Users and Permissions: Section 1 Transcript

Introduction

1/3

Welcome to the Users and Permissions module. The UNIX security model is based on the concept of superuser and permissions. In this model, one account, the root, has access to everything. All other users have access to files using a form of discretionary access control. Additionally, files are assigned a set of permissions such as "read," "write," and "execute" and the access to these files is enforced around the idea of which user and group owns the file.

In this module, we will discuss the concept of user accounts on a UNIX system. We will also discuss the concept of file permissions and how they relate to user accounts. We will continue by discussing special permissions, and we will round out our discussion by comparing the traditional security model of UNIX with other forms of access control.

Throughout this module, you'll be presented with opportunities to assess your learning with Knowledge Checks and then to apply what you've learned by completing Exercises

At the conclusion of this module, you will be able to:

- Discuss UNIX accounts and how they are represented on a UNIX system,
- Recognize UNIX files and directory ownership attributes, and
- Use tools to view and change UNIX file ownership attributes,
- Additionally, you will be able to determine how permissions affect access to files and directory objects, and
- Use tools to view and change UNIX files and directory permissions,
- Finally, you will be able to explain how special permissions change the inherited effective ownership identities of child processes.

Prerequisites

2/3

As you are traversing through this module, you are also expected to use Internet search engines and UNIX `man` pages to discover and learn about a variety of UNIX commands. Some of the commands you will need to research and become proficient with are: `chgrp`, `chmod`, `chown`, `getent`, `getfacl`, `groupadd`, `groupdel`, `groupmod`, `groups`, `id`, `ls`, `newgrp`, `passwd`, `userdel`, `setfacl`, `su`, `sudo`, `useradd`, `umask`, and `usermod`. Other commands that are specific to some UNIX variants you will need to research include `listusers`, `ppriv`, `chage`, `gpasswd`, `getcap`, and `setcap`.

Before continuing on, take the time to research the UNIX commands displayed - especially any that may be unfamiliar to you. You may print a copy of the commands (see Supplemental Materials under Resources on the toolbar) and/or use the "Notes" option on the toolbar to record data you learn during your research of each command.

Select Forward when you are ready to continue.

Bypass Exam Introduction

3/3

If you are already familiar with the subject matter presented in this module, you can choose to take a Bypass Exam to skip this module.

The Bypass Exam option provides a single opportunity to successfully demonstrate your competence with the material presented within the module. If you pass, you'll receive credit for completing the module, unlocking the content within, and you will be free to proceed to the next module. If you do not pass, you will need to successfully complete the module, including all exercises and the Module Exam, to receive credit.

Click the Next Section button to continue.

Users and Permissions: Section 2 Transcript

User Accounts

1/10

Unlike user account management in Windows Operating Systems, UNIX user account management is simplistic. UNIX uses three files to define a user account on the system: `/etc/passwd`, `/etc/shadow`, and `/etc/group`. Additionally, all the commands that manipulate or retrieve information about users and groups generally rely on at least one of these three files. Let's take a moment to discuss the content of each of these files, beginning with the `/etc/passwd` file.

User Accounts: /etc/passwd

2/10

The `/etc/passwd` file, also known as user database file or passwd file, is composed of lines that each represents an individual user account. Each line is composed of a variety of fields separated by colons. Click each icon displayed to learn more about each specific field. When you have finished exploring, click the Next Slide button in the navigation bar to continue with the module.

Username Field: The username field, also referred to as the login name, is comprised of the name of the user on the system. Since the username is used to uniquely identify a user on the system, this field will be different for each user.

Password Field: Historically, the password field has contained the encrypted password for the user. However, it now contains an "x" and the encrypted password is now stored in a different file. This change was motivated by the fact that the `/etc/passwd` file is world-readable, which means it is readable by any user on the system. When the password was within this field, any user, armed with good hardware, could potentially brute force passwords of other users on the system.

UID Field: The User Identification (UID) field stores a unique number that the system, and other applications, can use to identify the user and determine their access privileges. This field must also be different for each user as it uniquely identifies a user. An exception to this rule is the UID of zero (0). Any user with the UID 0 has superuser privileges on the system. We will discuss privileges later in this module.

GID field: The Group Identification (GID) field is a numerical value that represents the primary group of the user. This field plays an important role in determining user access privileges. While the GID represents the user's primary group, it is important to note that a user can belong to multiple groups, which is represented in the `/etc/group` file.

