Lab Assignment II

Varun Khadayate

NMIMS

Abstract

The varieties of operating systems (OS) have been emerged over the years due to the reason of having different features and functionalities. Understanding the functionalities of every OS guides users’ decisions about the OS to put in on their computers. Insight of this, a comparative analysis of various OS is required to supply details on the similarities and differences in recent styles of OS vis-à-vis their strengths and weaknesses. This paper focuses on the comparative analysis of Windows, Unix, Linux, Mac, Android, and iOS operating systems that supported the OS features and their strengths and weaknesses. An analysis of six different operating systems and results showed that Windows 10 had 0.04 malware files present while Windows 7 machine was 0.08. a better percentage of mobile malware targets Androids than iOS. Windows 10, Linux, UNIX, and Mac OS are more secured and reliable. Windows and Android are more popular, user-friendly, easy to use, and permit more application programs than Mac OS. Linux and Android are free while Windows is moderately costly, and Mac OS is exceptionally costly. Aside from Mac and iOS, others allow compatibility. Windows 10 and Mac OS integrated firewall. Windows and Android tend to be the foremost widely used especially the most recent versions. It's because they're affordable, secure, reliable, compatible, and user friendly. This study helps to supply some guides to both end-users and developers guiding them in taking decisions about operating systems that are best suited for them.

Keywords: Windows, Unix, Linux, Mac, Android, iOS.

**Lab Assignment II**

This OS Assignment mainly analyzes the basic differences between the permissions and models of Windows and UNIX file systems. A file system as the name suggests refers to the mechanism of storing and retrieving data on a computer disk. UNIX and Windows operating systems implement their file systems in different ways. This operating systems Assignment aims to discover the innate differences between these two operating systems in terms of file systems management. For the same purpose, this assignment compares the permissions of Windows and UNIX file systems to yield a basic idea about their working procedures. It is however evident that both the operating systems can conceptually handle the same set of files. However, the significant difference lies in the file permission structure, models, and management.

# Different Operating Systems.

## Windows OS.

Windows may be a personal computer’s OS from Microsoft corporation that, together comes with some commonly used applications. Windows has become a “standard” for common users in most organizations also as in most homes. Microsoft Windows is a series of operating systems and graphical user interfaces produced by Microsoft. Microsoft first introduced an operating environment named Windows as an add-on to MS-DOS in response to the growing interest in graphical user interfaces (GUIs). Windows has approximately 91% of the market share of the client operating systems. The most recent client version of Windows is Windows 10(version 20H2); the most recent server version is Windows Server 2019 NT 10.0.

## UNIX OS.

Unix may be a computer OS originally developed in 1969 by a gaggle of AT&T employees at Bell Labs. Today’s UNIX is split into several branches, developed over time by AT&T also as various commercial vendors and other organizations.

UNIX was created to supply a multiuser, multitasking system for users. the thing behind the planning of UNIX was to supply simple, yet powerful utilities that would be pieced together flexibly to perform a good sort of tasks. Unix may be a family of multi-user operating systems. Unix has a very strong security and model and comparatively simple design, making it popular and fairly easy to implement. Many operating systems are either supported or modeled after the primary Unix systems, like Linux, Solaris, or Mac OS X.

The UNIX OS comprises three parts: The kernel, the quality utility programs, and therefore the system configuration files.

# Features of Windows.

Microsoft was the primary to introduce the thought of an OS named Windows as an attachment to the MS-DOS back to the increasing curiosity in Graphical User Interfaces (GUIs). Microsoft Windows in the end started to monitor the world market of personal computers, going far ahead of Mac OS, which was the leader before its era.

Microsoft has designed and marketed the Windows operating system as a collection of several operating systems.

Windows is the most known OS to a layman. It is trustworthy from the user’s point of view.

First to introduce plug and play rule.

# Windows File System Permission Model.

In Windows operating system, the FAT32 (File Allocation Table 32) means that the file system stores data in chunks of 32 bits. The naming convention supported by the FAT32 file system requires an ASCII character set. The file name must start with a number or letter and does not support any special characters (. " / \ [ ] : ; | = ,). However, there are certain disadvantages in the FAT32 file system such as fragmentation, lack of storage efficiency for larger disk partitions, and lack of fault tolerance.

