

# Hardening the boot procedure

Fortificación de S.O.

Master en Seguridad Informática. 2024/2025

Universidade da Coruña

Universidade de Vigo

Antonio Yáñez Izquierdo

José Rodríguez Pereira

# Contents I

## 1 The Boot procedure

- Understanding the boot procedure
- booting steps
- BIOS and UEFI type firmware
- Partitions
- MBR and GPT partitions

## 2 Securing the firmware

## 3 The Grub boot loader

- The Grub boot loader
- The Grub boot loader. Vulnerabilities

## 4 Securing the Grub boot Loader

- Grub users and superusers
- Grub Passwords

## 5 Other boot loaders

# The Boot procedure

# The Boot procedure

## → Understanding the boot procedure

# booting

- booting is the process by which the O.S. is loaded and the system is ready to be used by users
- as the O.S. provides the services necessary for the system to be usable
  - those services would be necessary to load the O.S.
  - the O.S. must be loaded without those services in what we call the *bootstrapping* process
  - usually a loader of the O.S. is loaded and executed and it is this loader that loads the O.S.

# automatic booting

- the booting process is very hardware dependent
- we can distinguish between two ways of booting
  - automatic
  - manual
- *automatic booting* is the way the system boots most of the times.
  - it does not require human intervention
  - the system boots by its own and a multiuser environment is available after booting
- we perform the *manual booting* when there's some kind of error and we need to oversee the booting process

# The Boot procedure

## → booting steps

# booting steps

- although it is very dependent on the hardware, the booting process can be thought of consisting of the following steps
  - 1 loading and executing the *motherboard firmware* boot program
  - 2 loading and executing the *boot loader* (how this is done depends on the type of motherboard firmware: BIOS, UEFI, openboot . . . ). It can consist of several *stages*
  - 3 loading and executing the *linux* kernel
  - 4 running the initialization *scripts* and starting the system services

# first booting step: motherboard firmware

- the motherboard firmware contains some code to start the booting of the machine
- how this code works depends on the type of firmware.
- most present linux systems run on one of these platforms:  
*intel x86* platform or *amd64* platform
- for these platforms there are two types firmware
  - a) BIOS type firmware. All 32 bit intel x86 machines and some amd64 machines
  - b) UEFI type firmware. Newer amd64 machines
- this firmware does not load the kernel directly (neither it is capable of) but loads a program, *the boot loader* that itself loads the kernel

## second booting step: the boot loader

- the boot loader is (*should be*) a simple program which only has to load the kernel
- its configuration file has two essential items to define
  - which kernel to load (and where to find it)
  - which device to use as root file system when that kernel is loaded (and pass this information to the kernel)
- **UNFORTUNATELY** most of the present bootloaders include some *non essential* options such as *splash images*, *menus* . . . which makes them bigger, slower and more tedious to install and configure.
- some boot loaders understand filesystems, so the kernel location can be specified directly in the boot loader configuration file
- some boot loaders DO NOT UNDERSTAND filesystems, so some additional steps need to be taken to make the boot loader aware of the kernel location

## third booting step: loading and executing the linux kernel

- the kernel is loaded in memory and trasfered control
  - in linux the kernel typically resides in the file `/boot/vmlinuz....`
  - as the linux kernel is modular (some of its functionality resides on modules), sometimes it needs a minimal filesystem in memory to provide it with modules necessary to boot (the *initial ram disk* or *initrd* file)
- it creates its data structures, probes for devices and performs initialization routines
- creates *init*, the first “*user process*” in the system (some systems have substituted *init* by *systemd*) which will initiate the various services

## fourth booting step: running the initialization scripts

- the initialization scripts perform routine booting tasks (such as properly configure the hardware) and start the system services
  - `init` reads its configuration file (`/etc/inittab`) where it gets the *runlevel to boot to* (`systemd` has a default *target* to boot to, that can be changed with `systemctl set-default`. Available *targets* can be seen with `systemctl -list-units --type target`)
  - if there is some kind of error or the system is configured to boot into single user mode, a root shell is created with only the *root filesystem* mounted
  - otherwise the scripts initiating the system services are started (`/etc/rcN.d` directories for different *runlevels*). (Different *systemd targets* start a different set of services).
- among the services started are the login (both text and graphical) programs.
- after the scripts are run, the machine is ready to use