GECOS Field: The General Electric Comprehensive Operating System (GECOS) Field is often misrepresented. Many consider it simply a field in which the user's full name is stored. In actuality, this field can also be used to store other identifying information about the user, such as their phone number, address, and more. The important thing to remember is that all the entries should be separated by semi-colons. Since this field is optional, it can be left blank.

Home Directory Field: The Home Directory Field is the default directory in which the user is placed after login. The home directory holds session customization scripts, also known as startup scripts, and other personal user files. It is usually a subdirectory within the `/home` directory in most variants of UNIX. However, in Solaris it is a subdirectory within the `/export/home` directory.

Shell Field: The shell field represents the default shell launched by the system after the user logs in. This shell will be used by the user to interact with the system. On systems with a Graphical User Interface (GUI), this is also the shell that is launched by the terminal program in a desktop environment. For user accounts, such as services accounts that are not authorized to log in but are used internally, this field is usually empty or set to a program that displays a message when a user attempts to log in with any of these services accounts.

User Accounts: /etc/shadow

Unlike the passwd file, the `/etc/shadow` file, or shadow file, stores the encrypted password information for user accounts and is only accessible to the root user. Just like in the passwd file, each line in the shadow file provides information about a single user and is composed of fields separated by colons. Entries include mandatory items such as the username, or login name, as well as the encrypted password. Note that a blank encrypted password field indicates that a password is not required to log in IF you have access to the console or machine itself. If, however, you are attempting to ssh to the device as a user whose encrypted password field is empty, you will not be able to authenticate to the machine unless specific atypical and ill-advised sshd configurations have been implemented; a star (*) indicates that the account is disabled; and two bangs, or exclamation points, indicates that the account is locked. The shadow file also stores optional password aging information, such as the:

- Number of days from January 1, 1970 that have passed since the password was last changed. (Note that January 1, 1970 is considered to be the starting point for keeping time on UNIX systems. This starting point is commonly referred to as UNIX time, POSIX time, and/or Epoch time.)
- Number of days before the password may be changed again
- Number of days after which a password must be changed
- Number of days the user is warned before a password expires
- Number of days after a password expires before the account is disabled
- Number of days from January 1, 1970 that have passed since the account was disabled, and
- Reserved field

In modern versions of UNIX, most of the optional fields are usually blank. This is generally because another configuration file has been introduced to support these fields and other more advanced options. This being said, these configuration file are different for each variant of UNIX. For example, on Solaris the `/etc/security/policy.conf` file handles these options. However, on BSD and Linux systems the configuration files that handle these options are `/etc/login.conf` and `/etc/login.defs`, respectively.

Encrypted Password: /etc/shadow

The encrypted password in the shadow file usually consists of three parts separated by the dollar ($)

character. These parts include the:

- Algorithm-id - which has a value that specifies the type of hash algorithm used to encrypt the password, such as: 1 (MD5), 5 (SHA-256), or 6 (SHA-512); note that whenever the encrypted password section appears alone in the shadow file, the encryption algorithm used is the Data Encryption Standard, or DES
- Salt - which is a random string formed with alphanumeric characters that are used in the encryption algorithm to make sure the encrypted password is different every time, and the
- Actual encrypted password

Any user can set or change their own password by running the `passwd` command without options; however, only the root user can change any other user's password on the system. The default encryption algorithm is pulled from `/etc/security/policy.conf`, `/etc/login.defs` or `/etc/login.conf` in Solaris, Linux or BSD, respectively.

Since you now have basic familiarity with `/etc/passwd` and `/etc/shadow` files, let's examine another UNIX file that defines a user account on the system - the `/etc/group` file.

User Accounts: /etc/group

5/10

The `/etc/group` file contains the list of groups in the system. Like the `/etc/passwd` file, the `/etc/group` file is world-readable so that applications can verify associations between users and groups. Each line in this file represents a group and is composed of four standard fields separated by colons. Fields include the:

- Group Name
- Group Password: An optional field that when set allows users who are not members of the group to temporarily join the group after running the `newgrp` command. Those users that temporarily joined the group may leave the group by running the `exit` command.
- GID: Again, a numerical value that uniquely identifies the group
- Group Members: A comma-separated list of users that are part of this group. User names are not listed in their primary groups because it would be redundant to list them in this section and to specify that this is their primary group in the passwd file. Therefore, only users that have this group as a secondary group will be listed here.