In the 1990s, Microsoft built a more reliable, secure, and high-performance file system called NTFS (New Technology File System). The common permissions in the NTFS file system are, read, modify, change owner and delete. Setting the permission requires specifying the level of access for groups and users. NTFS permissions are the same for local and network users and granted at the Windows logon (Perrin, 2011). The Windows administrators provide access control over the files, folders, containers, and objects using the NTFS utility available on the network. The ‘Security Descriptor’ is responsible for deciding what kind of access (read/ modify/ delete/ change owner) is allowed for individual users and groups of users.

The standard permissions in NTFS include ‘full control’, ‘modify’, ‘read & execute’, ‘read’ and ‘write’. ‘Deny’ can also be used to deny access to a folder or deny full control permission. However, in the latest versions of NTFS, Microsoft has included certain advanced permissions that include the following:

**Traverse Folder/ Execute file:** Allows navigating through folders to reach files even if there is no permission to access the traversed files.

**List folder/ read data:** Allows viewing a list of contents and data files in a folder.

**Read attributes:** Allows viewing read-only and hidden attributes of a file or folder.

**Read extended attributes:** Allows viewing extended attributes

**Create files/ Write data:** Allows creation of files within a folder, making changes, and overwriting on existing content.

**Create folders/ Append data:** This permission allows creating folders within folders, make changes to the end of the file. However, it does not allow overwriting or deleting data in a file.

Apart from these, the other advanced permissions are written attributes, write extended, delete, change permissions, read permissions, and take ownership (Morgan, 2017).

# Features of UNIX.

The UNIX Operating System is available on systems with a large range of computing power, from microcomputers to mainframes, and on different manufacturers ’ machines. No other operating system can make this possible. We see the reasons for the popularity and success of UNIX.

The reasons are the following:

**Portability:** The system is designed in high-level language making it easier to read, understand, change, and therefore move to other machines. The code can be changed and complied with on a new machine.

**Machine-independence:** The System hides the machine architecture from the user, making it easier to write applications that can run on micros and mainframes.

**Multi-User Operations:** UNIX is a multi-user system designed to support a group of users simultaneously. The system allows for the sharing of processing power and peripheral resources, white at the same time providing excellent security features.

**Hierarchical File System:** UNIX uses a hierarchical file structure to store information. This structure has flexible in grouping information in a way that reflects its natural state. It allows for easy maintenance and efficient implementation.

**UNIX shell:** UNIX has a simple user interface called the shell that has the power to provide the services that the user wants. It protects the user from having to know the hardware details.

**Utilities:** UNIX has over 200 utility programs for various functions. New utilities can be built effortlessly by combining existing utilities.

# UNIX File System Permission Model.

UNIX file system deals with three core aspects such as user, groups, and ownership. UNIX operating system has files and directories in a tree structure where the basic permissions are, read, write, and execute. UNIX OS uses nine bits for specifying the access permission for a file, where each bit is marked by r (read), w (write), and x (execute). The UNIX file permissions depend on a simplified and abbreviated form of ACL (Access Control List) (Reynolds, 2014). The DAC (Discretionary Access Control) enables users to configure the level of access for the resources that they 'own'.

|  |  |  |
| --- | --- | --- |
| Symbolic Notation | Octal Notation | English |
| ———- | 0000 | no permissions |
| —x–x–x | 0111 | execute |
| –w–w–w- | 0222 | write |
| –wx-wx-wx | 0333 | write & execute |
| -r–r–r– | 0444 | read |
| -r-xr-xr-x | 0555 | read & execute |
| -rw-rw-rw- | 0666 | read & write |
| -rwxrwxrwx | 0777 | read, write, & execute |

In the 9-bit permission model, the permissions of files can be changed using the command ‘chmod’. Apart from that, the ‘setuid’ allows running executables with the permission of a file owner. The ‘setgid’ command is used as an equivalent of ‘setuid’ property of groups. The ‘Sticky bit’, when assigned to files and directories, prevents users from deleting (Doyle, 2011). For instance, ‘/tmp’ folder allows any user to store files. However, only the file owner has the access rights to delete or modify the file stored in ‘/tmp’. The ‘umask’ command defines the permissions to be masked when a file is created.

