liveng Documentation

Release 1.0

LumIT Labs

Feb 04, 2019

Contents

Next Generation Linux live distributions concepts

A live operating system allows booting from a removable medium, such a USB key, without the need of being installed to the hard drive.

CHAPTER 1

Why liveng

None of the existing ISO9660-based live operating systems provide a **kernel update feature**: the kernel and the initrd are the only components that a live operating system cannot update, because they lay outside of the data persistence partition and the system partition is ISO9660-formatted, so not writeable - which is the best option for a live, because of its strength against data corruption.

This will soon lead to an outdated operating system, particularly unsafe if used as a desktop-replacement or for security-critical activities.

CHAPTER 2

More features

Once written onto a USB key, a common live operating system is usually made up of one ISO9660 partition, containing the kernel, the initrd, the compressed filesystem.squashfs image and the second stage bootloader, usually *isolinux* (the boot sector code linking the second stage bootloader is contained within the MBR of the key). Modern lives also add a UEFI partition (some add a "fake" one).

If you need a live system which does data persistence, you will find (or need to create) another partition, usually an EXT4 one. This is pretty common as well.

There are only a few live distibutions which support the UEFI Secure Boot (Debian lives do not), and, as stated before, no distribution is capable of updating the kernel maintaining a ISO9660 filesystem.

The full aim of the liveng project is to give the Community a set of best practices in order to turn a common Debian Linux live into a live(ng) operating system which features:

- native **encrypted persistence**;
- **kernel update (on a live ISO 9660 filesystem)**;
- UEFI, with **UEFI Secure Boot compatibility**, with a real efi partition.

As the base of liveng we have chosen the Debian Stretch live distribution.

liveng is a LumIT Labs project.

CHAPTER 3

Table of contents

3.1 Standard Debian live

A live operating system allows booting from a removable medium, such a USB key, without the need of being installed to the hard drive.

3.1.1 How a standard Debian live system is made

Main components of a (Linux Debian) live operating system are:

- The Linux kernel image, usually named vmlinuz.
- The initial RAM disk image (initrd): a RAM disk containing all the modules needed to mount the root filesystem and some scripts and binaries to perform the task. *initramfs-tools* is responsible of creating such an image when updating the kernel or adding some modules to the system.
- The filesystem image containing all the operating system's files (kernel and initrd as well). Usually, a SquashFS compressed filesystem is used to minimize the Debian live image size.
- The bootloader: a small piece of code crafted to boot from the chosen medium. It loads the Linux kernel and its initrd to run with an associated filesystem.

Upon booting, for BIOS-based hardware, the computer's firmware will locate and load the first-stage bootloader (within the MBR), which in turn loads its second-stage, resposible for loading the kernel and the associated initrd into memory; then the bootloader passes the execution control to the kernel. For UEFI-based hardware, there is only a binary executable bootloader file located in a small FAT partition.

After some basic initializations, the kernel extracts the initrd archive and mounts it as a temporary root filesystem.

The initrd contains kernel modules and userspace programs (such as the tool to install kernel modules into the kernel) required to initialize the device(s) containing the real root filesystem.

The init script on the initrd loads modules and performs other neccessary steps, such as union-mounting (by the use of unionfs) the compressed filesystem image with a (possibly LUKS encrypted) persistence partition or with the RAM disk.

At the end of this stage the initrd is deleted from the memory, and the real root filesystem is mounted. Finally the kernel passes the control to the /sbin/init (Systemd) program on it.

With such setup the kernel size is kept under control by allowing most of the drivers to be compiled as modules - in an initrd-less setup the drivers neccessary for the boot-time initialization of the root device must be compiled into it.

3.2 Set up a standard Debian live

All the steps needed to write a standard Debian Stretch live operating system image onto a USB key briefely follow.

- download the iso-hybrid live image from: https://cdimage.debian.org/debian-cd/current-live/ for your PC's architecture, i386 or amd64;
- plug the USB key into your computer running any Linux-based OS (I am using Debian Stretch for this guide as my host system);
- open a terminal emulator and get root privileges;
- use `fdisk -l` in order to locate the USB key's device file, for example: */dev/sdx* (change for your case). A sample output on my PC:

```
Disk /dev/sdc: 29,4 GiB, 31608274944 bytes, 61734912 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xd8915d07

Device     Start      End  Sectors  Size  Id Type
/dev/sdc1   2048 61734911 61732864  29,4G b  W95 FAT32
```

- unmount the auto-mounted devices, if any (desktop environments usually have an auto-mount feature). You can see them via the `mount` command and unmount them by using `umount /dev/sdxN` for each device file;
- finally write the image with dd; in my case the command is:

```
dd if=/home/marco/Downloads/debian-live-9.5.0-amd64-gnome.iso of=/dev/sdx bs=50M
```

Be careful or you will overwrite your hard drive content! Here I replaced */dev/sdc* with */dev/sdx* to avoid blind copies and pastes. Setting the bs command line option to 10MB (or more), speeds up the process:

```
2365587456 bytes (2,4 GB, 2,2 GiB) copied, 31,882 s, 74,2 MB/s
```

Please note that you must refer to the whole disk, so do not use */dev/sdx1* but */dev/sdx*, as an example.

Resulting partitioning scheme:

```
Disk /dev/sdc: 29,4 GiB, 31608274944 bytes, 61734912 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x1dc1a6b1

Device      Boot Start     End Sectors  Size Id Type
/dev/sdc1   *        0 4620287 4620288  2,2G  0 Empty
/dev/sdc2         1416    2247     832  416K ef EFI (FAT-12/16/32)
```

Being Stretch UEFI compliant and not only BIOS compatible, you will notice two partitions out of the box here, one of which is a fake partition (an overlapping one) and to us not a good way of solving problems. Used technique is called *isohybrid feature for UEFI*.

