What is Linux and what is it doing

Although one often uses the term "Linux" to refer to the operating system, technically Linux refers to the operating system kernel. The kernel is what maintains coordination between hardware and software. Its duties include managing hardware, determining resource allocation for the different processes. The kernel typically runs in kernel space This is space in the memory specially allocated for the kernel to do its job. Most programs on our computers run in user space which is (more or less) the remaining space.

Let's be more precise about what some of the duties of the Linux kernel are:

- Driving hardware: The primary duty of the kernel is to provide an interface between software and hardware and control/manage the hardware accordingly. Let's walk through how this exactly works:
  1) Suppose a program wants to look at the mouse input. It will call `open("/dev/input/mouse0", O_RDONLY)`
  2) This is passed to the something called the virtual file system (VFS). The virtual file system is basically an interface between the kernel and the 'concrete file system' i.e. the virtual file system specifies the 'how' to read files, write to them and generally how to interact with them. <u>All</u> file handling is done through the VFS Of course one might see an issue here: a mouse click is not a file, so how is this being handled by VFS? The point is that we can simply define the VFS to handle this differently
  3) When the VFS sees that we are trying to read the mouse input, it knows that instead of looking at the hard drive (or wherever 'real' files are saved), it should call the mouse driver and pass along the request.

4) A device driver is basically what handles all interactions with a specific device. In this case, the mouse driver sees our request to access the mouse input and passes the data back to the kernel and is placed in a buffer *

5) The kernel will copy the data from the buffer to the user space where it can be accessed by the program

* The exact interactions between a kernel and a driver may vary from this.

Long story short is that the kernel provides a way for programs to interact with hardware through device files (these are the virtual files described above). There are 2 types of device files: block and character. Block device files have fixed sizes (i.e. blocks of data) such as with hard drives and USBs On the other hand, you have devices like keyboards and mice provide a continuous stream of data. These devices use character device files The main difference between the 2 types is that with block device files we can exploit the finiteness of the block to make data transfers more efficient e.g through caching and buffering With character device files data is coming in one character/byte at a time and needs to be processed accordingly.

Another important duty of the kernel is the unification of file systems A file system defines how data is stored and accessed on storage devices Typically data is stored in files which are themselves kept in directories, providing a nice hierarchical structure for the data. A file system will determine how to match these files and directories with the corresponding section of memory as well as manage things like access control and possibly even some data protection/backup capabilities. There are lots of different file systems; in principle each storage medium could have its own The kernel provides a way to access files from all of these drives and disks without having to worry about the exact details of the file system.

In Unix-like systems there is a single root at the top of the directory tree, represented by the '/' character. All other files and directories on the system are children (or grandchildren or great-grandchildren, etc.) of /. To access data from other storage devices (e.g. USBs) we must first mount the device. This means one of the directories in the tree now points at the additional storage device thus integrating it into our hierarchy.

Another significant duty of the kernel is process management. A process is an instance of a program. The kernel will create processes as well as track them and allocate resources for them. When a program is first run, the kernel will allocate a portion of memory for it, load the executable code into itself and start running it. It will also assign the process a PID (process identifier) which is how the kernel keeps track of all information corresponding to the process.

Most modern kernels are capable of multitasking, i.e. running multiple processes at once. In reality, only one process is being run per core but each process is given a small sliver of time (e.g. a few milliseconds) which gives the illusion that the processes are running all at once.

Finally the kernel is also responsible for rights and permissions management. For example read/write access to a file may be limited to some users so when open() or write() is called on a file, the kernel will check whether the user has access.

The Command Line

The command line is a (text-based) interface for uses to interact with their computer. The commands are sent through the terminal and executed by a command interpreter inside it (the shell) The default shell with Kali Linux is Bash (Bourne Again Shell). Using the command line is an important skill, especially as many Linux devices don't even have graphical interfaces

The most basic thing we may want to do on a computer is simply to navigate through its directories. We can do this through the command line by using the ls and cd commands. The ls <dir> command lists the files and directories in the given directory (if no argument given, it does so for the current directory). Then we can use cd <dir> to change the current directory to the given one. The parent directory is represented by .. (2 dots). If during navigation between directories we ever get 'lost', we can use the pwd command to print the working directory. The commands mkdir and rmdir allow us to make and remove (empty) directories respectively The mv <source> <target> command lets us move and rename files / directories cp <source> <target> is used to copy files. A lot of these commands have optional arguments that can be used for some slightly more sophisticated manipulation

A lot of these commands are in fact programs that the shell runs with the given arguments. This programs can be found in (one of) the directories in the PATH variable. Other commands, such as cd and pwd, are handled by the shell itself. PATH is an example of an environment variable

The File System

There is something called the Filesystem Hierarchy Standard (FHS). This standard defines some standard directories (and what goes in them) for Linux distributions. Some examples are:

- /bin/ stores basic programs (such as the commands above)
- /boot/ contains the Kali Linux kernel as well as other files required at bootup
- /dev/ is used for device files
- /etc/ is for configuration files
- /home/ is for the user's personal files
- /lib/ contains some basic libraries
- /sbin/ is for system programs (similar to /bin/)
- /tmp/ is for temporary data (wiped at boot time)
- /usr/ is for the user. It will also have /bin/, /sbin/ and /lib/ like the /root/ directory.
- /var/ holds variable data (i.e. data, files that are expected to change over time). A good example is log files
- /proc/ and /sys/ are 'virtual files' in the sense that they don't correspond to data on the disk but rather used by the Linux kernel to export data to user space
- /media/ is for mounting removable devices (CDs, USBs, etc.)
- /mnt/ is for temporary mount points
- /opt/ is for third-party applications
- /root/ has the admin's personal files
- /run/ is for temporary runtime data that is not kept across reboots
- /srv/ is used by servers being hosted on the system

Although there are no standards for user's home directories, there are some standards that people tend to stick to. For instance one can use the symbol ~ (tilde) to refer to the home directory (or whatever directory is stored in the HOME environment variable)

A common collection of files (traditionally) found in a user's home directory are application configuration files. Configuration files are used to change settings and modify application behaviour without changing its code. For example, you may wish to configure text editors to display .py and .txt files differently. The names of config files typically start with a dot . which makes them hidden files. The ls command does not display them unless the -a flag is used. These config files are also called dotfiles and we use so many of them now that a different convention is being proposed. The XDG Base Directory Specification recommends config files be placed in ~/.config, cache files in ~/.cache and application data in ~/.local.

Useful commands

The simplest thing we may want to do is simply read and edit files. We can use the cat <files> command (short for concatenate) to display the contents of file(s) on the terminal. The editor command can be used to start a text editor. Alternatively (for simpler files) the redirection operators > and >> can be used to save the output of a command to a file. For the former operator we overwrite the file while for the latter one we append to the file.

The find command can be used to find files by their names. Recall that * acts as a wildcard charcter. So for example, the command find /etc -names '*.txt' would look for all txt files in the /etc directory (including subdirectories). We can use grep to search the contents of files using regular expressions as well.

We can also use the terminal to check up on and manage processes. For example the ps aux command shows a list of currently running processes. We can use this to learn the PID of a process and send it (kill or otherwise) signals. We can also run processes in the background using the & operator and bg command. This means we can still use the terminal while the process is running. The jobs command can be used to get a list of background processes. We can use fg %job-number to bring any of these jobs to the foreground. A process in the foreground can also be paused using CTRL+Z and then restarted in the background using the bg command

The fin

Configuring Services

A service (also known as a daemon) are programs that run in the background to help the system. Let's start by looking at SSH.

SSH (Secure shell) is an industry standard tool and service to connect with machines remotely to transfer files, run commands, etc. As the name suggests, SSH is a secure way to connect with machines. The current version of the protocol is SSH-2, which technically consists of 3 distinct protocol layer. SSH-Trans (transport protocol), SSH-USERAUTH (authentication protocol and SSH-CONNECT (connection protocol). This is the order in which the protocols are run to maintain a secure connection

Setting up with SSH-TRANS

SSH-TRANS establishes a secure communication channel between client and server It does so through the following series of steps

1) S is open for connection requests

2) C sends a connection request to S

3) C and S will then exchange messages to decide on what key exchange algorithms to use, what encryption algorithms to use and so on

4) Then a key exchange is performed (based on the decisions made previously) which means the 2 parties securely decide on some shared secrets which may be used for encryption and so on.

5) The final step of the process is S authenticating itself to C. It will do so by sending its public key to C which C can verify against its own list of trusted keys. Of course this is not sufficient since in principle anyone could have sent the public key (it's public after all). Thus S will also send a signature which is a cryptographic message made using S's private key but can be verified using the public key. This verifies to C that S is indeed the server they requested (the next step is the authentication of C). In fact, the signature will need to be a bit more special before C can fully rest easy. In particular, the signature should depend on the current communication session. Otherwise an attacker could store signatures from previous sessions to authenticate itself as S (this is known as a replay attack). Moreover the signature should depend on the fact that C and S are communicating. This prevents an attacker from running a man-in-the-middle attack where an attacker has communication channels open with C and S simultaneously and carrying traffic between them while possibly altering them.

Once SSH-TRANS has been set up, SSH-USERAUTH is used to authenticate the user (e.g. through password checks) This is why it is important that the channel is secure since C may be sharing sensitive information when authenticating itself. Then the SSH-CONNECT protocol is used to manage and direct data streams to allow remote shell access, secure file transfers and SSH tunneling.

We should distinguish between the SSH client and the SSH server. The SSH client is the one that connects to a remote machine (the server) and typically has a human actively controlling it. The server is a machine that is listening for connection requests

The command to connect to an SSH server is ssh <user>@<host>. <user> must be an actual user at the remote server for this to work. <host> can be an IP address or a name although SSH does need a way for the name to be mapped to an IP address (e.g through a local file or a DNS server).

Most of the steps of the SSH-TRANS protocol are handled automatically by the machines but some of the settings can be configured through config files. As a client you can have system-wide configurations or user-specific ones. The SSH client OpenSSH gives precedence to user-level configurations over system ones Finally a client can have host-specific configuration as well.

A lot Linux distributions don't necessarily come with SSH-server application and one would need to be installed if needed Configuration for SSH servers is typically done at a system wide level and there is frequently no ability to have user-specific configurations There needs to be some basic compatibility between SSH clients and servers (e.g. both having some cryptographic algorithms in common)