Authentication Databases

6/10

As we discussed, the `/etc/passwd`, `/etc/shadow`, and `/etc/group` files are used for system authentication and authorization. Authentication typically requires a form of identification and credentials for access to be granted. In most cases, that is done through a username and password. The credentials provided by a user are then compared to the information stored in a database. Typically, by default, the database that is used on UNIX systems is the "files" database; which consists of `/etc/passwd`, `/etc/group`, and `/etc/shadow` files. However, there are other types of databases, such as network-based databases, that can also assist with authentication and authorization.

Some of the commonly used network authentication databases are LDAP (Lightweight Directory Access Protocol), NIS (Network Information Service), Winbind (Windows Berkeley Internet Name

Domain Server), and IPA (Identity, Policy, and Audit). LDAP is one of the more commonly used services to provide network based authentication in UNIX environments. Throughout this section, we will discuss LDAP as it relates to network-based authentication.

Network based authentication servers provide a centralized location to manage users, groups, and passwords. There are many benefits to using centralized network authentication. For example, when used in conjunction with a file sharing server, centralized network authentication can provide single sign on (SSO) with roaming profiles. This allows network users to sign into different systems throughout the network without having to create a system account on each one. Roaming profiles enable user home directories to automatically mount on any system they log into.

Pluggable Authentication Module and Name Service Switch

While there are many different authentication methods, most applications are configured to authenticate via the Pluggable Authentication Module (PAM). PAM is used to integrate many different authentication methods into a single interface. As the name suggests, it's modular.

Think of PAM as a power strip into which you can plug other authentication methods, such as Kerberos, LDAP, or the files databases. Authentication goes through PAM which can then authenticate via any number of methods. This provides a standard authentication Application Programming Interface (API) for developers and makes authentication dynamically configurable.

The Name Service Switch (NSS) is a facility that provides a list of databases to query for various system functions, including authentication. NSS is often used in conjunction with PAM and provides system authentication with a layer of transparency from an application or user's point of view.

PAM can authenticate users against a variety of databases configured by NSS. Multiple databases can be listed in priority order in the NSS configuration file, `/etc/nsswitch.conf`. When a user attempts to authenticate to a system, PAM searches the databases configured by NSS for user information. If it does not exist in the first database, it will then search the next one in line.

In the displayed example, the entries for the `passwd`, `shadow`, and `group` databases in this `nsswitch.conf` file have three different databases than can be used for authentication:

- The files database consists of the `passwd`, `shadow`, and `group` files.
- The sss database is provided via the System Security Service Daemon. It can manage access to several different remote directories and authentication mechanisms such as LDAP, FreeIPA, or Windows Active Directory, depending on its configuration.
- The ldap database is provided via the `nslcd` or the `nscd` daemon. `nslcd` is a daemon that can perform LDAP queries for local processes. The `nscd` daemon is a name service cache daemon that provides a cache for commonly used name service requests and is often used in conjunction with LDAP.

Client Authentication Services

Let's discuss the `sssd`, `nslcd`, and `nscd` client authentication services in more detail.

The service `sssd` allows PAM to authenticate against a local cache of account information. It can be used to store user credentials from a number of different databases which helps to reduce the load on authentication servers. Rather than having every client attempt to contact the servers directly, all of the clients can contact `sssd` which connects to the server on behalf of the client or authenticate against its cache. Thus, even if the authentication server is offline or there is no network connectivity, a user can still authenticate as a domain user to a local system.

`nslcd` or `nscd` are used to handle the authentication against LDAP servers. They provide an NSS module which enables PAM to go through them for LDAP authentication. `nslcd` and `nscd` are different, but can be packaged and installed together. For our purposes, the major difference between the two is that `nscd` provides credential caching while `nslcd` does not. You may see either or both of these running on a system depending on the administrator's feature requirements.

Next, let's look at a common utility which can query all of these authentication databases.

Enumerating Account Information

The `getent` command allows a user to "get entries" from the databases specified in the `/etc/nsswitch.conf` file using NSS libraries. NSS and `getent` are both bundled together in the `glibc` or The GNU C Library package. Since the `getent` command uses NSS, it can query for user information in a local flat file as well as network-based authentication databases.

The displayed table summarizes typical use of the `getent` command with the three database types discussed earlier:

- The `getent passwd` command retrieves account entries from all databases
- The `getent group` command retrieves group entries from all databases
- The `getent shadow` command retrieves password entries from all databases

As you can see in the table, you can also tell the `getent` command to query specific databases using the `-s` option followed by the desired database.