One example of the major limitation of the 9-bit permission model can be the command ‘setuid-root'. This command allows ordinary users to perform administrative tasks. Moreover, the 9-bit permission model used by UNIX operating system has no fine-grained access control measures for non-class users.

# UNIX VS. WINDOWS

**Stability:** UNIX systems are more stable than windows. There are many factors to support this fact but to name just big ones: UNIX handles high server loads better than Windows and UNIX machines seldom require reboots while Windows is frequently needed them. Servers running on UNIX enjoy extremely high up-time and high reliability.

**Performance:** While there is some doubt about which operating system performs better, in our view both perform comparably in low-stress conditions. However, UNIX servers under high load are strong than Windows.

**Compatibility:** Web sites designed and programmed to be host under a UNIX-based web server can easily be hosted on a Windows server, whereas the reverse is not always true. This makes the programming for UNIX sites the better choice.

**Security:** Unix has greater built-in security and permissions features than Windows. Until about the year 2000, i.e., before the launch of Microsoft Windows XP, Windows was considerably weaker in security as compared to Unix systems but today, it is really difficult to compare the two on this basis. Windows XP service pack 2 introduced various tools in areas such as emails, web browsing, memory, networking, and even the kernel of the OS. The newer versions of Microsoft Windows that have recently been launched such as Windows Vista and Windows Seven, lay too much emphasis on security. They are far much secure than their elder versions. They have built-in feasibility for permissions similar to UNIX which prohibits malicious programs from taking over the control of the system resources without user permission.

## Comparison

The initial design of UNIX software by AT&T included space at a premium and each file had three sets of permissions just like the access permissions for ‘everyone’, ‘the owning group’, and also the ‘owning user’. The three varieties of permissions are typically read, write, and execute. The execute permission allows the user to run a binary program or a script.

There is a challenge during this scenario. If a user has to write access to a specific file, then it's essential to make a replacement group allowing both the first and second users. This new group must replace ‘the owning group’. Hence, there is a high chance of these permissions becoming messy in large organizations.

However, within the Windows classification system, the file permissions model goes to the acute by defining permissions to every file for any number of users, groups, or everyone. Moreover, the category for ‘everyone’ encompasses system users, guests, authenticated users, network users so on. These fine-grained permissions are going to be set for inheritance by subfolders, which enforces any change within the parent folder to duplicate automatically to the subfolders.

On the alternative hand, UNIX enforces application-level and network-level permissions, during which certain applications are allowed to access specific folders (McDermott, 2017). However, whether or not this extra feature adds adequate flexibility in software package security, the duty of debugging permission-based errors is significantly difficult. It requires many days of fine-tuning so on implement a UNIX server and acquire it ready for production, because of the fine-grained application-level permissions.

Therefore, Windows OS offers fine-grained user-level control whereas UNIX involves application-level control. It's often said that Windows organization permissions are ideal for file storage and management. On the alternative hand, UNIX offers an efficient permission model for web and application servers.

The octal 777 permission in UNIX is analogous to ‘full control’ permission within the Windows organization. UNIX allows setting permission through the graphical shell or using ‘chmod’ command. Many modern UNIX distributors support ACLs (Access Control Lists) based security for directories and files. However, the sysadmins mostly use the UNIX owner-group-world permission model (Rafacz, 2011). So on delegate authority, UNIX users often use the ‘su’ and ‘sudo’ for administrator privileges. On the other hand, the Windows UAC (User Access Control) is compared with ‘sudo’ in UNIX. Windows users use ACLSs at the file-level while UNIX users depend upon the outdated model of the owner-group-world permission model.

# Result and Conclusion

In this case study of which is healthier Windows or Unix. From all the studies it was concluded that from the purpose of view of a lay user windows remains better for its easy access.

For the soundness, the UNIX is the best among all.