> The isohybrid feature for UEFI adds a partition to the MBR partition table pointing to the same file in the ISO 9660 filesystem (efiboot.img) as does the El Torito catalog entry for EFI. This file contains a FAT filesystem with boot equipment from which the UEFI firmware will be able to start the desired operating system. Furthermore, isohybrid for UEFI creates a GUID Partition Table with a partition pointing to that file.

Your USB key can now boot your PC if set as the boot device and UEFI Secure Boot is disabled on the PC's firmware. Older PCs (pre-Windows 10) does not have the Secure Boot feature and will boot anyway, if supported.

You can now enjoy a stock Debian live operating system with the desktop environment of your choice, without any data persistence: upon booting, the (live-build modified) initrd will union-mount the compressed filesystem with the RAM disk and pass that mount to the Linux kernel as / (the root). Reboot the system and every data will be lost.

3.3 liveng structure

We have previously deployed a stock Debian live with the use of dd, preserving all the structures contained within the downloaded ISO image, resulting in a live operating system without proper UEFI or Secure Boot support and without the wows of a kernel update (on a ISO9660 filesystem) feature.

In order to build a liveng operating system, we must now restart from scratch, however, and keep only a bunch of the old files - only the the **kernel**, the **initrd** and the **filesystem.squashfs** are really needed.

For future reference, the device file corresponding to the whole USB key (please plug the USB key in and use `fdisk -l` as stated previously) and the path of the downloaded Debian live image are kept in two variables, together with the size of the image file itself in a third one:

```
device="/dev/sdx"
imageFile="/path/to/debian-live-x.y.z-arch-desktop.iso"
imageFileSize=$(du -m ${imageFile} | awk '{print $1}')
```

As done before, I'm trying avoiding some hard drive wiping (and some heart attacks) by changing the device file name to */dev/sdx* for the whole liveng documentation, please remember to change it for your case.

We are also tracking the name of the files we'll need, which are contained inside the ISO image:

```
mount ${imageFile} /mnt

configFile=$(ls /mnt/live/ | awk '{print $1}' | grep config)
initrdFile=$(ls /mnt/live/ | awk '{print $1}' | grep initrd)
vmlinuzFile=$(ls /mnt/live/ | awk '{print $1}' | grep vmlinuz)
systemMapFile=$(ls /mnt/live/ | awk '{print $1}' | grep System)

umount /mnt
```

We will unomunt now every possible mountpoint that you or your desktop environment may have enstablished with the key's partitions, and wipe all the contained filesystems' key structures:

```
cd /tmp

# Unmount.
for i in $(mount | grep ${device} | awk '{print $1}'); do umount $i; done

```

(continues on next page)

(continued from previous page)

```
# Wipe.
wipefs -af ${device}
```

3.3.1 GUID partition table

Now we are creating a new empty GUID partition table (GPT for its friends); this operation deletes all the exisiting partitions and creates a new protective MBR (for backward compatibility with BIOS hardware):

```
printf "o\nY\nw\nY\n" | gdisk ${device} && sync
```

You can launch `gdisk ${device}` and type all the options by hand in order to see what the previous command actually does.

Resulting partitioning scheme (with `fdisk -l`):

```
Disk /dev/sdb: 29,4 GiB, 31608274944 bytes, 61734912 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 1D0FE347-D269-4E57-843C-AA5DAEA2A7A1
```

No partitions available so far. So good.

3.3.2 System partition

Now it's the turn of manually creating the first partition onto the USB key; we will always use gdisk, for the older fdisk is not GPT compliant. The first partition will host most of the files contained within the downloaded Debian live image, so we are creating it of the same size as the image, which is consistent with our scope:

```
printf "n\n\n\n\n+${imageFileSize}M\n8300\nw\nY\n" | gdisk ${device} && sync
```

Previous line means:

```
Command (? for help): n
Partition number (1-128, default 1):
First sector (34-61734878, default = 2048) or {+-}size{KMGTP}:
Last sector (2048-61734878, default = 61734878) or {+-}size{KMGTP}: +2300M
Current type is 'Linux filesystem'
Hex code or GUID (L to show codes, Enter = 8300):
Changed type of partition to 'Linux filesystem'

Command (? for help): w
Final checks complete. About to write GPT data. THIS WILL OVERWRITE EXISTING
PARTITIONS!!
Do you want to proceed? (Y/N): Y
OK; writing new GUID partition table (GPT) to /dev/sdb.
The operation has completed successfully.
```

Resulting partitioning scheme (with `fdisk -l`):

```
Disk /dev/sdb: 29,4 GiB, 31608274944 bytes, 61734912 sectors
Units: sectors of 1 * 512 = 512 bytes
```

(continues on next page)

(continued from previous page)

```
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 1D0FE347-D269-4E57-843C-AA5DAEA2A7A1

Device      Start     End Sectors  Size Type
/dev/sdb1    2048 4712447 4710400  2,3G Linux filesystem
```

One unformatted partition has been created.

With the next step, xorriso will help us writing the files to the new newely created USB's partition, maintaining the ISO9660 filesystem (rtfm of xorriso for more):

```
xorriso -indev ${imageFile} -boot_image any discard -overwrite on -volid LIVENG_
↪SYSTEM -move live/${configFile} live/config -move live/${initrdFile} live/initrd.
↪img -move live/${vmlinuzFile} live/vmlinuz -move live/${systemMapFile} live/System.
↪map -rm_r .disk boot d-i dists isolinux pool -- -outdev stdio:${device}1 -blank as_
↪needed
```

As the result:

```
ISO image produced: 1081069 sectors
Written to medium : 1081248 sectors at LBA 32
Writing to 'stdio:/dev/sdb1' completed successfully.
```

Having a look that everything is as expected:

```
mount ${device}1 /mnt/
ls /mnt/live/
```

Will display:

```
config  filesystem.squashfs  initrd.img  System.map  vmlinuz
```

As you may have noted, all the files have been renamed, in order for us to ease the configuration of GRUB. Now we can unmount the partition:

```
umount /mnt
```

3.3.3 Second system partition

With the same technique used before, we are now adding a second system partition to the key, which will contain the **kernel and the initrd files only**. The bootloader will be then instructed to **boot from this partition**, because the second system partition will always contain the **most updated kernel and initrd** files.

