecured network, such as the Internet, although files leaked by Edward Snowden indicate that the National Security Agency can sometimes decrypt SSH, allowing them to read the contents of SSH sessions.

https://en.wikipedia.org/wiki/Secure_Shell

To install sshd type the following in a terminal window

```
sudo apt-get install ssh
Do you want to continue? [Y/n] y
```

Session 4 Homework

Install an SSH server on your local machine and try to login to it (from itself) using SSH. Use netstat to make sure port 22 is open.

References:

https://en.wikipedia.org
http://www.webopedia.com/quick_ref/computer-architecture-study-guide.html
http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf
http://www.tldp.org/LDP/abs/abs-guide.pdf