# The Boot procedure

## → BIOS and UEFI type firmware

## booting with BIOS type firmware

- by construction, when the system is powered on (or when a reset is done) the motherboard executes the code at certain memory addresses, there resides the firmware
- this code contains some initialization routines and sometimes access to a system configuration menu
- a device is defined as the first boot device (CD/DVD, disk, tape, usb, floppy . . . ). An attempt is made to boot from that device, if unsuccessful, the defined as second boot device is tried and so on
- this firmware **DOES NOT UNDERSTAND** filesystems so: for this type of firmware *booting* from a device means **loading the first block and executing the code in it**
- there's no interface to this firmware to be accessed from the O.S.

# booting with BIOS type firmware

- in this type of firmware booting from a device means **LOADING THE CODE AT THE FIRST BLOCK OF THAT DEVICE AND EXECUTING IT**
- when the device is a disk the first block of the disk contains some boot code and the partition table
- the usual code to have in this first block of disk (also called Master Boot Record) is to have very simple program, that reads the partition table, looks which partition has the *active* flag on and then loads that partition's first block and executes it: the boot loader code (at least its first stage) can be copied to that disk block

## booting with BIOS type firmware

- the partitions scheme for this type of firmware is called MBR partitions
- to install a boot loader in one of this system we can
  - install it at the Master Boot Record (first block on disk): that bootloader will execute upon switching the machine on regardless of the *active* partition
  - install it at the first block of the partition: that bootloader will execute when the partition is marked active and there's no other bootloader at the MBR

## booting with UEFI type firmware

- by construction, when the system is powered on (or when a reset is done) the motherboard executes the code at certain memory addresses, there resides the firmware
- this code contains some initialization routines and sometimes access to a system configuration menu
- there's an interface to access the firmware booting configuration from the O.S. (in linux the program *efibootmgr* does it)
- this firmware **UNDERSTANDS THE FAT FILESYSTEM** so a boot loader is just a program in a FAT filesystem
- this firmware is capable of running executables in its own format (.efi). This is used to run the bootloaders

# booting with UEFI type firmware

- disks must be partitioned using a GPT partition table
- there must exist, at least, an EFI System Partition (ESP)
  - this partition must be formatted using either the FAT16 or FAT32 filesystems
  - this partition holds, among other things, the EFI drivers and the EFI bootloaders
  - Operating Systems typically place their bootloaders in a subdirectory of the EFI directory in the ESP
  - booting different operating systems can be done at the firmware level
- unless otherwise specified, the firmware will run the program  
    \EFI\BOOT\BOOTX64.EFI

# booting with UEFI type firmware

- installing a boot loader in a machine with UEFI firmware means
  - 1) copying the executable file (.efi format) to the EFI System Partition
  - 2) if we want that boot loader to be run at boot time we must tell the firmware to: we can do so in the firmware setup program or from the O.S. (in linux we can do that with the **efibootmgr** command)
- should this executables be required to be signed, the booting procedure would be known as *secure boot*
- the EFI variables define which of these .efi files must be loaded when booting

# The Boot procedure

## → Partitions

# disks

- disk are used to create filesystems on them
- several filesystems can exist on a disk device in what we usually call *partitions*
- several *partitions* can be combined into one filesystem via **Logical Volume Management** software

# partitions

- a disk is usually divided into several units called *partitions*
- filesystems are created in *partitions*, usually one filesystem in each *partition* although several *partitions* can be combined into one filesystem via Logical Volume Management Software
- we even can install different O.S.s in different partitions
- in linux disk are designated as `/dev/sda`, `/dev/sdb`  
`/dev/sdc` ...
- partitions on disk `/dev/sda` are designated as `/dev/sda1`,  
`/dev/sda2.`, ...