The syntax for the `getent` command `(getent [-s <database_type>] <database> [<entry>])` is displayed.

Please note that when using the `getent` command, the `sss` database does not allow recursive enumeration (enumerating all users/groups at once) of an entire database type by default. For instance, running `getent -s sss passwd` would not return any information unless enumeration was turned on. However, the `ldap` database does allow enumeration by default. So running `getent -s ldap passwd` would return all users on the LDAP server.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Users and Permissions: Section 3 Transcript

File Permission and Ownership: Classes of Users

Every file on a UNIX system is assigned ownership and permissions information at its creation. This information is stored within the corresponding inode to the file on disk. As it relates to file ownership, UNIX defines three classes of users - User, Group, and Other. Note that the acronym UGO (formed from the first letter of each class) is often used to refer to these three classes of owners. Select each tab to learn about each class of user. When you have finished learning about each class, select FORWARD in the navigation bar to continue with the module.

User (Owner): The User class represents the owner of the file. Generally, this is the person/entity that created the file.

Group: This class represents the group that owns the file. Only one group can own a file or a directory. The group of a file is usually set to the current GID of the user who created the file.

Other (world): The Other class, sometimes referred to as the World class, represents everyone on the system.

Ownership Commands

With the definition of each of the three ownership classes in mind, let's take a few moments to examine commands that allow the modification of ownership information for a file. UNIX offers two commands to modify file ownership information: `chown` and `chgrp`. These two commands adhere to the following rules enforced by the kernel:

- Only the root user can change the owner of a file
- A user can only change the group of a file when he or she is the root user or owns the file
- If a user owns a file, but he is not a root user, he can only change the group of a file to a group for which he or she is a member. For example, if the group of a file is "A" and the user belongs to the "B" group, but does not belong to the "C" group, then the user can change the group of the file from "A" to "B" (but may not change the group of the file to "C").

The most commonly used syntax for the `chown` command is: `chown newOwnerUsername Filename`

Notice that the `chown` command is followed by the username of the new file owner and then followed by the name of the file, or directory. This being said, another syntax option is to specify the new group name after the new owner name - just remember that each must be separated by a colon.

The most used syntax for the `chgrp` command is: `chgrp newOwnerGroup Filename`

Notice that new group name is followed by the name of the file or directory. Both the `chown` and `chgrp` commands accept the -R option, which essentially directs that the same ownership change be applied to everything (for which it has permission to change) in that directory and all its subdirectories. You should also note that the UID and GID of the new owner can be used in lieu of the username and group name, respectively.

Seems pretty simple, right? Now that you know the fundamentals about ownership commands, let's explore file access permissions.

File Access Permissions

In order to specify the permissions that determine who can access a file, each ownership class can be assigned one, or more, of three access permissions. Depending on whether you are dealing with a directory or a file, these access permissions are interpreted differently. Select each tab to learn about each type of access permission.

Read Permission:

- For a directory: Read means we can access the names of files in that directory
- For a file: Read means you can view or display the content of the file

Write Permission:

- For a directory: Write means you can modify information about the directory itself, such as its name and attributes, add files to the directory, delete files from the directory, and rename files in the directory
- For a file: Write means you can modify the file content

Execute Permission:

- For a directory: Execute means you can enter or traverse the directory to perform actions on its files and subdirectories
- For a file: Execute means you can execute the file when it is a script or a program

More About Permissions

A good way to remember the meaning of access permission for directories is to visualize a directory as a file that contains a list of filenames and a pointer to each file. Using this visualization construct and applying the permissions as if a directory was a file, you can easily deduce the read, write, and execute permission definitions for a directory.

It is also important to note that the *owner* access permissions supersede the *group* access permissions and *group* access permissions supersede the *other* access permissions. This means the owner of a file may be granted access to a file even though he or she belongs to the group that is denied access to that file.

Likewise, if the *other* class is permitted to read a file, but a user is a member of a group that is denied read access to that file, that user is denied read access to that file as well.

Next, let's explore access permissions using a concrete sample output of the `ls` command.