I conclude that windows are best for private use and the new user because it's widely used and simple to use. Moreover, drivers and support for the windows are basically and simply available. It also supports GUI which helps the user to interact easily with the OS.

Since there's no mention of the use of LINUX OS but if it was mentioned then for commercial use LINUX is best to use. Because Linux is incredibly stable and secure than windows ever. So for the massive industries where information is extremely important thing Linux is the most secure OS to be used. It's true that UNIX is more stable than Linux and also compatible, but Unix isn't used nowadays. Therefore Linux continues to be better and secure to use for commercial use.

So to conclude It is seen during this software Assignment that UNIX operating system has three specific permissions that will be applied to every class (user, group, other), which are, read, write, and execute. it's possible to vary permission behavior in UNIX with the setuid, setgid, and sticky bits. The sticky mode or text mode encourages the kernel to retain the image of an object beyond termination. In Windows, if an object or a file incorporates a null ACL, then it's no restriction. On the opposite hand, if the file has an empty ACL then nobody can access the file.

# Shell OS

Your interface to the operating system is called a shell.

The shell is the outermost layer of the operating system. Shells incorporate a programming language to control processes and files, as well as to start and control other programs. The shell manages the interaction between you and the operating system by prompting you for input, interpreting that input for the operating system, and then handling any resulting output from the operating system.

Shells provide a way for you to communicate with the operating system. This communication is carried out either interactively (input from the keyboard is acted upon immediately) or as a shell script. A shell script is a sequence of shell and operating system commands that is stored in a file.

When you log in to the system, the system locates the name of a shell program to execute. After it is executed, the shell displays a command prompt. This prompt is usually a $ (dollar sign). When you type a command at the prompt and press the Enter key, the shell evaluates the command and attempts to carry it out. Depending on your command instructions, the shell writes the command output to the screen or redirects the output. It then returns the command prompt and waits for you to type another command.

A command line is the line on which you type. It contains the shell prompt. The basic format for each line is as follows:

|  |
| --- |
| $ Command Argument(s) |

The shell considers the first word of a command line (up to the first blank space) as the command and all subsequent words as arguments.

The shell also provides a user environment that can be customized using initialization files. These files contain settings for user environment characteristics, such as:

* Search paths for finding commands.
* Default permissions on new files.
* Values for variables that other programs use.
* Values that can be customized.

There are 4 different types of shell:

## The Bourne Shell(sh)

It is the preferred shell for shell programming because of its compactness and speed. It’s drawback is that it lacks features for interactive use, such as the ability to recall previous commands (history). The Bourne shell is also lacking in built-in arithmetic and logical expression handling. For the Bourne shell the:

* Command full-path name:- **/bin/sh** and **/sbin/sh**.
* Non-root user default prompt:- **$**.
* Root user default prompt:- **#**.

## The C-Shell(csh)

It has incorporated features for interactive use, such as aliases and command history and also includes convenient programming features, such as built-in arithmetic and a C-like expression syntax. For the C shell the:

* Command full-path name:- **/bin/csh**.
* Non-root user default prompt:- hostname **%**.
* Root user default prompt:- hostname **#**.

## The Korn Shell(ksh)

It is a superset of the Bourne shell so, it supports everything that is present in the Bourne shell. It’s interactive features are comparable to those in the C shell and convenient programming features like built-in arithmetic and C-like arrays, functions, and string-manipulation facilities are also included. Being faster than the C-Shell, it runs the scripts written for the Bourne Shell. For the Korn shell the:

* Command full-path name:- **/bin/ksh**.
* Non-root user default prompt:- **$**.
* Root user default prompt:- **#**.

## The GNU Bourne-Again Shell(bash)

It is compatible to the features of Bourne Shell and incorporates features of C and Korn Shell. For the GNU Bourne-Again shell the:

* Command full-path name:- **/bin/bash**.
* Default prompt for a non-root user:- **bash-x.xx$**. (x.xx here is the shell version number)
* Root user default prompt:- **bash-x.xx#**.

# Filters

A filter is a small and specialized program in UNIX  that transforms plain text (i.e., human readable) data in some meaningful way and that can be used together with other filters and pipes to form a series of operations that produces highly specific results.

