Hardening Applications

Fortificación de S.O.

Master en Seguridad Informática. 2024/25

Universidade da Coruña

Universidade de Vigo

Antonio Yáñez Izquierdo
José Rodríguez Pereira

Contents I

Identifying and eliminating unused applications

Not needed applications

- every application in the system poses a risks, we can think of two kinds of applications
    - the ones with known security holes
    - the ones whose security holes we haven't yet heard off
- so we should get rid of al those applications we don't need.
    - if we don't know about an application: it clearly is not needed

Not needed applications

- this can be trickier on desktop systems as there are many dependencies among libraries and applications
- in the server world, for example, a mail server does not need the graphical desktop environment, so we can do with just the basic system and network utilities, the mail and ssh servers, and maybe some *spam* detection software, which makes it easier to harden (from an application point of view)
- the package management in our system can inform us of the packages installed and the files comprising each package

Not needed applications

- there are utilities that find unused packages based on the time stamps of the executable files
  - for example `https://codeload.github.com/epinna/Unusedpkg/zip/master` checks for unused packages in `'deb'` based distributions, and informs the number of days a package has not been used
- note that one of the recommended mount options for SSD (*noatime*) makes this type of results unreliable. Fortunately, wronly deleted packages can be installed again

Limiting applications resources

Limiting applications resources
→pam_limits

/etc/security/limits.conf

- we can establish limits in the the file `/etc/security/limits.conf` that affect a user session
- for that we have to specify *pam_limits* as a session module for the login (or @common) service
- we can impose limits on number of simultaneous logins, number of processes, or CPU or memory usage
- this limits, however, affect to a session, not to the individual applications
- *hard* limits are enforced and cannot be changed. Users however can change the *soft* limit within the values of the *hard* limit.

Limiting applications resources
→cpulimit and prlimit

setcpulimit

- the program **cpulimit** (available through `apt-get install cpulimit` on debian based systems) allows us to limit the amount od CPU (percentage) a process uses
- if does so by the use of SIGSTOP and SIGCONT
- the use of this sinals can have side effects, specially with some job control shells
- as always, `man cpulimits` gives us information on its usage
- the program **prlimit** allows us to impose limits on the resources allocated to a process
- limits can be set on resident set size, stack sice, number of open files ...
- we can use **prlimit** to limit the impact of some program on a running system

Limiting applications resources
→cgroups

cgroups

- **cgroups** is a feature of linux kernel tha enables us to to limit the resource usage for a set of processes.
- some of the virtualization tools, for example LXC, are based on *cgroups*
- from the cgroups manpage: *A cgroup is a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem*
- we can limit resources allocated to those proceses treating them as a unique set

cgroups

- cgroups provides[1]
  - **Resource limiting** a group can be configured not to exceed a specified memory limit or use more than the desired amount of processors or be limited to specific peripheral devices.
  - **Prioritization** one or more groups may be configured to utilize fewer or more CPUs or disk I/O throughput.
  - **Accounting** a group's resource usage is monitored and measured.
  - **Control** groups of processes can be frozen or stopped and restarted.

[1] Petros Koutoupis: Everything You Need to Know about Linux Containers

cgroups

- there are two implementation of cgroups (not compatible with earch other: cgroups1 and cgroups2)
- there's several interfaces to *cgroups* (not all of them available in every linux distribution)
  - manual interface through the `/sys/fs/cgroup` filesystem
  - through the programs `cgcreate, cgclassify ...` and the file `cgconfig.conf` available through the **libcgroups**
  - througfh the client `cgm` communicating with the *cgmanager*daemon (package **cgmanager**)
  - through `systemd`

cgroups: example

- example: we are going to show how we can limit the memory allocated to one process
- we create the cgroup *'limitadoMemoria'*

```
root@hardening:/home/antonio# mkdir /sys/fs/cgroup/memory/limitadoMemoria
root@hardening:/home/antonio#
```

- we define the max amount of memory

```
root@hardening:/home/antonio# echo 50000000>  /sys/fs/cgroup/memory/limitadoMemoria/memory.limit_i
root@hardening:/home/antonio#
```

- the only thing left to do is to add the pid of the process we want to limit to the file

```
root@hardening:/home/antonio# echo 3577 >> /sys/fs/cgroup/memory/limitadoMemoria/cgroup.procs
```

cgroups: example

- The previous example in a debian 12 machine (with cgroups v2) will look like this
- we create the cgroup *'limitadoMemoria'*

```
root@hardening:/home/antonio# mkdir /sys/fs/cgroup/limitadoMemoria
root@hardening:/home/antonio#
```