Access Permission Analysis

To display the access permissions for a file or directory use the `ls` command with the `-l` option. Reading right to left, we notice the:

- Filename: `/etc/passwd`
- Modification date and time: Aug 12 12:43
- Size of the file in bytes: 1134
- Group: system
- Owner: root
- Number of links to the file: 1
- File access permissions: rw-r--r--
- Type of file: a single dash (-) for an ordinary file. Keep in mind that this last field can take different values depending on the type of file. To name just a few we can have: "d" for a directory, "l" for a symbolic link, "c" for a special character file, and "b" for a special block file. To learn about other values, refer to the `ls` command in UNIX man pages.

Let's focus on the file access permissions. The string is composed of three blocks of three characters each placed side-by-side. The first character is reserved for the read access permission, the second character is reserved for the write access permission, and the third character is reserved for the execute access permission. Whenever one of the read, write, or execute permissions is not assigned, the no permission symbol, or (-) is used instead.

The leftmost block specifies the access permissions for the *owner*, the middle block specifies the access permissions of the *group*, and finally the rightmost block specifies the *other* user access permissions. In the displayed image, the *owner* has read and write access, but not execute access; the *group* has only read access, but not write or execute; and the *other* users have only read access, but not write or execute.

Changing File Access Permissions

The UNIX `chmod` command allows users to change file permissions. This being said, each user can only change the permissions of files for which he or she is the owner. The exception to this rule is that the root user can change permissions to all files on the system. The `chmod` command has two modes of operation: absolute-mode and symbolic-mode. Let's examine each mode. We will begin with symbolic-mode.

Symbolic-mode (Part One)

In the symbolic-mode the syntax for the `chmod` command is: `chmod` *`symbolic-mode-list`* `filename`.

In a symbolic-mode-list, every ownership class is represented by a letter. Specifically, "u" represents user, "g" represents group, "o" represents other (world) users, and "a" represents all users.

Letters for classes can be grouped to form one string and letters for file permissions (i.e., r, w and x) can also be grouped to form another string that determines permissions. These two strings are then appended one to another with separators, such as the "+" sign (indicating attached permissions are being added), the "-" sign (indicating attached permissions are being removed), or the "=" sign (indicating the setting of attached permissions) to form a symbolic-mode string. Multiple symbolic-mode strings can also be appended to each other in a comma-separated list to form the "symbolic-mode-list" part of the command.

Also, notice that the permissions string portion of the symbolic-mode string can be left blank. This indicates that no permissions are being added (+) or removed (-). Additionally, this indicates that the permission is being set (=) to no access for that class. Furthermore, as displayed, the permissions string portion of a symbolic-mode string can be set to indicate that we are adding, removing, or setting the permissions currently assigned of one class to another class.

Symbolic-mode (Part Two)

Whenever the `chmod` command is executed, the permissions in the symbolic-mode-list are consecutively applied to the attached user class for the file that is listed. For example, the displayed command adds the "read" and "write" permissions for the owner first and then sets the execute permission for the group. Last, it removes the "write" and "execute" permissions to the group and other (world) user types.

When using symbolic-mode syntax, there is an infinite number of ways to form the symbolic-mode-list to achieve the same result. Therefore, it's up to the user to compose the most optimal symbolic-mode-list that gets the job done. Let's examine some more command strings. What do you notice about the permissions in these strings?

Did you notice that the first two commands both set the permissions for all users to read? Did you recall that the third command is one that we reviewed earlier? This command adds read and write permission to the owner while setting the group permission to no access and setting the other or (world) permission to the same permissions as the owner. Great job!

Absolute-mode (Part One)

The absolute-mode is an alternative form of the `chmod` command. In absolute-mode, the syntax for the `chmod` command is: `chmod` *`absolute-mode`* `filename`.

In the absolute-mode a base value is assigned to each permission. Specifically, the number "4" is assigned to the read (r) permission, the number "2" is assigned to the write (w) permission, the number "1" is assigned to the execute permission, and "0" is assigned to represent no permission, or no access. Each of these base numbers, or values, can be used to calculate all the permissions you may wish to associate to a particular class.

Absolute-mode (Part Two)

To determine specific values of different combinations of permissions, we simply add together the base values for each desired permission. For example, let's imagine that for the file /home/johnd/comments.txt you would like to set user permissions to read (r) and write (w), group permissions to read (r) and execute (x), and other (world) permissions to write (w) and execute (x) as displayed.

To determine the desired user class permissions (in this case, read and write), we simply add together the values for the desired permissions, or 4 (for read) + 2 (for write), to determine that "6" is the permission value that should be used.

Next, to determine the desired group permissions (in this case, read and execute), we again add together the value for the desired permissions, or 4 (for read) + 1 (for execute), to determine that "5" is the permission value that should be used for setting the desired permissions.