Few Filters are:

## Cat

Displaying file line by line.

**Syntax**: cat [path]

## Head

Displays n lines from the start of the file. The default value for n is 10.

**Syntax**: head [-number\_of\_lines\_to\_print] [path]

## Tail

Reverse of head, displays line from the last of the file.

**Syntax**: tail [-number\_of\_lines\_to\_print] [path]

## Sort

Sorts the content of the line alphabetically in multiple orders but ascending by default.

**Syntax**: sort [-options] [path]

## Uniq

Removes continuous duplicate lines.

**Syntax**: uniq [options] [path]

## Wc

Displays no.of lines, words and characters in the file.

**Syntax**: wc [-options] [path]

## Grep

Search a particular information from a text file.

**Syntax**: grep [options] pattern [path]

## Tac

Reverse of cat i.e., displays file line by line but in reverse order.

**Syntax**: tac [path]

## Sed

Used to apply search and replace operation in the file.

**Syntax**: sed [path]

## Nl

Numbers the line of the files.

**Syntax**: nl [-options] [path]

# Shell Programming

The programming language for the shell is complex enough to handle many kinds of things that might otherwise be done in a language such as C, but in contrast is interpretive rather than compiled. The basic virtues are that

|  |  |
| --- | --- |
| * it can be used for encapsulating groups of shell commands that are to be executed. * it is often easier to accomplish a systems function in the shell using the shell programming capability than it is to write a high-level language program for the same purpose. |  |

Shell scripts are often used as a simpler alternative in form of small code compared to any hard written code performing the same function. For this circumstance, it is useful to be able to pass in arguments to the shell script. These are referenced inside the script as the positional parameters **$1, $2, $3, . . . .** For example, if we want to capture the ***sed*** procedure for indenting a file in this manner; i.e., the shell script ***tabin*** might simply consist of

|  |
| --- |
| *#!/bin/bash sed '/./s/^/\t>/' $1* |

Similar to other programming languages, the power of a shell programming language lies in the ability to use decisive logic for defining a shell script's behavior. The constructions used for this purpose are comparable to the standard structure of languages like C, C++, etc. They include loops "*if-then-else*", "*case*", "*for*", "*while*" and "*until*".

## if-then, if-then-else, if-then-elif-then-else:

*Syntax:*

if <*condition*>  
  then  
  <*if-part*>  
elif <*condition*>  
  then  
  <*elif-part*>  
else  
  <*else-part*>  
fi

elif can be omitted in the *if-then-else* construction whereas, else and elif both can be omitted for the *if-then* construction.

## case:

*Syntax:*

case <*test-value*> in  
  <*case1-val*>)  
    <*commands*>  
    ;;  
  <*case2-val*>)  
    <*commands*>  
    ;;  
  . . .  
  \*)      ««- *case that catches anything else*  
    <*commands*>  
    ;;  
esac

The identifier for each case must be on its own line and each case terminates when a double ";" is encountered.

## for loop:

*Syntax:*

for <*item*> in <*list*>  
  do  
    <*loop-body*>  
  done

## while loop:

*Syntax:*

while <*condition*>  
    do  
     <*loop-body*>  
   done

this type of construction is same as *if-then* construction.

# Programming with standard I/O

The Standard I/O system provides a wide variety of functions to perform input/output and associated tasks.

A file can be accessed by using Standard I/O in the following steps:

1. Opening a file using the function *fopen* or *afopen*. The name of the file to open is passed as an argument to the function. Both the functions return a pointer to an object of type **FILE.** The data addressed by this **FILE** pointer are used to control all further program access to the file.
2. Transfer data to and from the file using appropriate functions. The **FILE** pointer returned by *fopen* is passed to the other functions to identify the file to be processed.
3. Close the file. After the closing, all changes have been written to the file and the **FILE** pointer is no longer of any use. All the files that are not closed manually in the code are automatically closed by the compiler once the program terminates.