- we define the max amount of memory

```
root@hardening:/home/antonio# echo 50000000>  /sys/fs/cgroup/limitadoMemoria/memory.high
root@hardening:/home/antonio#
```

- the only thing left to do is to add the pid of the process we want to limit to the file

```
root@hardening:/home/antonio# echo 3577 >> /sys/fs/cgroup/limitadoMemoria/cgroup.procs
```

cgroups

- *cgroups* are hierarchical, a *cgroup* can be created inside another *cgroups*
- processses created by a process belonging to a *cgroup* belong to that *cgroup*, although they can be changed to another *cgroup*
- processes belongin to a *cgroup* can be frozen (or unfrozen) just by writing an 1 (or a 0) to the file `cgroup.freeze` in the *cgroup* directory
- a description of the files in a cgroup and their meaning can be seen at

`https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#interface-files`

cgroups: summary

- we can create a *cgroup* by just creating a directory in `/sys/fs/cgroups/` depending on which resource we want to control/monitor
- we can add processes to that croup by adding their PIDS to the file `cgroup.procs`
- we then can impose the limits (or check values) via the files in `/sys/fs/cgroups/name-of-group-created`
- alternatively (and depending on the linux distro) we can use the programs available in **libcgroup** or with the **cgmanager** interface

Executing in chroot jails

chroot system call

- by design, a process in linux knows only two directories
  - the current working directory
  - the root directory
- finding a file begins in the root directory (filename starts with "/") or in the current directory
- the *chdir* system call changes the current working directory for a process
- the *chroot* system call changes the root directory for a process (it is a privileged call)

chroot system call

- a chrooted program cannot access files outside its chroot as it does not have way to name them
- at startup, programs expect to find configuration files, device nodes and shared libraries at certain preset locations. For a chrooted program to successfully start, the chroot directory must be populated with a minimum set of these files. We could also have the files opened before and the descriptors preserved through the chroot system call
- we usually use *chroot*ed environmets
  - to test software, a chrooted environment with just its elements
  - some servers (ftp and web servers typically)
  - to rescue system, after booting from instalation media

chroot

- in linux, a *chrooted* environmet to be fully functional would also need the kernel virtual file systems. So, should we wanted a working environment chroot to $TARGET, we must

```
TARGET="/waterver/dir/we/want"
mount -t proc proc $TARGET/proc
mount -t sysfs sysfs $TARGET/sys
mount -t devtmpfs devtmpfs $TARGET/dev
mount -t tmpfs tmpfs $TARGET/dev/shm
mount -t devpts devpts $TARGET/dev/pts
```

- we'd also like to copy the `/etc/resolv.conf` before chrooting
- note that with access to the devices, a user with access to the devices can elude the *chroot*

Virtualization environments

virtualization environments

- we've seen so far that
  - we can limmit resource usage, for example through *cgroups*
  - we can limit what part of the filesystem they see through *chroot*
- the next step in isolating the O.S. from possible application *'malfunction'* is having it run in a virtualized environment (VE)
- an VE is different from a Virtual Machine (as created by tools like VirtualBox or VMWare) in that it requires much less resources and overhead as the VM includes the entire OS and machine setup, including hard drive, virtual processors and network interfaces
- we usually refer to this as *container based virtualization*

virtualization environments

- compared to VMs, containers generally offer less isolation because they share portions of the host kernel and operating system instance.
- linux has its container based virtualization environment called LXC (linux containers)
- the first thing is to create a container. We just have to provide a name for the container and a template to create the container from

```
root@abyecto:~# lxc-create -t ubuntu -n PruebaContainers
```

virtualization environments

- the list of templates available is usually a `/usr/share/lxc/templates`

```
antonio@abyecto:~$ ls -l /usr/share/lxc/templates/
total 408
-rwxr-xr-x 1 root root 13160 Jan 29  2018 lxc-alpine
-rwxr-xr-x 1 root root 13704 Jan 29  2018 lxc-altlinux
-rwxr-xr-x 1 root root 11373 Jan 29  2018 lxc-archlinux
-rwxr-xr-x 1 root root 12159 Jan 29  2018 lxc-busybox
-rwxr-xr-x 1 root root 29725 Jan 29  2018 lxc-centos
-rwxr-xr-x 1 root root 10374 Jan 29  2018 lxc-cirros
-rwxr-xr-x 1 root root 20243 Jan 29  2018 lxc-debian
-rwxr-xr-x 1 root root 17914 Jan 29  2018 lxc-download
-rwxr-xr-x 1 root root 49693 Jan 29  2018 lxc-fedora
-rwxr-xr-x 1 root root 28384 Jan 29  2018 lxc-gentoo
-rwxr-xr-x 1 root root 13868 Jan 29  2018 lxc-openmandriva
-rwxr-xr-x 1 root root 15946 Jan 29  2018 lxc-opensuse
-rwxr-xr-x 1 root root 41791 Jan 29  2018 lxc-oracle
-rwxr-xr-x 1 root root 11570 Jan 29  2018 lxc-plamo
-rwxr-xr-x 1 root root 19242 Jan 29  2018 lxc-slackware
-rwxr-xr-x 1 root root 26862 Jan 29  2018 lxc-sparclinux
-rwxr-xr-x 1 root root  6862 Jan 29  2018 lxc-sshd
-rwxr-xr-x 1 root root 25705 Jan 29  2018 lxc-ubuntu
-rwxr-xr-x 1 root root 11734 Jan 29  2018 lxc-ubuntu-cloud
antonio@abyecto:~$
```

virtualization environments

- we start the machine and see that is running ok

```
root@abyecto:~# lxc-ls -f
NAME            STATE   AUTOSTART GROUPS IPV4 IPV6
PruebaContainer STOPPED 0         -      -    -
root@abyecto:~#
root@abyecto:~#
root@abyecto:~# lxc-start -n PruebaContainer -f /var/lib/lxc/PruebaContainer/config
root@abyecto:~# lxc-ls -f
NAME            STATE   AUTOSTART GROUPS IPV4 IPV6
PruebaContainer RUNNING 0         -      -    -
root@abyecto:~#
```

virtualization environments

- the root file system for the container is at `/var/lib/lxc/`*container_name*`/rootfs`
- its configuration at `/var/lib/lxc/`*container_name*`/config`
- we start the machine in the foregound with -F
- to manipulate the machine we can use the lxc-* commands

```
 root@abyecto:~# lxc
lxc-attach       lxc-checkpoint  lxc-create       lxc-freeze       lxc-monitor      lxc-unfreeze
lxc-autostart    lxc-config      lxc-destroy      lxcfs            lxc-snapshot     lxc-unshare
lxc-cgroup       lxc-console     lxc-device       lxc-info         lxc-start        lxc-usernsexec
lxc-checkconfig  lxc-copy        lxc-execute      lxc-ls           lxc-stop         lxc-wait
root@abyecto:~# lxc
```

autostarting containers

- the configuration of the container is usually in `/var/lib/lxc/`*container_name*`config`
- here we can decide on the type of network connection of the container and othe things
- should we want the container to start automatically when the system boots win whould use

```
lxc.start.auto = 1
lxc.start.delay = 5
```

- which would autostart the container on boot, with a delay of five seconds

virtualization environments

- if you want to run lxc as a normal user you have to
  1. add the following lines to file `.config/lxc/default.conf`

     ```
     lxc.id_map = u 0 100000 65536
     lxc.id_map = g 0 100000 65536
     ```
  2. add the line `kernel.unprivileged_userns_clone=1` to the file `/etc/sysctl.d/local.conf` and then execute `sysctl --system`
  3. change the permissions of `.local` and `.local/share` to *rwxr-xr-x*
  4. use the download template

virtualization environments

- there are other container based virtualization solutions for linux
- the two most widespread are
  - LXD
  - docker
- both of them rely on *cgroups* and *lxc* libraries

M.A.C

D.A.C. versus M.A.C.

- D.A.C. stands for Discrectionary Access Control, meaning that the owner decides on *who can do what* on his/her files and directories. Most operating systems use D.A.C.
- M.A.C. stands for Mandatoy Access Control, means that the O.S. enforces a policy on *who can access what* regardless of the user's given permissions
  - a user might not mind that others users accessed his/her files, although it could pose a security risk or maybe go against his/her employer's policy
- in M.A.C. systems a least privilege approach is used, when a process wants to access a file,
  - first the D.A.C is checked, if it denies access, access is denied
  - if D.A.C allows access then the M.A.C. is checked and if M.A.C. denies access, access is denied

M.A.C. in linux

- the two M.A.C. solutions in linux are *SELinux* and *apparmor*
- In SELinux every object in the system is labeled and access is only permitted if there is a rule allowing it explicitly
- it is mainly used in used in redhat and derivatives (fedora . . .)
- in apparmor we have a file defining the privileges of an apliccation (called the app profile)
- it is mainly used in debian and derivatives (ubuntu, devuan . . .

