1. “...Download a copy of Arch Linux from: https://www.archlinux.org/download. ...”

2. “...Use Virtual Box to create a new virtual machine and hard disk to test (10 Gb should suffice [be enough!]) Ensure you check the Enable EFI option on the System Page of the virtual machine's settings to give the machine UEFI firmware. Attach the ISO image and start the virtual machine. It should eventually display a root prompt. Log in - there is no password.

You need to create four partitions. You can use [i]gdisk[/i] to do this. The sizes are suggestions sufficient for this exercise, but other values could be used. The ESP is created from sector 2048 (the default offered by [i]gdisk[/i]) and the BIOS boot partition is put in the space preceding the ESP. See the table for the suggested partition.

Number	Start	End	Size	Code Name

1	2048	411647	200.0 MiB	EF00 EFI System
2	34	2047	1007.0 KiB	EF02 BIOS boot partition
3	411648	821247	200.0 MiB	8300 Linux /boot filesystem
4	821248	20971486	200.0 MiB	8300 Linux /root filesystem

Start *gdisk* with gdisk /dev/sda and then use o to create a new empty partition table. Use n to create a partition, t to set its type code and (optionally) c to change its name. Write the new partition table with w.

Or you can use *parted:*

```
# parted /def/sda
(parted) unit s
(parted) mktable gpt
(parted) mkpart primary 2048 411647
(parted) set 1 boot on
(parted) name 1 “EFI System Partition”
(parted) mkpart primary 34 2047
(parted) name 2 “BIOS Boot Partition”
(parted) set 2 bios_grub on
(parted) mkpart primary ext2 411648 821247
(parted) name 3 “Linux /boot filesystem”
(parted) mkpart primary ext4 821248 20971486
(parted) name 4 “Linux /root filesystem”
(parted) quit
```

We've used GPT partitioning here, but MSDOS partitioning can be used instead if the disk is smaller than 2 TiB. In that scenario, omit the BIOS boot partition and use fdisk to change the partiton type of the EFI System Partition to OxEF. The *VirtualBox* UEFI firmware works with either GPT or MSDOS partitions, but other firmwares may only support GPT.

Make the filesystems and mount them:

```
# mkfs.vfat -F32 /dev/sda1
# mkfs.ext2 /dev/sda3
# mkfs.ext4 /dev/sda4
# mount /dev/sda4 /mnt
# mkdir /mnt/boot
# mount /dev/sda3 /mnt/boot
# mkdir /mnt/boot/efi
# mount /dev/sda1 /mnt/boot/efi
```

Use the ArchLinux pacstrap utility to install a new system on the prepared filesystems. An internet connection is required [uh oh! Not a member of USB group!]:

```
# pacstrap /mnt base
# genfstab -p -U /mnt | sed 's/cp437/437/' >>/
mnt/etc/fstab
# arch-chroot /mnt pacman -S grub-efi-x86_64
# modprobe efivars
# arch-chroot /mnt grub-install –target=x86_64-efi –efi-directory=/boot/efi –
bootloader-id=
arch_grub –recheck
# arch-chroot /mnt grub-mkconfig -o /boot/
grub/grub.cfg
# umount /mnt/boot/efi /mnt/boot/mnt
```

This installs the system files onto the root filesystem mounted at /mnt. It also appends to /etc/fstab to cater for the additional boot and EFI filesystems mounted

underneath /mnt (sed is used to get around a bug in genfstab).
We then install the grub package and ensure that the efivars kernel module is loaded (it makes the EFI variables available under /sys/firmware/efi). Next grub-install installs *Grub* into a new arch-grub subdirectory of the ESP that we mounted at /boot/efi. If all goes well, it should respond with "Installation finished: No error reported.". After that, we generate the *Grub* configuration file and unmount our filesystems from /mnt. If you reset the virtual machine, it should offer the newly configured *Grub* menu and boot our system.

Part of the *Grub* setup configures the EFI boot order in the NVRAM. This would normally configue UEFI so that it automatically loads *Grub* when the machine is started. However, *VirtualBox* does not persist the firmware NVRAM after the virtual machine is stopped, and this results in the boot settings being lost.