There are three standard files that are opened before the program executes which are accessible with the **FILE** pointers namely: *stdin*, *stdout*, and *stderr*. These identify the standard input stream, standard output stream, and standard error stream, respectively. For programs running under the USS shell, these **FILE** objects reference the standard files for the program that invoked them.

Standard I/O functions may be grouped into several categories:

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| Control Functions | control basic access to files |
| **fopen** | opens a file |
| **afopen** | opens a file with system-dependent options |
| **freopen** | reopens a file |
| **afreopen** | reopens a file with system-dependent options |
| **tmpfile** | creates and opens a temporary file |
| **tmpnam** | generates a unique filename |
| **fflush** | writes any buffered output data |
| **afflush** | forces any buffered output data to be written immediately |
| **fclose** | closes a file |
| **setbuf** | changes stream buffering |
| **setvbuf** | changes stream buffering |
|  |  |
| Character I/O Functions | read or write single characters |
| **fgetc** | reads a character |
| **getc** | reads a character (macro version) |
| **ungetc** | pushes back a previously read character |
| **getchar** | reads a character from **stdin** |
| **fputc** | writes a character |
| **putc** | writes a character (macro version) |
| **putchar** | writes a character to **stdout** |
| String I/O Functions | read or write character strings |
| **fgets** | reads a line into a string |
| **gets** | reads a line from **stdin** into a string |
| **fputs** | writes a string |
| **puts** | writes a line to **stdout** |
| Array I/O Functions | read or write arrays or objects of any data type |
| **fread** | reads on or more data elements |
| **fwrite** | writes one or more data elements |
| Record I/O Functions | read or write entire funtions |
| **afread** | reads a record |
| **afread0** | reads a record (possibly length 0) |
| **afreadh** | reads the initial part of a record |
| **afwrite** | writes a record |
| **afwrite0** | writes a record (possibly length 0) |
| **afwriteh** | writes the initial part of a record |
| Formatted I/O Functions | easily read or write formatted data |
| **fprintf** | writes one or more formatted items |
| **printf** | writes one or more formatted items to **stdout** |
| **sprintf** | formats items into a string |
| **snprintf** | formats items into a string (with maximum length) |
| **fscanf** | reads one or more formatted items |
| **scanf** | reads one or more formatted items from **stdin** |
| **sscanf** | obtains formatted data from a string |
| **vfprintf** | writes formatted data to a file |
| **vprintf** | writes formatted data to a standard output string |
| **vsprintf** | writes formatted data to a string |
| **vsnprintf** | writes formatted data to a string (with maximum length) |
| File Positioning Functions | interrogate and change the file position |
| **fseek** | positions a file |
| **fsetpos** | positions a file |
| **rewind** | positions a file to the first byte |
| **ftell** | returns current file position for **fseek** |
| **fgetpos** | returns current file position for **fsetpos** |
| Keyed Access Functions | read, write and position a keyed stream |
| **kdelete** | delete a record from a keyed file |
| **kgetpos** | return position of current keyed file record |
| **kinsert** | add a new record to a keyed file |
| **kreplace** | replace a new record in a keyed file |
| **kretrv** | retrieve a record from a keyed file |
| **ksearch** | search for record in a keyed file |
| **kseek** | reposition a keyed file |
| **ktell** | return RBA of current record of keyed file |
| Error-Handling Functions | test for and continue execution after I/O errors and other I/O conditions |
| **feof** | tests for end of file |
| **ferror** | tests for error |
| **clearerr** | resets previous error condition |
| **clrerr** | resets previous error condition |
| File Inquiry Functions | obtain information about an open file at execution time |
| **fattr** | returns file attributes |
| **fileno** | returns file number |
| **ffixed** | tests whether a file has fixed length records |
| **fnm** | returns the name of a file |
| **fterm** | tests whether a file is the user's terminal |

These functions needs stdio.h file to be include in the header using #include to work.

# Unix System Calls

System calls in Unix are used for file system control, process control, interprocess communication etc. Access to the Unix kernel is only available through these system calls. They are similar to function calls, the only difference is that they remove the control from the user process.