M.A.C
→AppArmor

AppArmor

- AppArmor is a mandatory access control system for Linux.
- In AppArmor the kernel imposes restrictions on paths, sockets, ports, and various input/output mechanisms
- It was developed by Immunex and now is maintained by SUSE
- It requires kernel 2.6.36 and is installed by default in debian since debian 10 (*buster*)

AppArmor

- we can check whether it is enabled with

  `# cat /sys/module/apparmor/parameters/enabled`
- the command `aa-status` lists all the loaded profiles
- the `-Z` option of command *ps* shows the status of confinement of processes

AppArmor

- for each application under apparmor control we have a profile in

    `/etc/apparmor.d/`

- we can see the profiles loaded with aa-status
- the profile file contains the restrictions imposed to the program in represents
- apparmor has two modes of operation
    - **enforce mode** restrictions are actually imposed
    - **complain mode** violations of restrictions are allowed but logged

AppArmor

- we can load an application profile with `-r` replaces the one in use (if any)

  ```
  # apparmor_parser -r /etc/apparmor.d/profile_name
  ```

- we can disable an application profile with (disabled profiles are put on `/etc/apparmor/disable`)

  ```
  # aa-disable /path/to/executable
  ```

- and we can put the in enforce or complain mode with

  ```
  # aa-enforce /path/to/executable
  # aa-complain /path/to/executable
  ```

AppArmor

- We can create an (empty) profile with

```
# aa-easyprof ejecutable > /etc/apparmor.d/nombre_ejecutable
```

- After that we edit the profile file (is a plain text file) to meet our needs
- We load it

```
# apparmor_parser -r /etc/apparmor.d/profile_name
```

- And we have that app *'apparmored'*
- The following page shows the profile for an executable file (`/usr/bin/listar`), that can only access the /usr directory and ALL of its descendants except directories under `/usr/share/doc`

AppArmor

```
# vim:syntax=apparmor
# AppArmor policy for list
# ###AUTHOR###
# ###COPYRIGHT###
# ###COMMENT###

#include <tunables/global>
# No template variables specified
/usr/bin/listar {
  #include <abstractions/base>

  # No abstractions specified

  # No policy groups specified

  # No read paths specified
/usr/  r,
/usr/** r,
deny /usr/share/doc/** r,
  # No write paths specified
}
```

M.A.C
→SELinux

SElinux

- is a series of kernel patches that allows linux to use M.A.C
- in SElinux every object (applications, files . . .) is labeled
- access is only permitted if there is an specific rule in the system's policy allowing it
- when there is not specific rule access is denied
- example:
  - the executable for the web server is labeled `httpd_exec_t`,
  - its configuration file has label `httpd_config_t`
  - . . .
  - any process running in the `httpd` context can only interact with objects labeled `httpd_*_t`

SElinux policy

- the working of SELINUX is controlled by its policy
- a SELINUX policy is a collection of SELINUX rules
- each rule describes an iteraction between al process (textbflabel) and a file (textbflabel)

```
ALLOW apache_process apache_log:FILE READ;
```

SElinux

- SElinux can be enabled or disabled
- if it is enabled in can be in either one of this two modes
  - **enforcing** the active policy is enforced, denying access when necesary (a log entry is generated only the first time a access is denied)
  - **permissive** the policy is not enforce, each time an access should be denied, a log entry is generated
- the commands **getenforce** and **setenforce** allow us to view and set the current mode

SElinux in debian

- Although SELinux it is not the prefered method of M.A.C in debian, we could enable syslinux in that distro
- to enable SElinux in debian we must
  - have debian installed in ext2, ext3. ext4 or jfs file systems
  - get the default policy and the basic utilities by installing the following packages

```
# apt-get install selinux-basics selinux-utils selinux-policy-default auditd
```

  - run `selinux-activate` to configure the grub and get the system relabeled at the next reboot (through the existence of the file `/.autorelabel`)
  - `enforcing` or `permisive` mode will be defined in `/etc/selinux/config`

SElinux in debian

- once we have SElinux running, each file and/or process is labeled with what we call a *selinux context*.
- a *selinux context* consists of four labels
  `selinux_user:selinux_role:selinux_type:selinux_level`
- we can see the the context of files and/or processes adding the parameter `-Z` to the *ls* and/or *ps* commands

  ```
  $ ls -Z /etc/passwd
  system_u:object_r:passwd_file_t:s0 /etc/passwd
  ```
- the commands *chcon*, *restorecon*, *secon* and *runcon* allow us to acsess/modify the context of files or processes