CHAPTER 10: BASH ADMINISTRATION

BY HANSIKA BHAMBHANEY

OBJECTIVES:

1.Installing bash as the Standard Shell

- How to replace the default shell with bash.
- POSIX mode.
- Command-line options for installation.

2.Environment Customization

- Setting file creation masks with `umask`.
- Limiting resources with `ulimit`.
- Types of global system-wide customizations.

3.System Security Features

- Using restricted shell mode (`rbash` or restricted options).
- Example of a system break-in scenario.

10.1 Installing bash as the Standard Shell

By default, many UNIX or Linux systems use shells like `sh`, `csh`, or `ksh` as the login shell. Replacing it means making `bash` the shell that automatically runs when you log in.

1.Find where bash is installed
Use commands like:

```
which bash
```

This will give you the path, e.g., `/bin/bash` or `/usr/local/bin/bash`.

2.Change your login shell
Use the `chsh` (change shell) command:

```
chsh -s /bin/bash
```

We might be asked to enter your password. After changing, log out and log back in — your new shell will be `bash`.

1.We get all the features of bash (like command-line editing, better scripting features, improved completion, etc.).

2.Easier to customize.

```
#!/bin/bash
IFS=:
for d in $PATH; do
    echo "Checking $d:"

    if [ -d "$d" ]; then
        cd "$d" || continue
        # Find shell scripts in the directory
```

```
        scripts=$(file * 2>/dev/null | grep 'shell script' | cut -d: -f1)
        for f in $scripts; do
            # Search for caret character (literal ^)
            if grep -q '\^' "$f"; then
                echo "  Found caret in $f"
            fi
        done
        # Return to previous directory
        cd - >/dev/null
    fi
done
```

1.Check if `$d` is a directory before `cd`.

2. Use `cd "$d" || continue` to skip if `cd` fails.

3. Quote variables (`"$f"`, `"$d"`) to handle spaces in names.

4. Suppress error messages with `2>/dev/null` when using `file *`.

5. Add a message when a caret character is found, to make the output clearer.

6. `cd - >/dev/null` returns to the previous directory silently.

10.1.1POSIX mode

POSIX (Portable Operating System Interface) is a standard that defines how UNIX-like systems behave to ensure compatibility. Bash includes a **POSIX compatibility mode**, which makes it behave like a standard POSIX shell (`sh`).

1.If you want your scripts to run consistently on different UNIX systems that only guarantee POSIX features.

2.To avoid using bash-specific extensions accidentally.

- **How to enable POSIX mode?**

Start bash with the `--posix` option:

```
bash --posix
```

10.1.2 COMMAND - LINE OPTIONS

bash has several command-line options that change the behavior of and pass information to the shell. The options fall into two sets: single character options, and multicharacter options, which are a relatively recent improvement to UNIX utilities.

Common options:

- `--login`: Make bash act as a login shell (executes login-related startup files like `.bash_profile`).
- `--noprofile`: Do not read any profile files on startup.
- `--norc`: Do not read `.bashrc` on startup (useful to troubleshoot).
- `--posix`: Enable POSIX mode.
- `-c 'command'`: Execute a given command string and exit.

The multicharacter options have to appear on the command line before the single-character options. In addition to these, any set option can be used on the command line. Like shell built-ins, using a + instead of - turns an option off.

```
bash --login --noprofile
```

- Starts a login shell but skips loading profile files (useful for clean testing).

Option	Meaning
	is not C or POSIX. This also turns on the **-n** option.
-i	Interactive shell. Ignores signals TERM, INT, and QUIT. With job control in effect, TTIN, TTOU, and TSTP are also ignored.
-l	Makes *bash* act as if invoked as a login shell.
-o *option*	Takes the same arguments as **set -o**.
-O, +O *shopt-option*	*shopt-option* is one of the shell options accepted by the **shopt** builtin. If *shopt-option* is present, **-O** sets the value of that option; **+O** unsets it. If *shopt-option* is not supplied, the names and values of the shell options accepted by **shopt** are printed on the standard output.

Option	Meaning
	If the invocation option is **+O**, the output is displayed in a format that may be reused as input.
-s	Reads commands from the standard input. If an argument is given to *bash*, this flag takes precedence (i.e., the argument won't be treated as a script name and standard input will be read).
-r	Restricted shell. See the Section 10.3.1 later in this chapter.
-v	Prints shell input lines as they're read.
-	Signals the end of options and disables further option processing. Any options after this are treated as filenames and arguments. — is synonymous with **-**.

Option	Meaning
—noprofile	Does not read the startup file */etc/profile* or any of the personal initialization files.
—norc	Does not read the initialization file *~/.bashrc* if the shell is interactive. This is on by default if the shell is invoked as *sh*.
—posix	Changes the behavior of *bash* to follow the POSIX guidelines more closely where the default operation of *bash* is different.
—quiet	Shows no information on shell startup. This is the default.
—rcfile *file*, —init-file *file*	Executes commands read from *file* instead of from the initialization file *~/.bashrc* if the shell is interactive.

Option	Meaning
—debugger	Arranges for the debugger profile to be executed before the shell starts. Turns on extended debugging mode and shell function tracing.[7]
—dump-strings	Does the same as **-D**.
—dump-po-strings	Does the same as **-D** but the output is in the GNU *gettext* po (portable object) file format.
—help	Displays a usage message and exits.
—login	Makes *bash* act as if invoked as a login shell. Same as **-l**.
—noediting	Does not use the GNU *readline* library to read command lines if interactive.

10.2 Environment Customization

File creation control with umask:

- umask defines which permission bits should be turned off when new files and directories are created.
- Default permissions are 666 for files and 777 for directories before umask is applied.
- For example, a umask of 022 results in file permissions 644 (rw-r--r--) and directory permissions 755 (rwxr-xr-x).
- Helps ensure files are not accidentally made writable by others, improving security.
- umask can be set in shell configuration files to apply automatically to all new sessions.

Limiting resources with `ulimit`:

- `ulimit` restricts the amount of system resources a user or process can consume.
- It can limit maximum file size (`-f`), number of open files (`-n`), number of processes (`-u`), CPU time (`-t`), and other resources.
- Example: `ulimit -n 100` limits the number of open files to 100.
- Prevents individual users or runaway processes from overloading the system can be configured in global or user-specific profile scripts to enforce limits consistently.

- **Global system-wide customizations:**
    - Administrators can define environment settings that apply to all users across the system.
    - These customizations are typically placed in files like `/etc/profile`, `/etc/bash.bashrc`, and `/etc/environment`.
    - Examples include setting the default `PATH`, defining standard `umask`, creating useful aliases, setting default editors, and customizing prompts.
    - Ensures a consistent environment for all users, reduces configuration mistakes, and enforces basic security and usability policies.
    - Simplifies administration by centralizing important shell behaviors in one place.

10.2.2 ulimit

Option	Resource limited
-m	Maximum resident set size
-n	File descriptors
-p	Pipe size (512 byte blocks)
-s	Process stack segment (Kb)
-t	Process CPU time (seconds)
-u	Maximum number of processes available to a user
-v	Virtual memory (Kb)

Option	Resource limited
-a	All limits (for printing values only)
-c	Core file size (1 Kb blocks)
-d	Process data segment (Kb)
-f	File size (1 Kb blocks)
-l	Maximum size of a process that can be locked in memory (Kb)[9]

10.3 System Security Features

Using restricted shell mode (rbash or restricted options):

- Bash can be started in **restricted mode**, either by invoking it as `rbash` or by using the `--restricted` option.

- In restricted mode, users are prevented from performing certain potentially dangerous operations. Examples include:

    - Changing the working directory using `cd`.

    - Setting or changing the `PATH` or `SHELL` variables.

    - Redirecting output to files (`>`, `>>`).

    - Executing commands containing slashes (`/`), which prevents running arbitrary binaries from unexpected locations.

- This mode is useful when giving limited shell access to guest accounts or in controlled environments like application shells or kiosk systems.

- Helps restrict user actions and reduce the risk of accidental or intentional misconfigurations or escapes.

Example of a system break-in scenario:

- A user might try to exploit weak permissions or misconfigured environments to gain unauthorized access or elevate their privileges.

- For example, a user might overwrite or modify startup files (such as `.bash_profile` or `.bashrc`) to inject malicious commands that run when other users log in.

- Another scenario: if users can set their own `PATH`, they might place a malicious executable with the same name as a common system command (like `ls` or `cat`) earlier in the `PATH`, tricking others into running it.

- Improperly secured temporary files or world-writable files can also be replaced or tampered with during a session, leading to code execution or data theft.

- These examples highlight the need for strict permissions, cautious environment variable handling, and secure shell configurations.

Privileged mode and how to handle it safely:

- When Bash is run with **elevated privileges** (such as root), it enters what is called privileged mode.

- In privileged mode, Bash ignores certain user-specific startup files (like `.bashrc` and `.profile`) to avoid executing potentially unsafe or malicious code from user directories.

- This behavior is crucial for preventing privilege escalation through manipulated user config files.

- Administrators should always verify that shell configurations and environment variables are secure when running as root.

- Using `sudo` or direct `su` to root should be done carefully, and scripts that run with root privileges should explicitly define and sanitize their environment.

Codes and commands

1.Switching to restricted shell (rbash)

```
rbash
```

These commands start a restricted shell session, limiting the user's capabilities.

2. Changing `umask`

```
umask 022
```

Sets default file permissions so that group and others do not get write access.

3. Setting resource limits

```
ulimit -n 100
```

Limits the number of open file descriptors to 100.

Example to install bash as login shell

```
chsh -s /bin/bash
```

This command changes your default login shell to bash (requires the correct path to bash).

4.Checking current shell

```
echo $SHELL
```

5.Changing PATH carefully (as part of explaining break-in scenarios)

```
export PATH=/safe/dir:/usr/bin
```

This shows explicitly setting a safe PATH to avoid malicious binaries.