For the other (world) class, you notice that the desired permissions displayed are write and execute. Using the base values to calculate the desired permissions, what permission value should we assign to the other user group?

Did you come up with the value of "3" - or 2 (for write) + 1 (for execute)? Wonderful!

To set the desired permissions using the absolute-mode notation, these total permission values are placed side-by-side as displayed. In other words, the owner class permission number is displayed first, then the group class, and finally the other class permission number.

Now that we have determined how to set permissions using `chmod`, let's take a moment to explore some results that will display when the `chmod` command is executed.

chmod Results

Notice that prior to running the `chmod` command the owner and group permissions are set to read, write, and execute (or rwx) and the other permissions are set to read (or r) only.

If we run the command `chmod 653 /home/johnd/comments.txt` followed by an `ls -l`, we notice that the owner permissions change to read and write (rw), the group permissions change to read and execute (rx), and the other permissions change to write and execute (wx).

Additionally, by running the command `chmod u+x,g-x,o= /home/johnd/comments.txt` followed by an `ls -l`, we notice that the owner permissions change to read, write, and execute, the group permissions change to read only, and the other permissions change so that this class of users has no permission, or access, to the file.

One disadvantage of the absolute-mode over the symbolic-mode is that you are unable to set permissions for a single user class, but must set the permissions for each class each time you are setting permissions. Another disadvantage of the absolute-mode is that you are unable to add or remove a specific permission to or from a specific class as you can when using the symbolic mode. In other words, you can only set permissions when using the absolute-mode.

umask Value

It is important to note that whenever files or directories are created on a UNIX system, they are assigned default access permissions. These permissions are usually 777 (or rwxrwxrwx) for directories and 666 (or rw-rw-rw-) for files, when the absolute-mode notation is used. The **umask**

value defines the permissions that are removed from the default permissions whenever a new file or a new directory is created. For example, if the umask value is 022; when a directory is created, the write permission will be removed from the group and other classes, which will give us rwxr-xr-x.

The umask value can be set with the `umask` command, using either absolute-mode or symbolic-mode notations as displayed. For more information about the `umask` command, consult the `man` pages or the Internet.

Knowledge Check Introduction

It is time for a Knowledge Check. This Knowledge Check will not be scored, but may indicate areas that you need to review prior to the Module Exam.

Users and Permissions: Section 4 Transcript

Special Permissions

1/10

Do you know that a user never directly accesses files on the filesystem? The user actually runs programs that lead to the creation of processes and these processes perform actions on the files for the user. For example, the user can run the `cat` command to open the `/etc/shadow file`. This will lead to the shell creating a process from the `/bin/cat` program; this process in turn will attempt to access the `/etc/shadow file`.

Each of these processes has two properties called the Effective User Identification (EUID) and the Effective Group Identification (EGID) that are usually set respectively to the UID and GID of the user that created the process. These two properties, the EUID and EGID, are used to determine what actions are allowed, depending on the permissions of each file. In our example, notice that the EUID of the cat process does not match the UID of the owner of the `/etc/shadow` file. Additionally, the EGID does not match the GID of the group of the file. Since there is no match, the other (or world) class permissions will be applied, which in this case means the user cannot read the file.

This being said, UNIX has special access permissions - Set User Identification (SUID) and Set Group Identification (SGID) - that can affect the default behavior we just discussed. We will examine both of these special permissions. In addition, we will also discuss another special permission known as a Sticky bit, even though it functions differently and is not directly related.

Special Permissions: SUID and SGID

2/10

Suppose the SUID bit is set on a program to which a user is granted execute permission. When the user runs that program, the process that results from executing that file has the EUID set to the UID of the user who owns the program file. This means the process gains all the access permissions of the user that owns the program file, even though the owner did not execute the program. Therefore, the user that executed the program effectively has permission to access whatever the owner of that program could access through that program.

For example, when a user executes the `passwd` command to change his password, the shell runs the`/bin/passwd` program that is owned by root. Since the `/bin/passwd` program has the SUID bit set, the resulting process will have the EUID equal to the UID of root. This makes it possible for the user to modify his password in the `/etc/shadow` file since it is owned by root and root has write access.

Additionally, UNIX systems also implement an SGID permission bit. If the SGID bit is set on a program file, when a user runs the program file, the process that results from executing that program has EGID set to the GID of the group that owns the program file. This means the process gains all the access permissions of the group that owns the program file.

For example, let's imagine that the `/bin/cat` program has the SGID bit set. Now, if a user is trying to read the `/etc/shadow` file by running the `cat` command, the cat process will have its EGID set to the GID of the group that owns the `/bin/cat` program. This would mean that the user would be able to read the `/etc/shadow` file since it is owned by the sys group and the sys group has read access.

In practice, an SGID bit is not seen nearly as often as SUID. Additionally, it is important to note that when applied to a directory, the SGID bit also causes the group that owns the directory to be assigned as the owning group of all files created within that directory.

Special Permissions: Sticky Bit 3/10

Historically, the last special permission we will discuss, the Sticky bit, has been used on UNIX systems to force the loading of executable files in such a manner that they stick in memory. In other words, once they were executed the first time, they would not need to be reloaded to be run again.

With modern versions of the UNIX operating system, virtual memory has been introduced, so each program is loaded in a separate private address space. This change means that it is no longer possible to maintain and share an executable file in memory. Therefore, the Sticky bit is no longer a workable option on executable files as it has no effect. This being said, the Sticky bit is still a viable option to use on a directory - although for a different purpose.

When a user is granted write permission for a directory, we know that the user can add files to, remove files from, and rename files in that directory. However, on some variants of UNIX, when the Sticky bit is set on a directory, it prevents unprivileged users from removing or renaming a file in the directory, unless they own the file or the directory. Since the Sticky bit restricts actions on a directory, it is sometimes referred to as a "restricted deletion flag."

The Sticky bit is usually set in world-writable directories like `/tmp` or tmp. Setting Sticky bit permissions is very similar to setting SUID and SGID permissions as users can set each of these special permissions using either the symbolic-mode or the absolute-mode. Let's take a few moments to examine the steps involved with setting each of these permissions.

Special Permissions: Symbolic-mode 4/10

In symbolic-mode, SUID and SGID permissions are enabled by adding the letter "s" to the user and/or group classes. The "s" is not added to the other (or world) class as adding it has no effect.

A Sticky bit permission is enabled by adding the letter "t" to the other (or world) class. It can also be enabled using +t without any class listed. As you may have already surmised, the "t" is not added to the owner and/or group classes as adding a Sticky bit has no effect on these classes.

Not too complicated, right? Good! Let's next explore how to set special permissions using the absolute-mode of the `chmod` command.

Special Permissions: Absolute-mode 5/10

In order to set special permissions using the absolute-mode of the `chmod` command, we simply assign a fourth number that is prepended to the usual absolute-mode permissions. Like with absolute-mode permissions discussed earlier, there are three base values that are used to calculate this digit. Assigned values include the number "4" for the SUID bit, the number "2" for the SGID bit, and the number "1" for the Sticky bit. As with other permissions, setting more than one of these bits can be accomplished by adding their values. That is it for enabling special permissions using the absolute-mode!

Now it's time to examine each of these special permissions in action. Let's begin by running an `ls -l` to view current permissions.

SUID and SGID Results

Take a few moments to review the current permissions for the files displayed. In particular, notice which classes have execute permissions set. Now let's use symbolic-mode to enable SUID and SGID permissions for the two files displayed.

When the SUID bit is enabled on a file and the file permissions are listed with the `ls -l` command as displayed, the letter "s" appears where the "x" would have been in the owner portion of the permissions string. However, this is only the case when the file has the owner execute permission set. In cases when the SUID bit is enabled and the owner execute permission is not set, the capital "S" appears in the owner portion of the permissions string.

Similar to the SUID bit, if both the SGID and group execute permissions are set, the output of `ls -l` will display the letter "s" in place of "x" in the group portion of the permissions string. Additionally, if the SGID bit is enabled and the group execute is not set, the permissions string will display either a capital "S" or "L" - depending on the operating system.

Next, let's examine a Sticky bit in action.

Sticky Bit Results

When the Sticky bit is enabled for a directory and permissions are listed with the `ls -l` command as displayed, the letter "t" appears where the "x" would have been in the other class section. However, this is only the case for a directory marked as executable. When a directory is not executable, a capital "T" appears. Notice that enabling the Sticky bit has no effect on files that are not directories.

At this point, you should be fairly comfortable with the symbolic-mode for SUID, SGID, and Sticky bit. Let's now discuss two useful SUID programs commonly found on UNIX systems: `su` and `sudo`. We will begin with the `su` command.

su - A Special SUID Program

The `su` command is a root SUID program that allows a user to impersonate another user without logging off. The syntax of the `su` command is: `su [-] [username]`.

When invoked, the `su` command requests the password of the listed username to be entered. This being said, if the invoker is root, then no password is required.

If the password is entered correctly, `su` creates a new shell process that has the EUID and EGID, as well as the real UID and GID and supplementary group list set to those of the target username. The new shell will be the default shell for username as it appears in the password file. If the target user doesn't have a shell listed, `/bin/sh` will be used instead.

When username is omitted, root is used as target username instead. When the tack option is used, `su` starts a login shell with an environment similar to a real login. This means that: all environment variables, except TERM, are cleared; environment variables HOME, SHELL, USER, LOGNAME, and PATH are initialized; and the current working directory is switched to the target user home directory and appropriate shell startup scripts are executed.

sudo - Another Special SUID Program

The `sudo` program is a root SUID program that allows users to run specific programs with the security privileges of another user (typically those of the superuser). It uses a configuration file, typically `/etc/sudoers`, which lists the users along with specific programs they are allowed to execute.

When `sudo` is used, the real UID and GID, as well as EUID and EGID are set to match those of the target user. Additionally, the group vector is initialized based on the group file and the shell environment variables are passed to the new process. The `runas` program in Windows provides similar functionality, but with more restrictions.

In the displayed image notice that the `sudo` command is being used to execute a `setup.sh` script as the root user.

Knowledge Check Introduction

It is time for a Knowledge Check. This Knowledge Check will not be scored, but may indicate areas that you need to review prior to the Module Exam.

Users and Permissions: Section 5 Transcript

Advanced Permissions - DAC

1/4

Some UNIX variants, such as Solaris, HP-UX, and Linux are equipped with features that allow the owner of a file to grant access permissions to that file to specific users, which is known as Discretionary Access Control (DAC). When using DAC to grant permissions, it is important to note that the permissions are not applied by following the classes of users, but are applied on a user-to-user basis. This is essentially the same way a Windows operating system performs access restriction to files.

Solaris and Linux offer two commands to perform discretionary access control: `setfacl` and `getfacl`. Take a few minutes to read more about these two commands using the `man` pages or the Internet, then select Forward in the navigation bar to continue with the module.

Advanced Permissions - MAC

2/4

The Mandatory Access Control (MAC) model greatly diverges from the DAC model. While the DAC model allows the owner of a file to grant specific permissions for files to specific users, the MAC model assigns security attributes to users and files and defines an authorization rule. Whenever a user tries to access a file, the kernel uses the authorization rule to determine if the access should be granted.

There are various implementations of MAC for UNIX variants. For example, AppArmor and SELinux are available on Linux, and Solaris Trusted Extension is available in Solaris version 10 and up.

Advanced Permissions - RBAC

3/4

Solaris (10 and up) and Linux (kernel 2.2 and up) implement a security model based on the concept of privileges. A privilege is a discrete right that allows a process to perform specific operations that were only granted to the superuser in the traditional model.

In the traditional model, some functions such as raw network access (i.e., the ability to send data frames directly on the physical medium) require full root privileges. It is for this reason that you will often see the `ping` command having the SUID permission bit set.

In the privilege model, however, you can assign these privileges in a much more granular manner. For instance, a program would now need to have `net_rawaccess` on Solaris systems or `CAP_NET_RAW` on Linux systems to be allowed raw access to the network.

Additionally, Solaris pushes the privilege model a little further by defining special user accounts called roles and rights profiles that list all the privileges that are associated to a specific role. Normal user accounts are then assigned to those roles in order to be able to perform various tasks on the system.

For more information about Solaris and Linux privileges, consult the Solaris privileges `man` page and the Linux capabilities `man` page.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Users and Permissions: Section 6 Transcript

Summary

You have completed the Users and Permissions module. You should now be able to:

- Discuss UNIX accounts and how they are represented on a UNIX system,
- Recognize UNIX files and directory ownership attributes,
- Use tools to view and change UNIX file ownership attributes,
- Determine how permissions affect access to files and directory objects,
- Use tools to view and change UNIX files and directory permissions, and
- Explain how special permissions change the inherited effective ownership identities of child processes.

To receive credit and advance to the next module, you must achieve a passing score on the Module Exam. Click the Next Section button to begin the Module Exam.