At every kernel update, this small partition will be overwritten, with the use of xorriso, by the postinst script of the liveng kernel package. (First) System partition files are kept at their default state and can be useful in case of recovery or when a complete persistence reset is performed.

Creating the second partition:

```
printf "n\n\n\n\n+256M\n8300\nw\nY\n" | gdisk ${device}
```

`fdisk -l`:

```
Device        Start     End Sectors  Size Type
/dev/sdc1      2048 4624383 4622336  2,2G Linux filesystem
/dev/sdc2   4624384 5148671  524288  256M Linux filesystem
```

With the next step, xorriso will write the kernel and initrd files into the second system partition (ISO9660):

```
xorriso -indev ${imageFile} -boot_image any discard -overwrite on -volid 'LIVENG_
↪SYSTEM2' -move live/${configFile} live/config -move live/${initrdFile} live/initrd.
↪img -move live/${vmlinuzFile} live/vmlinuz -move live/${systemMapFile} live/System.
↪map -rm_r .disk boot d-i dists isolinux pool live/filesystem.squashfs -- -outdev␣
↪stdio:${device}2 -blank as_needed
```

Having a look that everything is as expected:

```
mount ${device}2 /mnt/
ls /mnt/live/
umount /mnt
```

Will display:

```
config    initrd.img    System.map    vmlinuz
```

3.3.4 GRUB bootloader (UEFI)

The GPT partitioning scheme has been set up and the system partitions have been created. We are now adding the UEFI partition, which must be FAT (guess who introduced the UEFI Secure Boot. . .).

Every standard UEFI firmware must look into the /efi/boot/ folder of the FAT partition of the booting device for a file named *boot{arch}.efi*, so we are just creating the folder on the USB drive and copy a GRUB UEFI-compliant bootloader binary to this location.

But first we need to generate our own GRUB bootloader image with some modules included. The following command will generate the GRUB image. *grub-efi-amd64** amd64 Debian packages are needed - novices are advised to download the *bootx64.efi* hosted at liveng Github repository (**docs/grub-bin/**) instead of creating a new one:

```
grub-mkimage -o bootx64.efi -p /efi/boot -O x86_64-efi \
    fat iso9660 part_gpt part_msdos \
    normal boot linux configfile loopback chain keylayouts \
    efifwsetup efi_gop efi_uga jpeg png \
    ls search search_label search_fs_uuid search_fs_file \
    gfxterm gfxterm_background gfxterm_menu test all_video loadenv \
    exfat ext2 ntfs udf password password_pbkdf2 pbkdf2 linuxefi
```

We are ready to create the third (UEFI, FAT) partition, sized 32MB, onto the key:

```
printf "n\n\n\n+32M\nef00\nw\nY\n" | gdisk ${device} && sync
mkfs.vfat -n "UEFI Boot" ${device}3
```

Resulting partitioning scheme (with `fdisk -l`):

```
Device        Start     End Sectors  Size Type
/dev/sdc1      2048 4624383 4622336  2,2G Linux filesystem
/dev/sdc2   4624384 5148671  524288  256M Linux filesystem
/dev/sdc3   5148672 5214207   65536   32M EFI System
```

Now we will copy the GRUB binary to the UEFI partition, as stated before:

```
mount ${device}3 /mnt/
mkdir -p /mnt/efi/boot
cp /path/to/bootx64.efi /mnt/efi/boot
```

and setup the GRUB config file so that GRUB will be able to locate the kernel, initrd and filesystem.squashfs files upon booting. Kernel and initrd files will be loaded from the second system partition and **the filesystem.squashfs will be loaded from the first system partition** (thanks to the *fromiso* live-boot's directive):

```
    cat > /mnt/efi/boot/grub.cfg <<EOF
menuentry 'liveng standard boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt2 --fs-uuid $(blkid -s UUID $
↪{device}2 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence
    initrd /live/initrd.img
}
EOF
```

We also add a **fallback boot**, which loads all the files from the original (first) system partition, just in case something goes wrong:

```
    cat >> /mnt/efi/boot/grub.cfg <<EOF
menuentry 'liveng fallback boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt1 --fs-uuid $(blkid -s UUID $
↪{device}1 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence liveng-fallback
    initrd /live/initrd.img
}
EOF

umount /mnt
```

The `$(blkid -s UUID ${device}x | awk -F\" '{print $2}')` command gets the UUID of the *${device}x* partition, given during the partition creation.

We will use the `persistence` directive later on; if no persistence partition is found, the (live-boot modified) initrd will union-mount the filesystem.squashfs with the RAM disk. The `liveng-fallback` will be of use later on, too.

3.3.5 GRUB bootloader (BIOS)

The GPT partitioning scheme we previously set up also contains a protective MBR, which assures BIOS compatibility and the possibility to create more than 4 primary partitions.