# partition tables

- each disk has a table, usually located at the first block, that defines the partitions on that disk
- there are many standard formats to that table
  - MBR partitions
  - BSD disklabel
  - Solaris VTOC label
  - GUID Partition Table (GPT)
  - others... (Amiga Rigid Disk Block, RDB), (Apple Partition Map, APM)
- linux in the intel\_x86 and amd64 uses MBR and GPT partitions

# The Boot procedure

## → MBR and GPT partitions

# MBR partitions

- the partition is located in the first sector of the disk
- widespread in PC architecture
- used mainly in Windows and Linux systems
- up to 4 partitions, called *primary partitions*, can be defined in a disk

# MBR table format

offset	size	Description
0x000	446	reserved
0x1be	16	partition entry 1
0x1ce	16	partition entry 2
0x1de	16	partition entry 3
0x1ee	16	partition entry 4
0x1fe	2	0xaa55 (little endian)

# MBR partitions

- one of the partitions can be defined as *extended partition*
- this partition can be subdivided into what is called *logical partitions*
- the first sector of that partition, called EBR (Extended Boot Record), has the same format as the MBR table except for
  - only the first two entries are used
  - if more partitions are needed, one of these two is defined as *extended partition*, thus allowing for an "infinite", .i.e. 32 number of partitions

# format of a partion entry

offset	size	Description
0x00	1 byte	80h for active partition, otherwise 00h
0x01	1	head of partition start
0x02	2	cylinder/sector (10/6bits) of partition start
0x04	1	code of partition type
0x05	3	CHS of partition end
0x08	4	LBA partition start
0x0C	4	partition size

## creating MBR partitions

- partitions on disks using the MBR partition scheme are limited to 2 Terabytes
- MBR partitions can be created, and manipulated with the `fdisk` or `cfdisk` utilities on linux systems
- grub usually names them (`hdN,msdosM`)

# GUID Partition Table

- defined as part of the EFI (Extensible Firmware Interface) standard
- sometimes referred to as the *EFI label*, or EFI partition table
- the MBR uses 32 bits for Logical Block Addressing, hence its limitations in size
- GPT uses 64 bits for LBA, this limits the maximum partition size to  $2^{64} - 1$  sectors
- most modern O.S. support GPT although some still have some restrictions to boot from such partitions

# GUID Partition Table

- two copies of the GPT exist, the *primary GPT* at the beginning of the disk, and the *secondary GPT* at the end
- GBT uses logical block addressing
- the first sector of the disk has a MBR partition table called *protective MBR* that allows the disk to be booted from a system with traditional BIOS
- following sector is the header of the primary GPT
- the GPT partition table consists of 128 bytes entries. The minimum size of the table is 16Kbytes

# format of a GPT partion entry

offset	size	Description
0x00	16 bytes	GUID partition type
0x10	16 bytes	Partition GUID
0x20	8 bytes	Partition start LBA
0x28	8 bytes	Partition end LBA
0x30	8 bytes	Attribute flags
0x38	72 bytes	Partition name

# Comparison of MBR and GPT

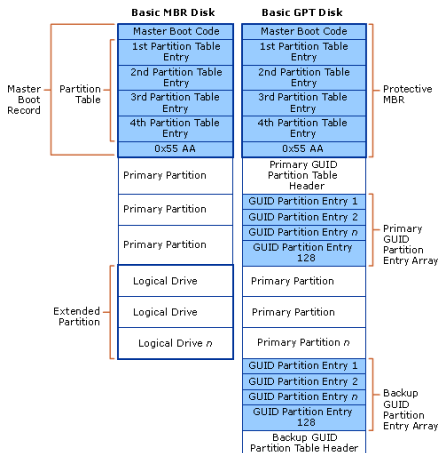


Figure: <https://i-technet.sec.s-msft.com/dynimg/IC197579.gif>

# GPT Partition Table

- to access the GPT on linux the following commands can be used
  - parted and gdisk
  - modern versions of fdisk
- grub usually names them (hdN,gptM)

# Securing the firmware

# Firmware configuration

- Most systems have a small program for configuring (to a certain extent) the firmware
- This program is usually accessed pressing some of the Function keys during POST (Power On Self Test) or pressing a dedicated key
- Among other things, with this program we usually can configure the boot device (or which bootloader to run on UEFI machines)

# Firmware configuration threat: booting from external devices

- Booting from external devices **MUST** be disabled
- If we allow booting from external devices in the firmware
  - anyone with physical access to the machine can boot an installation media or a live system or ...
  - then it can access our files getting information, changing the configuration, installing a backdoor ...