If this happens, the EFI shell will be started and you'll need to manually launch the

bootloader.

```
2.0 Shell> fs0:\EFI\arch_grub\grubx64.efi
```

Once the OS has booted, to work around the lack of persistent NVRAM, set up a default bootloader:

```
# mkdir /boot/efi/EFI/BOOT
# cp /boot/efi/EFI/{arch_grub/grub.BOOT/BOOT}x64.efi
```

The firmware will fall back to EFI\BOOT\BOOTx64.efi if no boot order is cofigured, as will be the case with nothing in the NVRAM. This should result in a successful boot after starting the virtual machine. Now, to make the system boot on a BIOS, it is a simple matter of also setting up the BIOS boot configuration:

```
# dhcpd
# pacman -S grub-bios
# grub-install –target=i386-pc –recheck /dev/sda
```

Start by establishing network connectivity – using dhcpd is one way to do this. Then install the grub-bios package and install the BIOS version of *Grub* to the disk. This will set up the boot code in the master boot record and the BIOS boot partition that we set up earlier. It uses the same *Grub* configuration file, /boot/grub/grub.cfg, that we set up before.

Shut down the virtual machine and change it from EFI to BIOS by unchecking the Enable EFI option on the System page of its settings. When the virtual machine is restartec, it will boot through its BIOS. This example demonstrates that it is easy to install a system onto a GPT or MSDOS partitioned disk that works under UEFI and BIOS. It is worth mentioning that the ride with some other distributions can be less smooth.

OK, let's tackle the elephant in the room: secure boot. The gist of secure boot is that UEFI won't load something that isn't signed with a recognised key. New PCs with

Windows-8 pre-installed ship with a key from Microsoft to allow their latest operating system to boot securely. If such a system has secure boot enabled, and only has that key, then the system will only boot something signed with that key. Most new PCs are pre-configured in this way in order to gain Windows 8 certification.

Machines with Windows 8 pre-installed will have secure boot enabled by default. You can, however, disable secure boot. At this time, the most practical solution to the secure boot issue for [GNU/]Linux users is to disable secure boot.

However, some people may want or need to use secure boot, and this requires having [GNU/]Linux signed with a key that the firmware recognises.

For [GNU/]Linux to secure boot, there are two basic options. The first is to get OEMs to include additional keys that can be used to sign [GNU/]Linux. The second is to sign [GNU/]Linux with the Microsoft key. There are practical considerations that

make both these schemes unattractive.

Signing the [GNU/]Linux kernel is impractical, due to its fast-changing nature and the fact that many people build their own custom kernel. So the approach being taken is to use a signed bootloader that can then load any kernel image.

Examples of signed bootloaders include the [GNU/]Linux Foundation's pre-bootloader and solutions from distro-makers, like the Fedora Shim [this latter one goes against the Policy Ruling of the Free Software Foundation – not pointed out in the article!]. By being signed with Microsoft's key, these should work on any machine shipped with Windows 8. However, getting them signed is proving difficult (http://bit.ly/VFEAV9).

To avoid the potential for these solutions to be used as a way for malware to affect secure systems, they all require a human user to be present who must confirm they

wish to proceed with an unsigned bootloader.

The idea behind the [GNU/]Linux Foundation's pre-bootloader is to provide a signed loader which will chain-load another bootloader (like *Grub*) that then loads [GNU/]Linux. Secure verification stops after the pre-bootloader, which makes this approach no more secure than having UEFI secure boot disabled. It does however, allow [GNU/]Linux and other insecure operating systems to be booted in a secure boot environment. This solution is being provided by the Foundation as an interim solution until distributions implement fully secure solutions. [Comment: this seems to be contradictory as the Foundation have stipulated that *Grub* has to be implemented in any solution – go figure! - brings back the argument that as users we should all be migrating to Lemote hardware!]

The Fedora Shim aims to go a step further by enabling secure boot of a [GNU/]Linux system. It loads a special version of *Grub* that contains a Fedora public key, and it

will securely boot kernels that can be validated with that key. It will also boot other kernels, but doing so will require a physical presence during boot. Securely booted kernels will also restrict the boot command-line and require kernel modules to be signed. The shim is a way to securely boot a pre-determined [GNU/]Linux configuration, but it still proves difficult when customisations are required. This would need a custom shim containing locally produced keys that is signed with a key recognised by the firmware. How one would achieve this is not yet clear. [Migrate to Lemote hardware!!!] Similar approaches are being taken by SUSE and Ubuntu.

The signing mechanism for secure boot is called Microsoft Authenticode, and is managed by Symantec (formerly VeriSign). [And we all know why that is – because some dopey M$ employee lost VeriSign keys!] For a $99 annual fee, you can sign as many binaries as you wish, via the Microsoft Developer Center at https://sysdev.microsoft.com (you'll need a Windows Live ID to sign in).

There isn't yet a straightforward answer to the secure boot question. [Oh yes there is - migrate your hardware thinking to Lemote!] There are methods that allow working around the problem, but it's too early to know how easy it will be to run [GNU/]Linux within a secure boot environment. If you feel strongly about this, you may be interested in the FSF's campaign: (http://bit.ly/nHYBRN). And there is always the option of disabling secure boot – you'll be no worse off than you are today."