We will now set up GRUB for the BIOS compatibility - as said, unlike UEFI, BIOS needs a first-stage bootloader (the boot sector code) contained within the MBR of the key:

```
mount ${device}3 /mnt/
cp -R /path/to/grub-bios /mnt/boot
```

Where *grub-bios* is a folder containing "standard" GRUB binaries for BIOS. You can download a compressed archive of it from the liveng Github repository (**docs/grub-bin/**).

grub.cfg:

```
    cat > /mnt/boot/grub/grub.cfg <<EOF
menuentry 'liveng standard boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt2 --fs-uuid $(blkid -s UUID $
↪{device}2 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence
    initrd /live/initrd.img
}
EOF
```

We also add a **fallback boot**, which loads all the files from the original (first) system partition, just in case something goes wrong:

```
    cat >> /mnt/boot/grub/grub.cfg <<EOF
menuentry 'liveng fallback boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt1 --fs-uuid $(blkid -s UUID $
↪{device}1 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence liveng-fallback
    initrd /live/initrd.img
}
EOF
```

Finally, install first-stage GRUB:

```
grub-install --root-directory=/mnt ${device} --force

umount /mnt
```

3.3.6 So far, so good

So far, we have set up a USB key with a GPT and protective MBR partitioning scheme, with three partitions, two for the system and one for the UEFI compliance. We also used and installed GRUB as the bootloader instead of syslinux/isolinux - this will give us much more flexibility.

We will go back to kernel update later on.

Live system is still non-persistent, please do a couple of UEFI (non-Secure Boot) and BIOS test bootings to verify all is set up correctly…

3.4 UEFI Secure Boot

We previously deployed a USB key with a GPT and protective MBR partitioning scheme, with three partitions, two for the system and one for the UEFI compliance. We also used and installed GRUB as the bootloader instead of syslinux/isolinux.

While UEFI compatibility is assured by the efi FAT partition and by the GRUB binary image inside it, Secure Boot machines won't natively run the live system, because there is no signed bootloader yet.

In brief, while UEFI Secure Boot is enabled, the hardware restricts the execution of code at boot time only to binary code signed by trusted keys. Trusted keys belong to a chain of trust with a root key stored on firmware and referred to as the Platform Key. The UEFI firmware verifies that the boot loader image has a cryptographically valid signature before running it.

Easiest way to *bypass the problem* is to use the Linux Foundation's preloader.

Idea is simple: the preloader binary is signed with a (Microsoft) key and it is therefore suitable for being loaded by every UEFI Secure Boot firmware.

When run, the preloader tries to launch *loader.efi*, a chained executable, which will be the "real" bootloader - GRUB in our case.

Preloader uses a whitelist mechanism called Machine Owner Key list. If the hash of the launching binary is in the MokList, the preloader will execute it, if not, it will launch a key management utility (the HashTool) which allows enrolling the hash or key by the user:

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                            Failed to start loader                            │
│                                                                              │
│            It should be called loader.efi (in the current directory)         │
│                      Please enrol its hash and try again                     │
│                                                                              │
│                  I will now execute HashTool for you to do this              │
│                                                                              │
│                                                                              │
│                                    ┌────┐                                    │
│                                    │ OK │                                    │
│                                    └────┘                                    │
│                                                                              │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────────────┐
│                                Select Binary                                 │
│                                                                              │
│                The Selected Binary will have its hash Enrolled               │
│             This means it will Subsequently Boot with no prompting           │
│     Remember to make sure it is a genuine binary before Enroling its hash    │
│                                                                              │
│                                                                              │
│                             ┌─────────────────────┐                          │
│                             │     Enroll Hash     │                          │
│                             │ Reboot to UEFI Menu │                          │
│                             │    Reboot System    │                          │
│                             │         Exit        │                          │
│                             └─────────────────────┘                          │
│                                                                              │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────────────┐
│                                Select Binary                                 │
│                                                                              │
│                The Selected Binary will have its hash Enrolled               │
│             This means it will Subsequently Boot with no prompting           │
│     Remember to make sure it is a genuine binary before Enroling its hash    │
│                                                                              │
│                                                                              │
│                                 ┌─────────────┐                              │
│                                 │     ../     │                              │
│                                 │  loader.efi │                              │
│                                 │ HashTool.efi│                              │
│                                 │ bootx64.efi │                              │
```

(continues on next page)

(continued from previous page)

```
│                                  └───────────────                             │
│                                                                                │
└────────────────────────────────────────────────────────────────────────────────┘

───────────────────────────────────────────────────────────────────────────────
│                       Enroll this hash into MOK database?                      │
│                                                                                │
│                              File: \loader.efi                                 │
│     Hash: 8D1B74227CB2EE6B23B829595B761BAA34D171337F70D44ABF542D5318BDBA08      │
│                                                                                │
│                                                                                │
│                                                                                │
│                                     ─────                                     │
│                                    │ No  │                                     │
│                                    │ Yes │                                     │
│                                    └─────                                     │
│                                                                                │
└────────────────────────────────────────────────────────────────────────────────┘
```

3.4.1 In practice

Setting up UEFI Secure Boot compatibility is now easy. We download the Linux Foundation preloader and HashTool, rename bootx64.efi as loader.efi, then move the Linux Foundation's files into the efi folder of the USB key (efi partition):

```
cd /tmp
wget https://blog.hansenpartnership.com/wp-uploads/2013/PreLoader.efi
wget https://blog.hansenpartnership.com/wp-uploads/2013/HashTool.efi

mount ${device}3 /mnt/
mv /mnt/efi/boot/bootx64.efi /mnt/efi/boot/loader.efi

mv PreLoader.efi /mnt/efi/boot/bootx64.efi
mv HashTool.efi /mnt/efi/boot/

umount /mnt
```

liveng system is now capable of UEFI Secure Boot the clean way.

3.5 Persistence

Once setup a USB key with a GPT and protective MBR partitioning scheme, with three partitions, two for the system (kernel, initrd, filesystem.squashfs and kernel, initrd respectively) and one for the UEFI/UEFI Secure Boot compliance with the use of GRUB, we will now make it data-persistent capable.

Data persistence is accomplished by a fourth partition on the USB key where system is instructed to write into, or better explained, which the system will union-mount with the filesystem.squashfs image.

Kernel boot parameter *persistence* tells the live-boot modified initrd's /lib/live/* scripts to search (by default) for a device partition named *persistence* and with a config file inside, named *persistence.conf*. You can see /lib/live/* scripts' content by decompressing an initrd image (it's a cpio compressed image).

Because we already put the *persistence* parameter witin the list of the kernel boot parameters:

```
linux /live/vmlinuz initrd=/live/initrd.img fromiso=2018-10-01-10-22-29-00 boot=live␣
↪live-noconfig **persistence**
```

we now just need to create a fourth partition on the USB key (as big as the space left on the key), labelled *persistence*:

```
printf "n\n\n\n\n\nw\nY\n" | gdisk ${device} && sync
mkfs.ext4 -F -L "persistence" ${device}4
```

and to add the *persistence.conf* file inside:

```
mount ${device}4 /mnt
echo "/ union" > /mnt/persistence.conf
umount /mnt
```

With `/ union` we instruct the initrd to union-mount this partition for the whole filesystem.

> Encrypted persistence for a liveng system with a stock Debian live image as the base is not as straightforward as you might think and requires a lot of work. However, a persistent liveng operating system without encryption is as well suitable for demonstrating the kernel update on a ISO9660 filesystem, which after all is the most important liveng feature. So, if primarily interested in the kernel update, you can skip the next sections of this chapter.

3.5.1 Encrypted persistence

Kernel boot parameter *persistence-encryption=luks* tells the initrd to search for a LUKS-encrypted device partition named *persistence* for the union-mount. The initrd's /lib/live/* scripts will ask a user for the decryption password during the boot and will decrypt the partition with cryptsetup.

First of all, we have to add the parameter to the list of kernel boot parameters in the grub.cfg's:

```
mount ${device}3 /mnt/

    cat > /mnt/efi/boot/grub.cfg <<EOF
menuentry 'liveng standard boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt2 --fs-uuid $(blkid -s UUID $
↪{device}2 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence persistence-
↪encryption=luks
    initrd /live/initrd.img
}
EOF

    cat >> /mnt/efi/boot/grub.cfg <<EOF
menuentry 'liveng fallback boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt1 --fs-uuid $(blkid -s UUID $
↪{device}1 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence persistence-
↪encryption=luks liveng-fallback
    initrd /live/initrd.img
}
EOF
```

(continues on next page)

(continued from previous page)

```
    cat > /mnt/boot/grub/grub.cfg <<EOF
menuentry 'liveng standard boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt2 --fs-uuid $(blkid -s UUID $
↪{device}2 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence persistence-
↪encryption=luks
    initrd /live/initrd.img
}
EOF

    cat >> /mnt/boot/grub/grub.cfg <<EOF
menuentry 'liveng fallback boot' --unrestricted {
    insmod iso9660
    search --no-floppy --set=root --hint-efi=hd0,gpt1 --fs-uuid $(blkid -s UUID $
↪{device}1 | awk -F\" '{print $2}')
    linux /live/vmlinuz initrd=/live/initrd.img fromiso=$(blkid -s UUID ${device}1 |␣
↪awk -F\" '{print $2}') boot=live live-noconfig persistence persistence-
↪encryption=luks liveng-fallback
    initrd /live/initrd.img
}
EOF

umount /mnt
```

Then we must initialize the fourth partition as a LUKS one, format and label it (of course, we are overwriting what we've just done the step before):

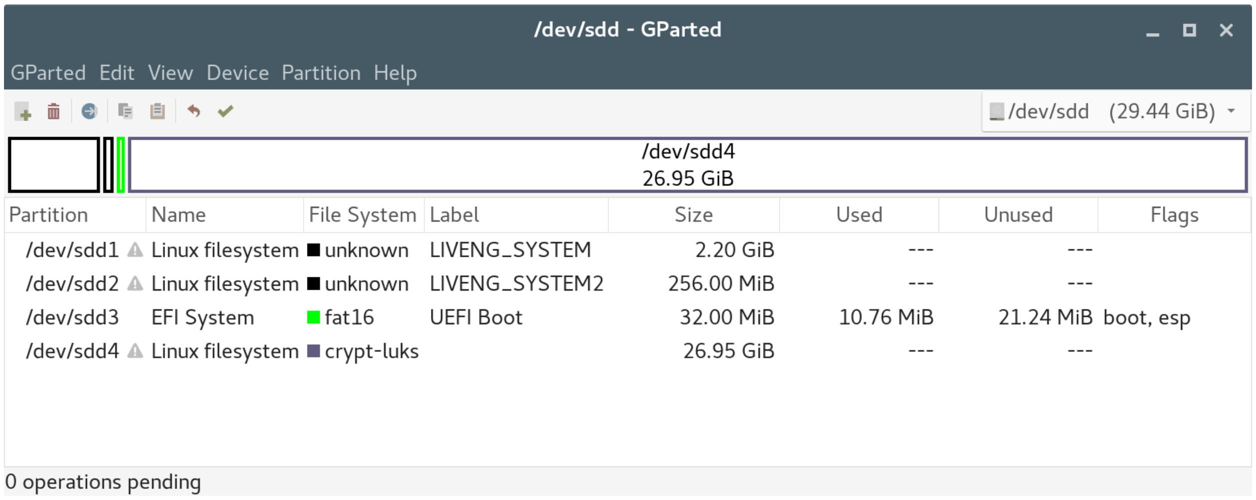
```
# Recreate partition.
printf "d\n4\nw\nY\n" | gdisk ${device} && sync
printf "n\n\n\n\n\n\nw\nY\n" | gdisk ${device} && sync
mkfs.ext4 -F -L "persistence" ${device}4

# Initialize a LUKS ext4 partition.
echo -n "PASSWORD" | cryptsetup --hash=sha512 --cipher=aes-xts-plain64 --key-size=512␣
↪luksFormat ${device}4 -
echo -n "PASSWORD" | cryptsetup luksOpen ${device}4 encrypted -

mkfs.ext4 /dev/mapper/encrypted && sync
e2label /dev/mapper/encrypted persistence && sync