# Firmware configuration threat: accessing the firmware configuration

- We **MUST** install a password to avoid anyone with physical access to the machine change the booting configuration
- Some firmware configuration programs define more than one password
  - password to boot the machine
  - password to change the firmware configuration (usually called *setup*)

# The Grub boot loader

# The Grub boot loader

→ The Grub boot loader

# The Grub boot loader

- **Grand Unified Boot Loader** is the boot loader of choice in linux since several years ago
- It can boot directly linux and other O.S.: freeBSD, OpenBSD, Solaris ... or chainload to other bootloaders
- Highly configurable, can display splash images, a menu ...
- It understands different file systems (ext2fs, ext4fs, ntfs, fat, ufs ...) and different partition types (MBR, gpt ...) through loadable modules
- The boot menu can be edited at boot time and it has a rescue mode command line interpreter capable of accessing filesystems

# The Grub boot loader. Versions

- There are two versions
- Grub version 1, also referred to as Grub legacy
  - Configuration file editable by hand, typically `/boot/grub/menu.lst`
  - Can only boot systems with BIOS type firmware
- Grub version 2, the one that is being installed at present by mostly every linux distro
  - Script generated configuration file (non editable by hand). Typically `/boot/grub/grub.cfg`
  - Can boot both BIOS type and UEFI type firmware
- Both versions provide a boottime-editable boot menu and a rescue mode command line interpreter capable of accessing filesystems

# The Grub boot loader. Basic usage

- to install the grub with BIOS type firmware:

`grub-install device`: Example

```
# grub-install /dev/sda2
```

- to install the grub with UEFI type : `grub-install`

`--efi-directory dir_with_ESP`: Example

```
# grub-install --efi-directory /boot/efi
```

# The Grub boot loader. Basic usage

- To change de grub legacy configuration
  - edit the file `/boot/grub/menu.lst`
- To change de grub version 1 configuration
  - edit the corresponding file in `/etc/grub.d`
  - update the grub configuration file (`/boot/grub/grub.cfg`) with `grub-mkconfig`
    - # `grub-mkconfig -o /boot/grub/grub.cfg`
  - some versions allow for some customization to the configuration file at boot time using `/boot/grub/custom.cfg` or use `update-grub` or `update-grub2`

# Grub Menu



Figure: Grub menu

# The Grub boot loader

## → The Grub boot loader. Vulnerabilities

# The Grub boot loader. Accesing files from the Grub

- The Grub has a comand line interface and can understand filesystems
- We can use the command line interface (accesed when we press 'c' at the boot menu)
- There we can inspect the contents of different partitions (may need to load different modules)
- We can even see the contents on some files

# Grub command line

```
GNU GRUB version 2.02~beta3-5+deb9u1

Minimal BASH-like line editing is supported. For the first word,
TAB lists possible command completions. Anywhere else TAB lists
possible device or file completions. ESC at any time exits.

grub> _
```

Figure: Grub command line

# Accessing passwords with grub

```
grub> set pager=1
grub> cat /etc/shadow
root:$6$IBLrq1Ec$NLxxWst1900ASwYDzo388p7vKEVBuwpzy9BjaHrxoRv34ghX1JkVh/6m1s
gdz87jvudCkhX8ogyyD9U0qsVZ0:17921:0:99999:7:::
daemon*:17921:0:99999:7:::
bin*:17921:0:99999:7:::
sys*:17921:0:99999:7:::
sync*:17921:0:99999:7:::
games*:17921:0:99999:7:::
man*:17921:0:99999:7:::
lp*:17921:0:99999:7:::
mail*:17921:0:99999:7:::
news*:17921:0:99999:7:::
uucp*:17921:0:99999:7:::
proxy*:17921:0:99999:7:::
www-data*:17921:0:99999:7:::
backup*:17921:0:99999:7:::
list*:17921:0:99999:7:::
irc*:17921:0:99999:7:::
gnats*:17921:0:99999:7:::
nobody*:17921:0:99999:7:::
_apt*:17921:0:99999:7:::
rtkit*:17921:0:99999:7:::
avahi-autoipd*:17921:0:99999:7:::
messagebus*:17921:0:99999:7:::
usbmux*:17921:0:99999:7:::
speech-dispatcher:!:17921:0:99999:7:::
--MORE--
```

Figure: Crypted form of the passwords

# Accessing passwords with grub

```
GNU GRUB  version 2.02~beta3-5+deb9u1

insmod part_msdos
insmod ext2
set root='hd0,msdos3'
if [ x${feature_platform_search_hint} = xy ]; then
  search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos3\
--hint-efi=hd0,msdos3 --hint-baremetal=ahci0,msdos3  2eab4ff5-09c9-4df8\
-b945-412bb3bed388
else
  search --no-floppy --fs-uuid --set=root 2eab4ff5-09c9-4df8-b94\
5-412bb3bed388
fi
echo          'Loading Linux 4.9.0-6-amd64 ...'
linux        /boot/vmlinuz-4.9.0-6-amd64 root=UUID=2eab4ff5-09c9\
-4df8-b945-412bb3bed388 ro quiet init=/bin/sh_
echo          'Loading initial ramdisk ...'
```

Minimum Emacs-like screen editing is supported. TAB lists completions. Press Ctrl-x or F10 to boot, Ctrl-c or F2 for a command-line or ESC to discard edits and return to the GRUB menu.

Figure: Editing one Grub Menu item

# The Grub boot loader. Booting single user mode

- We can modify Grub menu lines at boot time (pressing 'e' at the menu)
- With the adequate parameters we can change the menu items so that when we boot we boot in single user mode without being prompted with a password
- Now we can access the whole system with administrator privileges

# Securing the Grub boot Loader

# Securing the grub boot loader

- we want to disable access to grub functionality that allows us to view files or to modify menu items
- in grub2 we can define the variable *superusers* as a list of names of users who have access to the command line mode or editing mode
  - these need not be system's actual user names, just names for the grub to identify by
- if the variable is defined but left empty, no one will have access to the command line or editing mode.

# Securing the Grub boot Loader

## → Grub users and superusers

# Grub users and superusers

- in grub2 we can also define the variable *users* as a list of names of users who have access to the corresponding menuentries
  - again these need not be system's actual user names, just names for the grub to identify by
  - a menuentry is usable by an user if stated so when defining the menuentry

```
menuentry "Only to be booted by a superuser or user1" --users user1 {  
    set root=(hd0,2)
```

- menuentries can be used by anyone if `--unrestricted` is specified

```
menuentry "Anyone can boot this" --unrestricted {  
    set root=(hd0,2)
```

- Menuentries without the `--unrestricted` or `-user` can only be used by a superuser

# Securing the Grub boot Loader

## → Grub Passwords

# Grub passwords

- passwords for users and superusers can be set in the grub configuration file with the command *passwd*
- if we do not want the password be in clear text in this configuration file *very reasonable!!* we can use the *password\_pbkdf2* comand
- to generate this form of the passwd we can use the *grub-mkpasswd-pbkdf2* command

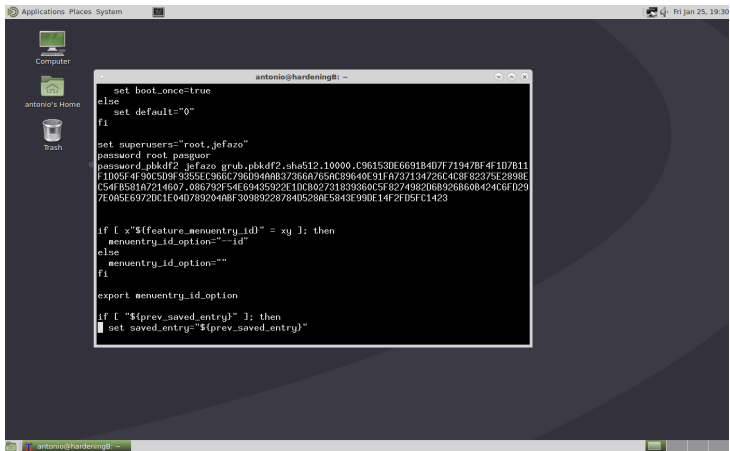
```
# grub-mkpasswd-pbkdf2
```

```
Enter password:
```

```
Reenter password:
```

```
PBKDF2 hash of your password is grub.pbkdf2.sha512.10000.ACE955DF7381D3E5E2C4E4C27302FF6  
7ADD3A4FF0F40CD71739634DF926C301C1E829287D493AF9801D3B80754FA56D47837225D3218CBFFD5FCFB1  
66AC039B1.7BD5B5A1AE2D19B4723E0E4A8ACBC551FB56A969EF2FF5772DC695B9C54BC064872BD4BFEB868D  
6CD598C2BDFDEF3A59681BE405180063CB78ED603C82621494
```

# Accessing passwords with grub



The screenshot shows a Linux desktop environment with a dark theme. In the background, there are icons for 'Computer', 'antonio's Home', and 'Trash'. A terminal window titled 'antonio@hardening8: ~' is open, displaying the contents of a sample `grub.cfg` file. The file includes password settings for the root user and conditional menuentry options.

```
antonio@hardening8: ~  
set boot_once=true  
else  
  set default="0"  
fi  
  
set superusers="root,jefazo"  
password root pasguor  
password_pbkdf2 jefazo grub.pbkdf2.sha512.10000.C96153DE6691B4D7F71947BF4F1D7B11  
F1D05F4F90C509F9355EC966C796D94AAB37366A765AC89640E91FA737134726C4C8F82375E2898E  
C54FB581A7214607.086792F54E69435922C1DCB02731839360C5F8274982068926860B424C6FD29  
7E0A5E6972DC1E04D789204ABF30989228784D528AE5843E99DE14F2FD5FC1423  
  
if [ x"${feature_menuentry_id}" = xy ]; then  
  menuentry_id_option="--id"  
else  
  menuentry_id_option=""  
fi  
  
export menuentry_id_option  
  
if [ "${prev_saved_entry}" ]; then  
  set saved_entry="${prev_saved_entry}"
```

Figure: Sample grub.cfg with passwords

# Notes on grub authentication

- superusers, users and passwords must not be defined directly in the *grub.cfg* file as this file will be overwritten the next time grub configuration is updated
- they should be defined in one of the files in */etc/grub.d*
- as of now, *grub-mkconfig* doesn't handle authentication, so if we want any menu entry to be unrestricted or usable by users we must
  - create a corresponding menuentry in one of the files in */etc/grub.d*
  - modify the *grub.cfg* file by hand (and expect to remodify it each time *grub-mkconfig* is run)

## Other boot loaders

# Other boot loaders

- although grub is most widespread bootloader in the linux world, its not the only one
- there are others much simpler: syslinux (typically used for removable FAT formatted media) lilo, elilo ...
- we'll comment briefly the lilo and elilo bootloaders

# LILO boot loader

- Only boots with BIOS type firmware
- Can chainload to other boot loaders
- Its configuration resides by default in the file `/etc/lilo.conf`
- It doesn't understand any filesystem so it cannot read its configuration: the program `/sbin/lilo` must be run after changing the configuration file to update its internal tables
- a password can be set for each entry with the command *password* in its configuration file
- with *restricted* we require the password not to boot, but to append kernel parameters

# Sample LILO configuration file

- A password (*win*) is required to boot windows
- A password (*pasguor*) is required to append parameters to the linux kernel
- The default image (Linux) can boot with no password

```
image = /boot/vmlinuz-4.9.0-6-amd64
    label = "Linux"
    read-only
    password=pasguor
    restricted
    initrd = /boot/initrd.img-4.9.0-6-amd64

other=/dev/sda1
    password=win
    label=windows
```

- if the password is set to the empty string, it will be prompted when running `/sbin/lilo`

# ELILO boot loader

- Only boots with UEFI type firmware
- Only understands FAT filesystem, so the loader (*elilo.efi*), its configuration file (*elilo.conf*) the kernel and the *initrd* must reside in the EFI System partition
- Its configuration file resembles that of LILO

# others

- There are other boot loaders for linux on the amd64 architecture, depending on the firmware type
- *rEFInd* *syslinux-efi* and *systemd-boot* for UEFI type firmware
- *syslinux*, *em* *loadlin* for BIOS type firmware