There are five types of system calls namely:

* 1. Process Control
  2. File Management
  3. Device Management
  4. Information Maintenance
  5. Communication

Few examples for each of them are:-

|  |  |
| --- | --- |
| Process Control | fork() exit() wait() |
| File Manipulation | open() read() write() close() |
| Device Manipulation | ioctl() read() write() |
| Information Maintenance | getpid() alarm() sleep() |
| Communication | pipe() shmget() mmap() |
| Protection | chmod() umask() chown() |

# Bibliography

1. McDermott, J. (2017). Introduction to UNIX-family File Permissions | Learning Tree Blog. [online] Learning Tree Blog. Available at: https://blog.learningtree.com/introduction-unix-family-file-permissions/ [Accessed 12 Sep. 2018].

In this article, the author has discussed the authorization specifics in the UNIX file permission model. McDermott (2017) has reflected on the basic permission types (read, write, and execute) supported by the UNIX operating system along with a brief overview of changing permissions with the chmod command. The paper also discussed some common file modes in UNIX.

2. Doyle, M. (2011). Understanding File Permissions. [online] Elated.com. Available at: https://www.elated.com/articles/understanding-permissions/ [Accessed 12 Sep. 2018].

In this article, Doyle (2011) has discussed the basic purpose and importance of file system permissions. In this context, the author has focused on defining permissions from the UNIX operating system perspective. The paper explains the 9-bit permission model in UNIX that defines permission for each class including the users (u), groups (g), and other (o). The author also discussed setting permissions with FTP and SSH.

3. Mfillpot. (2010). Understanding Linux File Permissions. [online] Linux.com | The source for Linux information. Available at: https://www.linux.com/learn/understanding-linux-file-permissions [Accessed 12 Sep. 2018].

In this article, the author has researched the Linux file permissions along with the three basic permission groups and permission types. The author further defines the ways to view, understand, and modify the permissions. The paper also discusses the methods for explicitly defining permissions as well as using binary references to set permissions.

4. White, K. (2012). Understanding File System Permissions | Mac OS X Support Essentials: File Systems | Peachpit. [online] Peachpit.com. Available at: http://www.peachpit.com/articles/article.aspx?p=1403238&seqNum=5 [Accessed 12 Sep. 2018].

In this article, White (2012) discusses the ownerships for permissions along with the traditional UNIX permissions. In addition to that, the research also encompasses the ACLs (Access Control Lists), permission to hierarchical context, permissions for sharing, home folder sharing, and securing new files or objects.

5. Perrin, C. (2011). Understand the setuid and setgid permissions to improve security. [online] TechRepublic. Available at: https://www.techrepublic.com/blog/it-security/understand-the-setuid-and-setgid-permissions-to-improve-security/ [Accessed 12 Sep. 2018].

In this article, Perrin (2011) discussed how the basic UNIX permissions in the operating system file management work and why it is important for security reasons. More precisely, the significance of the two commands such as setuid and setgid are discussed in terms of enhancing the security of UNIX file systems. The author also compared and contrasted between good and bad security in this context.

6. Morgan, M. (2017). Windows Share Permissions Best Practices (Server 2016) | Global Knowledge Blog. [online] Global Knowledge Blog. Available at: https://www.globalknowledge.com/blog/2017/09/19/best-practices-for-share-permissions-in-windows-server-2016/ [Accessed 12 Sep. 2018].

In this article, Morgan (2017) has discussed the best practices for share permissions in Windows Server 2016. For this purpose, the author has also discussed certain limitations and drawbacks about the share permissions in Windows. The paper also considers how NTFS solves the identified issues.

7. Reynolds, R. (2014). Managing Permissions on Windows with Access Control Lists. [online] Puppet. Available at: https://puppet.com/blog/managing-permissions-on-windows-access-control-lists [Accessed 12 Sep. 2018].

In this article, the author Rob (2014) has reflected on the role of ACLs along with their most fundamental features and functions. The author has discussed the ACL type structure and permission properties along with examples, granular permission, and website setup with ACL.

8. Rafacz, R. (2011). Linux File Permissions. [online] Pluralsight.com. Available at: https://www.pluralsight.com/blog/it-ops/linux-file-permissions [Accessed 12 Sep. 2018].