/sbin/cryptsetup luksClose /dev/mapper/encrypted
```

Previous lines initialize a LUKS partition with a random master key and with *PASSWORD* as the first-slot password which encrypts the master key.



Partition	Name	File System	Label	Size	Used	Unused	Flags
/dev/sdd1	Linux filesystem	■ unknown	LIVENG_SYSTEM	2.20 GiB	---	---	
/dev/sdd2	Linux filesystem	■ unknown	LIVENG_SYSTEM2	256.00 MiB	---	---	
/dev/sdd3	EFI System	🟩 fat16	UEFI Boot	32.00 MiB	10.76 MiB	21.24 MiB	boot, esp
/dev/sdd4	Linux filesystem	■ crypt-luks		26.95 GiB	---	---	

0 operations pending

Finally, we must put the *persistence.conf* file inside it:

```
echo -n "PASSWORD" | cryptsetup luksOpen ${device}4 encrypted -
mount -t auto /dev/mapper/encrypted /mnt

echo "/ union" > /mnt/persistence.conf

umount /mnt
cryptsetup luksClose /dev/mapper/encrypted
```

liveng should now be persistent, but unfortunately **the stock Debian live initrd image misses cryptsetup binaries and libraries** (bacause it is not *live-built* for this purpose). By opening the cpio archive you can notice that a lot of the following needed files are missing:

```
./lib/x86_64-linux-gnu/libcryptsetup.so.4.7.0
./lib/x86_64-linux-gnu/libgcrypt.so.20.1.6
./lib/x86_64-linux-gnu/libcryptsetup.so.4
./lib/x86_64-linux-gnu/libgcrypt.so.20
./lib/cryptsetup
./lib/modules/4.9.0-7-amd64/kernel/crypto
./lib/modules/4.9.0-7-amd64/kernel/crypto/cryptd.ko
./lib/modules/4.9.0-7-amd64/kernel/fs/crypto
./lib/modules/4.9.0-7-amd64/kernel/fs/crypto/fscrypto.ko
./lib/modules/4.9.0-7-amd64/kernel/arch/x86/crypto
./lib/modules/4.9.0-7-amd64/kernel/drivers/crypto
./lib/modules/4.9.0-7-amd64/kernel/drivers/md/dm-crypt.ko
./sbin/cryptsetup
```

Example is at the time of writing, late 2018.

We have to add them: fasten your seat belts. Please note that the following steps must be done because the stock Debian live images lack cryptsetup presence into the initrd; the steps below are described for completeness: starting from a Debian-derivative live image which already has cryptsetup built in the initrd will simplify the liveng transoformation.

3.5.2 Adding cryptsetup to liveng: create a modified initrd

Fastest way to add those files into the initrd image of our liveng is to boot and run the liveng with cleartext persistence we obtained the step before *"Encrypted persistence"*, and launch an initrd update after having installed cryptsetup into the system and having enabled the ad-hoc initramfs-hook.

So, boot the liveng image, get a terminal emulator as root and install cryptsetup:

> apt-get install cryptsetup

then modify */etc/cryptsetup-initramfs/conf-hook* in order to enable the initramfs hook (it will instruct initramfs-tools what to do when updating the initrd):

```
sed -i 's/#CRYPTSETUP=/CRYPTSETUP=y/g' /etc/cryptsetup-initramfs/conf-hook
```

then manually trigger the initrd update (update is by default disabled, being liveng a live operating system with a ISO filesystem):

```
mkinitramfs -o $(ls /boot | grep initrd)
```

Finally copy the *initrd.img-** from the current directory to an external USB key, power off the computer and go back to the host system where forging liveng.

> Note: when modifying the initrd image, we must also modify the filesystem.squashfs content adding the cryptsetup binary and, most important, enabling the initramfs hook in */etc/cryptsetup-initramfs/conf-hook*, in order to keep the cryptsetup binaries in the initrd on future kernel updates. This step is not covered here.

3.5.3 Replacing stock image's initrd

We now need to replace the initrd into the original Debian live image with the new one: we will accomplish this task with the use of xorriso in three discrete steps.

Remove ye old initrd.img-version from the stock Debian live image and save a temporary image as the result:

```
xorriso -indev ${imageFile} -boot_image any discard -overwrite on -volid LIVENG_
↪SYSTEM -rm_r live/initrd.img-4.9.0-7-amd64 -- -outdev ${imageFile}1 -blank as_needed
```

Add the initrd.img-version previously built:

```
xorriso -indev ${imageFile}1 -boot_image any discard -overwrite on -volid LIVENG_
↪SYSTEM -add initrd.img-4.9.0-7-amd64 -- -outdev ${imageFile}2 -blank as_needed
```

Move it into the internel *live/* folder:

```
xorriso -indev ${imageFile}2 -boot_image any discard -overwrite on -volid LIVENG_
↪SYSTEM -move initrd.img-4.9.0-7-amd64 live/initrd.img-4.9.0-7-amd64 -- -outdev $
↪{imageFile}3 -blank as_needed
```

We finally obtain an image identical to the original Debian live, but with an initrd capable of doing cryptsetup:

```
imageFile=${imageFile}3
```

If we repeat all the liveng steps, starting from the beginning, with this image as the base, we get an encrypted persisten live as a result!

3.5.4 Conclusion

liveng is now Secure Boot capable and persistent, with the data partition encrypted with the passphrase of your choice (here *PASSWORD*). When you will choose another passphrase, just keep in mind that the keymap used during the system booting is en_US.

3.6 Kernel update

So far, liveng is a Secure Boot capable and persistent (with or without AES encryption) live operating system. The partitioning scheme we used is made up of two system partitions, one containing all the "interesting" files (kenel, initrd, filesystem.squashfs) and the other one containing kernel and initrd only.

The bootloader is instructed to boot by default from the second system partition, because that one always contains the most updated kernel and initrd files: at every kernel update, this small partition will be overwritten, and we are now covering how to do the rewrite in great detail.

(First) System partition files are kept at their default state and can be useful in case of recovery or when a complete persistence reset is performed (a fallback boot menu has been added as well).

3.6.1 Prerequisites

If you opted for having a liveng system with an encrypted data persistence partition, because we are starting from a stock Debian live image which lacks cryptsetup files in its initrd, we need to check that cryptsetup won't break the initrd when updating the kernel (see why). On a running liveng operating system, do:

```
apt-get install -y cryptsetup
sed -i 's/#CRYPTSETUP=/CRYPTSETUP=y/g' /etc/cryptsetup-initramfs/conf-hook
```

If you opted for a liveng without encryption you can skip the above steps.

Everyone must finally install xorriso onto the system:

```
apt-get install -y xorriso
```

3.6.2 Manually updating the kernel

When normally updating the kernel on a live operating system, only the content of / is modified (and persisted if on a persistent live), but the content of the outside-kernel and ouside-initrd files to which the bootloader links remains unchanged (and untouched).

In addition, the initrd build isn't performed while on a ISO9660 live operating system, so we have to force it.

We start by creating an update script:

```
#!/bin/bash

kernelRelease=$1

if [ ! -z $kernelRelease ]; then
    livengBootDevice=/dev/$(mount | grep persistence | head -1 | awk -F'[0-9/]' '
↪{print $4}')
    if [ ! -z $livengBootDevice ]; then
        livengSecondSystemPartition=${livengBootDevice}2
        livengSecondSystemPartitionUUID=$(blkid -s UUID ${livengSecondSystemPartition}
↪ | awk -F\" '{print $2}')

        cd /boot
        if [ -f vmlinuz-${kernelRelease} ] && [ -f initrd.img-${kernelRelease} ]; then
            mkdir -p temp/live
            cp -a vmlinuz-${kernelRelease} temp/live/vmlinuz
            cp -a initrd.img-${kernelRelease} temp/live/initrd.img
```

(continues on next page)

(continued from previous page)

```
            cd temp
            echo "Writing $livengSecondSystemPartition (UUID:
↪$livengSecondSystemPartitionUUID)"
            umount $livengSecondSystemPartition

            if xorrisofs -volid LIVENG_SYSTEM2 --modification-date=$(echo
↪$livengSecondSystemPartitionUUID | sed -e 's#-##g') -o $livengSecondSystemPartition␣
↪. ; then
                exit 0
            fi
        fi
    fi
fi

exit 1
```

We can name the script as *livengkupd.sh* and move it to /sbin/ with root:root, 750 permissions.

The script must be called by passing it the release of the initrd (= kernel) release we are updating. It will locate the second system partition and rewrite it with the new files, using xorriso and maintaining its old UUID.

As example, we will now update the liveng kernel from the one currently running:

```
uname -a

Linux debian 4.9.0-7-amd64 #1 SMP Debian 4.9.110-1 (2018-07-05) x86_64 GNU/Linux
```

to the one available in the Debian Backports repository:

```
echo "deb http://deb.debian.org/debian/ stretch-backports main contrib non-free" > /
↪etc/apt/sources.list.d/backports.list
apt-get update

apt-cache search linux | grep image
apt-get install -y linux-image-4.17.0-0.bpo.3-amd64
```

Note that, as stated before, the update won't trigger the initrd rebuild:

```
etc/kernel/postinst.d/initramfs-tools:
I: update-initramfs is disabled (live system is running on read-only media).
```

In fact the new initrd is missing:

```
cd /boot
ls -al

-rw-r--r-- 1 root root 23504103 Jul 14 16:58 initrd.img-4.9.0-7-amd64

-rw-r--r-- 1 root root  5068656 Aug 27 08:20 vmlinuz-4.17.0-0.bpo.3-amd64
-rw-r--r-- 1 root root  4224800 Jul  5 01:29 vmlinuz-4.9.0-7-amd64
```

So me must manually force its build:

```
cd /boot
mkinitramfs -o initrd.img-4.17.0-0.bpo.3-amd64 4.17.0-0.bpo.3-amd64
```

The task will add a new file:

```
rw-r--r-- 1 root root 27882891 Oct  3 08:41 initrd.img-4.17.0-0.bpo.3-amd64
```

We now only need to call the *livengkupd* script:

```
/sbin/livengkupd.sh 4.17.0-0.bpo.3-amd64
```

which will rewrite the second system partition:

```
Writing /dev/sdb2 (UUID: 2018-10-02-14-46-02-00)
xorriso 1.4.6 : RockRidge filesystem manipulator, libburnia project.

Drive current: -outdev 'stdio:/dev/sdb2'
Media current: stdio file, overwriteable
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data,  256m free
Added to ISO image: directory '/'='/boot/temp'
xorriso : UPDATE : 3 files added in 1 seconds
xorriso : UPDATE : 3 files added in 1 seconds
xorriso : UPDATE : Thank you for being patient. Working since 0 seconds.
xorriso : UPDATE : Thank you for being patient. Working since 1 seconds.
ISO image produced: 16273 sectors
Written to medium : 16273 sectors at LBA 0
Writing to 'stdio:/dev/sdb2' completed successfully.
```

If we reboot and give:

```
uname -a
```

we can see the new running kernel:

```
Linux debian 4.17.0-0.bpo.3-amd64 #1 SMP Debian 4.17.17-1~bpo9+1 (2018-08-27) x86_64␣
↪GNU/Linux
```

proving that now **liveng is capable of kernel update** on a ISO9660 system partition!!

3.7 Kernel update, the Debian way

So far, liveng is a Secure Boot compliant and persistent live operating system, capable of kernel update on a ISO9660 system partition.

We must now **package all the code previously given programmatically "by hand" into a couple of Debian (.deb) packages**. While users who simply need to improve a standard Debian live can follow all the previously described steps "by hand" at every kernel release (i.e. on every *linux-image-architecture* package release), if you want to **create a liveng-derived operating system**, you will need to **host liveng kernel-update related packages in your package repository**.

On a Debian system, any *linux-image-architecture-version* update is started by an update of the meta package *linux-image-architecture* (for example: *linux-image-amd64* on x64 machines), which depends on the real *linux-image-architecture-version* package, so updating *linux-image-architecture* will download and install *linux-image-architecture-version*.

Instead of relying on the *linux-image-architecture* meta-package for a kernel update, as distribution maintainers we need to install and update the *linux-image-liveng-architecture* meta-package, which:

- depends on the *linux-image-liveng-common* package, which installs the *livengkupd.sh* script;
- depends (or pre-depends) on the real liveng kernel package (*linux-image-liveng-architecture-version*).

linux-image-liveng-architecture-version must have a *postinst* file which mimes what previously done:

- triggers the initrd rebuild;
- calls the *livengkupd.sh* script with the correct kernel-version parameter. *livengkupd.sh* will then locate the second system partition and rewrite it with the new files, using xorriso.

At the end of the day, *linux-image-liveng-architecture-version* is exactly like a standard *linux-image-architecture-version* but adds a special *postinst* file.

We are not going to give further explanations, bacause of course distributions' maintainers already well know how to create and manage Debian packages and a package repository.

3.8 Kernel update improvements

So far, liveng is a Secure Boot compliant and persistent (with or without AES encryption) live operating system, capable of kernel update on a ISO9660 system partition with a proper Debian package-driven procedure.

A liveng system is finally quite ready, we just need to provide a way-out for the system in case something goes wrong when kernel-updating (for example if a power failure occurs when xorriso is writing).

3.8.1 Handle kernel update failures

When normally updating the kernel on a Debian live operating system, only the content of / is modified (and persisted if on a persistent live), but the content of the outside-kernel and ouside-initrd files to which the bootloader links remains unchanged. On liveng, the postinst script of *linux-image-liveng-architecture-version* is responsible to force the initrd update (update-initramfs, usually undone being the filesystem a readonly ISO9660), while */sbin/livengkupd.sh* locates the second system partition and rewrites it with the new files, using xorriso. Task will take very few seconds:

```
Writing /dev/sdb2 (UUID: 2018-10-02-14-46-02-00)
xorriso 1.4.6 : RockRidge filesystem manipulator, libburnia project.

Drive current: -outdev 'stdio:/dev/sdb2'
Media current: stdio file, overwriteable
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data,  256m free
Added to ISO image: directory '/'='/boot/temp'
xorriso : UPDATE : 3 files added in 1 seconds
xorriso : UPDATE : 3 files added in 1 seconds
xorriso : UPDATE : Thank you for being patient. Working since 0 seconds.
xorriso : UPDATE : Thank you for being patient. Working since 1 seconds.
ISO image produced: 16273 sectors
Written to medium : 16273 sectors at LBA 0
Writing to 'stdio:/dev/sdb2' completed successfully.
```

but any power failure during this phase can be disastrous leaving the system in an unbootable state; this is why liveng comes with a second, fallback, GRUB boot parameter.

We need a "handler" for this, which may be an ad-hoc Systemd unit file or some simple lines of code in the (deprecated but always useful) */etc/rc.local* init script:

```
# Rewrite kernel partition if started in fallback boot.
if cat /proc/cmdline | grep -q liveng-fallback; then
    ( /usr/bin/dpkg --configure -a && /usr/bin/apt-get install --reinstall linux-
↪image-liveng-architecture) &
fi
```

Change linux-image-liveng-architecture according to your system's architecture (for example: *linux-image-liveng-amd64*).

Above code will "fix" the state of apt/dpkg (remember, we left it in an unconsistent state caused by a power failure) and will reinstall the liveng kernel package, which in its turn will re-trigger the initrd build and re-launch */sbin/livengkupd.sh*.

3.9 About liveng

LumIT S.p.A. is a high added value System Integrator, specialised in IT services tailored for large customers mainly in the Finance and Telecommunications markets.

LumIT S.p.A. was born in 2009 from the idea of two young Italian entrepreneurs who quickly create a team of IT professionals specialised in network security, in the management and implementation of complex business projects and in the optimisation of critical processes.

> Professionalism and experience are our key words and we make customer satisfaction our main goal.
>
> Our technical team consists of highly qualified engineers in the field of network and security, managed IT services, open source and virtualisation, with years of experience in the field.
>
> Our flexibility and our speed of intervention at all levels have earned us the title of "True Problem Solver" at our customers!
>
> Through skills that are constantly updated, we study the integration and interoperability of technological solutions to simplify their use, reduce management costs and minimise risk exposure. We also host the development of new IT technologies, as well the improvement of some of the existing ones, at LumIT Labs, our innovation laboratories.
>
> Precisely for these reasons, we are able to support our customers, providing services tailored to their needs and always paying attention to their requests rather than to a specific brand.

The **liveng project** has been created by **LumIT Labs**, an innovation laboratory from LumIT S.p.A. (liveng team: Marco Buratto, Michele Sartori).

3.10 liveng-based operating systems and tools

- Secure-K OS, "Security & privacy oriented encrypted live operating system for everyone" (it's a *freemium* operating system);
- Open Secure-K OS, free and open source next generation live operating system, on which Secure-K OS is built. Code and documentation contributions are welcome.
- Open Secure-K OS Deployer is the deployment system for writing Open Secure-K OS onto a USB key - it will **create the liveng partitioning scheme** with ease.

3.11 License

LumIT-Labs/liveng is licensed under the GNU General Public License v3.0.

Permissions of this strong copyleft license are conditioned on making available complete source code of licensed works and modifications, which include larger works using a licensed work, under the same license. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights.