(/) **Wiki**

Configuring the Linux kernel

From Gentoo Wiki

< Handbook:AMD64 (/wiki/Special:MyLanguage/Handbook:AMD64) | Installation (/wiki/Special:MyLanguage/Handbook:AMD64/Installation)

Jump to:navigation Jump to:search

AMD64 Handbook (/wiki/Handbook:AMD64)
Installation (/wiki/Handbook:AMD64/Full/Installation)
About the installation (/wiki/Handbook:AMD64/Installation/About)
Choosing the media (/wiki/Handbook:AMD64/Installation/Media)
Configuring the network (/wiki/Handbook:AMD64/Installation/Networking)
Preparing the disks (/wiki/Handbook:AMD64/Installation/Disks)
The stage file (/wiki/Handbook:AMD64/Installation/Stage)
Installing base system (/wiki/Handbook:AMD64/Installation/Base)
Configuring the kernel
Configuring the system (/wiki/Handbook:AMD64/Installation/System)
Installing tools (/wiki/Handbook:AMD64/Installation/Tools)
Configuring the bootloader (/wiki/Handbook:AMD64/Installation/Bootloader)
Finalizing (/wiki/Handbook:AMD64/Installation/Finalizing)
Working with Gentoo (/wiki/Handbook:AMD64/Full/Working)
Portage introduction (/wiki/Handbook:AMD64/Working/Portage)
USE flags (/wiki/Handbook:AMD64/Working/USE)
Portage features (/wiki/Handbook:AMD64/Working/Features)
Initscript system (/wiki/Handbook:AMD64/Working/Initscripts)
Environment variables (/wiki/Handbook:AMD64/Working/EnvVar)
Working with Portage (/wiki/Handbook:AMD64/Full/Portage)
Files and directories (/wiki/Handbook:AMD64/Portage/Files)
Variables (/wiki/Handbook:AMD64/Portage/Variables)

Contents

Optional: Installing firmware and/or microcode

Firmware

Suggested: Linux Firmware

On many systems, non-FOSS firmware is required for certain hardware to function. The `sys-kernel/linux-firmware` (https://packages.gentoo.org/packages/sys-kernel/linux-firmware) package contains firmware for many, but not all, devices.

Tip
Most wireless cards and GPUs require firmware to function.

```
root # emerge --ask sys-kernel/linux-firmware
```

Note
Installing certain firmware packages often requires accepting the associated firmware licenses. If necessary, visit the license handling section (/wiki/Handbook:AMD64/Working/Portage#Licenses) of the Handbook for help on accepting licenses.

Firmware Loading

Firmware files are typically loaded when the associated kernel module is loaded. This means the firmware must be built into the kernel using **CONFIG_EXTRA_FIRMWARE** if the kernel module is set to *Y* instead of *M*. In most cases, building-in a module which required firmware can complicate or break loading.

SOF Firmware

Sound Open Firmware (SOF) is a new open source audio driver meant to replace the proprietary Smart Sound Technology (SST) audio driver from Intel. 10th gen+ and Apollo Lake (Atom E3900, Celeron N3350, and Pentium N4200) Intel CPUs require this firmware for certain features and certain AMD APUs also have support for this firmware. SOF's supported platforms matrix can be found here (https://thesofproject.github.io/latest/platforms/index.html) for more information.

```
root # emerge --ask sys-firmware/sof-firmware
```

Microcode

In addition to discrete graphics hardware and network interfaces, CPUs also can require firmware updates. Typically this kind of firmware is referred to as *microcode*. Newer revisions of microcode are sometimes necessary to patch instability, security concerns, or other miscellaneous bugs in CPU hardware.

Microcode updates for AMD CPUs are distributed within the aforementioned `sys-kernel/linux-firmware` (https://packages.gentoo.org/packages/sys-kernel/linux-firmware) package. Microcode for Intel CPUs can be found within the `sys-firmware/intel-microcode` (https://packages.gentoo.org/packages/sys-firmware/intel-microcode) package, which will need to be installed separately. See the Microcode article (/wiki/Microcode) for more information on how to apply microcode updates.

sys-kernel/installkernel

Installkernel (/wiki/Installkernel) may be used to automate the kernel installation, initramfs (/wiki/Initramfs) generation, unified kernel image (/wiki/Unified_kernel_image) generation and/or bootloader configuration among other things. `sys-kernel/installkernel` (https://packages.gentoo.org/packages/sys-kernel/installkernel) implements two paths of achieving this: the traditional **`installkernel`** originating from Debian and systemd (/wiki/Systemd)'s **`kernel-install`**. Which one to choose depends, among other things, on the system's bootloader. By default, systemd's **`kernel-install`** is used on systemd profiles, while the traditional **`installkernel`** is the default for other profiles.

Bootloader

Now is the time to think about which bootloader the user wants for the system, if unsure, follow the 'Traditional layout' subsection below.

GRUB

Users of GRUB can use either systemd's **`kernel-install`** or the traditional Debian **`installkernel`**. The `systemd (https://packages.gentoo.org/useflags/systemd) (/wiki/USE_flag)` USE flag switches between these implementations. To automatically run **`grub-mkconfig`** when installing the kernel, enable the `grub (https://packages.gentoo.org/useflags/grub) (/wiki/USE_flag)` USE flag (/wiki/USE_flag).

FILE **`/etc/portage/package.use/installkernel`**

```
sys-kernel/installkernel grub
```

```
root # emerge --ask sys-kernel/installkernel
```

systemd-boot

When using systemd-boot (/wiki/Systemd/systemd-boot) (formerly gummiboot) as the bootloader, systemd's **`kernel-install`** must be used. Therefore ensure the `systemd (https://packages.gentoo.org/useflags/systemd) (/wiki/USE_flag)` and the `systemd-boot (https://packages.gentoo.org/useflags/systemd-boot) (/wiki/USE_flag)` USE flags are enabled on `sys-kernel/installkernel` (https://packages.gentoo.org/packages/sys-kernel/installkernel), and then install the relevant package for systemd-boot.

On OpenRC systems:

FILE **`/etc/portage/package.use/systemd-boot`**

```
sys-apps/systemd-utils boot kernel-install
sys-kernel/installkernel systemd systemd-boot
```

```
root # emerge --ask sys-apps/systemd-utils sys-kernel/installkernel
```

On systemd systems:

FILE **`/etc/portage/package.use/systemd`**

```
sys-apps/systemd boot
sys-kernel/installkernel systemd-boot
```

```
root # emerge --ask sys-apps/systemd sys-kernel/installkernel
```

The kernel command line to use for new kernels should be specified in **/etc/kernel/cmdline**, for example:

FILE **/etc/kernel/cmdline**

```
quiet splash
```

EFI stub

UEFI-based computer systems technically do not need secondary bootloaders in order to boot kernels. Secondary bootloaders exist to *extend* the functionality of UEFI firmware during the boot process. That being said, using a secondary bootloader is typically easier and more robust because it offers a more flexible approach for quickly modifying kernel parameters at boot time. Note also that UEFI implentations strongly differ between vendors and between models and there is no guarantee that a given firmware follows the UEFI specification. Therefore, EFI Stub booting is not guaranteed to work on every UEFI-based system, and hence the USE flag is stable masked and testing keywords must be accepted for installkernel to use this feature.

FILE **/etc/portage/package.accept_keywords/installkernel**

```
sys-kernel/installkernel
sys-boot/uefi-mkconfig
app-emulation/virt-firmware
```

FILE **/etc/portage/package.use/installkernel**

```
sys-kernel/installkernel efistub
```

```
root # emerge --ask sys-kernel/installkernel
root # mkdir -p /efi/EFI/Gentoo
```

Traditional layout, other bootloaders (e.g. (e)lilo, syslinux, etc.)

The traditional **/boot** layout (for e.g. (e)LILO, syslinux, etc.) is used by default if the `grub (https://packages.gentoo.org/useflags/grub)` `(/wiki/USE_flag)`, `systemd-boot (https://packages.gentoo.org/useflags/systemd-boot)` `(/wiki/USE_flag)`, `efistub (https://packages.gentoo.org/useflags/efistub)` `(/wiki/USE_flag)` and `uki (https://packages.gentoo.org/useflags/uki)` `(/wiki/USE_flag)` USE flags are **not** enabled. No further action is required.

Initramfs

An **i**nitial **ram**-based **f**ile **s**ystem, or initramfs (/wiki/Initramfs), may be required for a system to boot. A wide of variety of cases may necessitate one, but common cases include:

- Kernels where storage/filesystem drivers are modules.
- Layouts with **/usr/** or **/var/** on separate partitions.
- Encrypted root filesystems.

Tip
Distribution kernels (/wiki/Project:Distribution_Kernel) are designed to be used with an initramfs, as many storage and filesystem drivers are built as modules.

In addition to mounting the root filesystem, an initramfs may also perform other tasks such as:

- Running **f**ile **s**ystem **c**onsistency chec**k** **fsck**, a tool to check and repair consistency of a file system in such events of uncleanly shutdown a system.
- Providing a recovery environment in the event of late-boot failures.

Installkernel (/wiki/Installkernel) can automatically generate an initramfs when installing the kernel if the `dracut (https://packages.gentoo.org/useflags/dracut)` `(/wiki/USE_flag)` or `ugrd (https://packages.gentoo.org/useflags/ugrd)` `(/wiki/USE_flag)` USE flag is enabled:

FILE **/etc/portage/package.use/installkernel**

```
sys-kernel/installkernel dracut
```

```
root # emerge --ask sys-kernel/installkernel
```

Optional: Unified Kernel Image

A Unified Kernel Image (/wiki/Unified_Kernel_Image) (UKI) combines, among other things, the kernel, the initramfs and the kernel command line into a single executable. Since the kernel command line is embedded into the unified kernel image, it should be specified before generating the unified kernel image (see below). Note that any kernel command line arguments supplied by the bootloader or firmware at boot are ignored when booting with secure boot enabled.

A unified kernel image requires a stub loader. Currently, the only one available is **systemd-stub**. To enable it:

For systemd systems:

FILE **/etc/portage/package.use/uki**

```
sys-apps/systemd boot
```

```
root # emerge --ask sys-apps/systemd
```

For OpenRC systems:

FILE **/etc/portage/package.use/uki**

```
sys-apps/systemd-utils boot kernel-install
```

```
root # emerge --ask sys-apps/systemd-utils
```

Installkernel (/wiki/Installkernel) can automatically generate a unified kernel image using either dracut (/wiki/Unified_kernel_image#dracut) or ukify (/wiki/Unified_kernel_image#ukify) by enabling the respective flag and the `uki (https://packages.gentoo.org/useflags/uki)` `(/wiki/USE_flag)` USE flag.

For dracut:

FILE **/etc/portage/package.use/uki**

```
sys-kernel/installkernel dracut uki
```

FILE **/etc/dracut.conf.d/uki.conf**

```
uefi="yes"
kernel_cmdline="some-kernel-command-line-arguments"
```

```
root # emerge --ask sys-kernel/installkernel
```

For ukify:

FILE **/etc/portage/package.use/uki**

```
sys-apps/systemd boot ukify                         # For systemd systems
sys-apps/systemd-utils kernel-install boot ukify    # For OpenRC systems
sys-kernel/installkernel dracut ukify uki
```

FILE **/etc/kernel/cmdline**

```
some-kernel-command-line-arguments
```

```
root # emerge --ask sys-kernel/installkernel
```

Note that while dracut can generate both an initramfs and a unified kernel image, ukify can only generate the latter and therefore the initramfs must be generated separately with dracut.

Important

In the above configuration examples (for both Dracut and ukify) it is important to specify at least an appropriate *root*= parameter for the kernel command line to ensure that the Unified Kernel Image can find the root partition. This is not required for systemd based systems following the Discoverable Partitions Specification (DPS), in that case the embedded initramfs will be able to dynamically find the root partition.

Generic Unified Kernel Image

The prebuilt `sys-kernel/gentoo-kernel-bin` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel-bin) can optionally install a prebuilt generic unified kernel image containing a generic initramfs that is able to boot most systemd based systems. It can be installed by enabling the `generic-uki (https://packages.gentoo.org/useflags/generic-uki)` `(/wiki/USE_flag)` USE flag, and configuring installkernel (/wiki/Installkernel) to not generate a custom initramfs or unified kernel image:

FILE **/etc/portage/package.use/uki**

```
sys-apps/systemd boot                         # For systemd systems
sys-apps/systemd-utils kernel-install boot     # For OpenRC systems
sys-kernel/gentoo-kernel-bin generic-uki
sys-kernel/installkernel -dracut -ukify -ugrd uki
```

Secure Boot

Warning

If following this section and manually compiling your own kernel, then make sure to follow the steps outlined in Signing the kernel (/wiki/Kernel#Optional:_Signing_the_kernel_image_.28Secure_Boot.29)

The generic Unified Kernel Image optionally distributed by `sys-kernel/gentoo-kernel-bin` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel-bin) is already pre-signed. How to sign a locally generated unified kernel image depends on whether dracut or ukify is used. Note that the location of the key and certificate should be the same as the *SECUREBOOT_SIGN_KEY* and *SECUREBOOT_SIGN_CERT* as specified in **`/etc/portage/make.conf`**.

For dracut:

FILE **/etc/dracut.conf.d/uki.conf**

```
uefi="yes"
kernel_cmdline="some-kernel-command-line-arguments"
uefi_secureboot_key="/path/to/kernel_key.pem"
uefi_secureboot_cert="/path/to/kernel_key.pem"
```

For ukify:

FILE **/etc/kernel/uki.conf**

```
[UKI]
SecureBootPrivateKey=/path/to/kernel_key.pem
SecureBootCertificate=/path/to/kernel_key.pem
```

Kernel configuration and compilation

Tip
It's can be a wise move to use the dist-kernel on the first boot as it provides a very simple method to rule out system issues and kernel config issues. Always having a known working kernel to fallback on can speed up debugging and alleviate anxiety when updating that your system will no longer boot.

Now it is time to configure and compile the kernel sources. For the purposes of the installation, three approaches to kernel management will be presented, however at any point post-installation a new approach can be employed.

Ranked from least involved to most involved:

Full automation approach: Distribution kernels (/wiki/Handbook:AMD64/Installation/Kernel#Distribution_kernels)
A Distribution Kernel (/wiki/Project:Distribution_Kernel) is used to configure, automatically build, and install the Linux kernel, its associated modules, and (optionally, but enabled by default) an initramfs file. Future kernel updates are fully automated since they are handled through the package manager, just like any other system package. It is possible provide a custom kernel configuration file (/wiki /Project:Distribution_Kernel#Modifying_kernel_configuration) if customization is necessary. This is the least involved process and is perfect for new Gentoo users due to it working out-of-the-box and offering minimal involvement from the system administrator.
Full manual approach (/wiki/Handbook:AMD64/Installation/Kernel#Alternative:_Manual_configuration)
New kernel sources are installed via the system package manager. The kernel is manually configured, built, and installed using the **`eselect kernel`** and a slew of **`make`** commands. Future kernel updates repeat the manual process of configuring, building, and installing the kernel files. This is the most involved process, but offers maximum control over the kernel update process.
Hybrid approach: Genkernel (/wiki/Handbook:AMD64/Installation/Kernel#Alternative:_Genkernel)
We use the term hybrid here but, do note that the dist-kernel and manual sources, both include methods to achieve the same goal. New kernel sources are installed via the system package manager. System administrators may use Gentoo's **`genkernel`** tool to configure, build, and install the Linux kernel, its associated modules, and (optionally, but ***not*** enabled by default) an initramfs file. It is possible provide a custom kernel configuration file if customization is necessary. Future kernel configuration, compilation, and installation require the system administrator's involvement in the form of running **`eselect kernel`**, **`genkernel`**, and potentially other commands for each update. This option should only considered for users that know they have a need for **`genkernel`**

The core around which all distributions are built is the Linux kernel. It is the layer between the user's programs and the system hardware. Although the handbook provides its users several possible kernel sources, a more comprehensive listing with more detailed descriptions is available at the Kernel overview page (/wiki/Kernel/Overview).

Tip
Kernel installation tasks such as copying the kernel image to **`/boot`** or the EFI System Partition (/wiki/EFI_System_Partition), generating an initramfs (/wiki/Initramfs) and/or Unified Kernel Image (/wiki/Unified_Kernel_Image), updating bootloader configuration, can be automated with installkernel (/wiki/Installkernel). Users may wish to configure and install `sys-kernel/installkernel` (https://packages.gentoo.org/packages/sys-kernel/installkernel) before proceeding. See the Kernel installation section below (/wiki/Handbook:AMD64/Installation/Kernel#Kernel_installation) for more more information.

Distribution kernels

Distribution Kernels (/wiki/Project:Distribution_Kernel) are ebuilds that cover the complete process of unpacking, configuring, compiling, and installing the kernel. The primary advantage of this method is that the kernels are updated to new versions by the package manager as part of @world upgrade. This requires no more involvement than running an **`emerge`** command. Distribution kernels default to a configuration supporting the majority of hardware, however two mechanisms are offered for customization: savedconfig and config snippets. See the project page for more details on configuration. (/wiki/Project:Distribution_Kernel#Modifying_kernel_configuration)

Optional: Signed kernel modules

The kernel modules in the prebuilt distribution kernel (`sys-kernel/gentoo-kernel-bin` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel-bin)) are already signed. To sign the modules of kernels built from source enable the `modules-sign (https://packages.gentoo.org/useflags/modules-sign)` `(/wiki/USE_flag)` USE flag, and optionally specify which key to use for signing in /etc/portage/make.conf (/wiki//etc/portage/make.conf):

FILE **`/etc/portage/make.conf`** **Enable module signing**

```
USE="modules-sign"

# Optionally, to use custom signing keys.
MODULES_SIGN_KEY="/path/to/kernel_key.pem"
MODULES_SIGN_CERT="/path/to/kernel_key.pem" # Only required if the MODULES_SIGN_KEY does not also contain the certificate.
MODULES_SIGN_HASH="sha512" # Defaults to sha512.
```

If *MODULES_SIGN_KEY* is not specified the kernel build system will generate a key, it will be stored in **`/usr/src/linux-x.y.z/certs`**. It is recommended to manually generate a key to ensure that it will be the same for each kernel release. A key may be generated with:

```
root # openssl req -new -nodes -utf8 -sha256 -x509 -outform PEM -out kernel_key.pem -keyout kernel_key.pem
```

Note
The *MODULES_SIGN_KEY* and *MODULES_SIGN_CERT* may be different files. For this example the pem file generated by OpenSSL includes both the key and the accompanying certificate, and thus both variables are set to the same value.

OpenSSL will ask some questions about the user generating the key, it is recommended to fill in these questions as detailed as possible.

Store the key in a safe location, at the very least the key should be readable only by the root user. Verify this with:

```
root # ls -l kernel_key.pem
```

```
 -r-------- 1 root root 3164 Jan  4 10:38 kernel_key.pem
```

If this outputs anything other then the above, correct the permissions with:

```
root # chown root:root kernel_key.pem
root # chmod 400 kernel_key.pem
```

Optional: Signing the kernel image (Secure Boot)

The kernel image in the prebuilt distribution kernel (`sys-kernel/gentoo-kernel-bin` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel-bin)) is already signed for use with Secure Boot (/wiki/Secure_Boot). To sign the kernel image of kernels built from source enable the `secureboot (https://packages.gentoo.org/useflags/secureboot)` `(/wiki/USE_flag)` USE flag, and optionally specify which key to use for signing in /etc/portage/make.conf (/wiki//etc/portage/make.conf). Note that signing the kernel image for use with secureboot requires that the kernel modules are also signed, the same key may be used to sign both the kernel image and the kernel modules:

FILE **`/etc/portage/make.conf`** **Enable custom signing keys**

```
USE="modules-sign secureboot"

# Optionally, to use custom signing keys.
MODULES_SIGN_KEY="/path/to/kernel_key.pem"
MODULES_SIGN_CERT="/path/to/kernel_key.pem" # Only required if the MODULES_SIGN_KEY does not also contain the certificate.
MODULES_SIGN_HASH="sha512" # Defaults to sha512.

# Optionally, to boot with secureboot enabled, may be the same or different signing key.
SECUREBOOT_SIGN_KEY="/path/to/kernel_key.pem"
SECUREBOOT_SIGN_CERT="/path/to/kernel_key.pem"
```

Note
The **`SECUREBOOT_SIGN_KEY`** and **`SECUREBOOT_SIGN_CERT`** may be different files. For this example the pem file generated by OpenSSL includes both the key and the accompanying certificate, and thus both variables are set to the same value.

Note
For this example the same key that was generated to sign the modules is used to sign the kernel image. It is also possible to generate and use a second separate key for signing the kernel image. The same OpenSSL command as in the previous section may be used again.

See the above section for instructions on generating a new key, the steps may be repeated if a separate key should be used to sign the kernel image.

To successfully boot with Secure Boot enabled, the used bootloader must also be signed and the certificate must be accepted by the UEFI (/wiki/UEFI) firmware or Shim (/wiki/Shim). This will be explained later in the handbook.

Installing a distribution kernel

To build a kernel with Gentoo patches from source, type:

```
root # emerge --ask sys-kernel/gentoo-kernel
```

System administrators who want to avoid compiling the kernel sources locally can instead use precompiled kernel images:

```
root # emerge --ask sys-kernel/gentoo-kernel-bin
```

Important
Distribution Kernels (/wiki/Project:Distribution_Kernel), such as `sys-kernel/gentoo-kernel` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel) and `sys-kernel/gentoo-kernel-bin` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel-bin), by default, expect to be installed alongside an initramfs (/wiki/Handbook:AMD64/Installation/Kernel#Initramfs). Before running emerge to install the kernel users should ensure that `sys-kernel/installkernel` (https://packages.gentoo.org/packages/sys-kernel/installkernel) has been configured to utilize an initramfs generator (for example Dracut (/wiki/Dracut)) as described in the installkernel section (/wiki /Handbook:AMD64/Installation/Kernel#Initramfs).

Upgrading and cleaning up

Once the kernel is installed, the package manager will automatically update it to newer versions. The previous versions will be kept until the package manager is requested to clean up stale packages. To reclaim disk space, stale packages can be trimmed by periodically running emerge with the `--depclean` option:

```
root # emerge --depclean
```

Alternatively, to specifically clean up old kernel versions:

```
root # emerge --prune sys-kernel/gentoo-kernel sys-kernel/gentoo-kernel-bin
```

Tip
By design, emerge only removes the kernel build directory. It does not actually remove the kernel modules, nor the installed kernel image. To completely clean-up old kernels, the `app-admin/eclean-kernel` (https://packages.gentoo.org/packages/app-admin/eclean-kernel) tool may be used.

Post-install/upgrade tasks

An upgrade of a distribution kernel is capable of triggering an automatic rebuild for external kernel modules installed by other packages (for example: `sys-fs/zfs-kmod` (https://packages.gentoo.org/packages/sys-fs/zfs-kmod) or `x11-drivers/nvidia-drivers` (https://packages.gentoo.org/packages/x11-drivers/nvidia-drivers)). This automated behaviour is enabled by enabling the `dist-kernel (https://packages.gentoo.org/useflags/dist-kernel) (/wiki/USE_flag)` USE flag. When required, this same flag will also trigger re-generation of the initramfs (/wiki/Initramfs).

It is highly recommended to enable this flag globally via **/etc/portage/make.conf** when using a distribution kernel:

FILE **/etc/portage/make.conf** **Enabling USE=dist-kernel**

```
USE="dist-kernel"
```

Manually rebuilding the initramfs or Unified Kernel Image

If required, manually trigger such rebuilds by, after a kernel upgrade, executing:

```
root # emerge --ask @module-rebuild
```

If any kernel modules (e.g. ZFS) are needed at early boot, rebuild the initramfs afterward via:

```
root # emerge --config sys-kernel/gentoo-kernel
root # emerge --config sys-kernel/gentoo-kernel-bin
```

Installing the kernel sources

When installing and compiling the kernel for amd64-based systems, Gentoo recommends the `sys-kernel/gentoo-sources` (https://packages.gentoo.org/packages/sys-kernel/gentoo-sources) package.

Choose an appropriate kernel source and install it using **emerge**:

```
root # emerge --ask sys-kernel/gentoo-sources
```

This will install the Linux kernel sources in **/usr/src/** using the specific kernel version in the path. It will not create a symbolic link by itself without the `symlink (https://packages.gentoo.org/useflags/symlink) (/wiki/USE_flag)` USE flag being enabled on the chosen kernel sources package.

It is conventional for a **/usr/src/linux** symlink to be maintained, such that it refers to whichever sources correspond with the currently running kernel. However, this symbolic link will not be created by default. An easy way to create the symbolic link is to utilize eselect's kernel module.

For further information regarding the purpose of the symlink, and how to manage it, please refer to Kernel/Upgrade (/wiki/Kernel/Upgrade).

First, list all installed kernels:

```
root # eselect kernel list
```

```
Available kernel symlink targets:
  [1]   linux-6.6.21-gentoo
```

In order to create a symbolic link called **linux**, use:

```
root # eselect kernel set 1
root # ls -l /usr/src/linux
```

```
lrwxrwxrwx    1 root   root    12 Oct 13 11:04 /usr/src/linux -> linux-6.6.21-gentoo
```

Alternative: Manual configuration

Note
In case it was missed, this section requires the kernel sources to be installed. (/wiki/Handbook:AMD64/Installation/Kernel#Installing_the_kernel_sources) Be sure to obtain the relevant kernel sources, then return here for the rest of section.

Manually configuring a kernel is commonly seen as one of the most difficult procedures a system administrator has to perform. Nothing is less true - after configuring a few kernels no one remembers that it was difficult! There are two ways for a Gentoo user to manage a manual kernel system, both of which are listed below:

Modprobed-db process

A very easy way to manage the kernel is to first install `sys-kernel/gentoo-kernel-bin` (https://packages.gentoo.org/packages/sys-kernel/gentoo-kernel-bin) and use the `sys-kernel/modprobed-db` (https://packages.gentoo.org/packages/sys-kernel/modprobed-db) to collect information about what the system requires. **`modprobed-db`** is a tool which monitors the system via crontab to add all modules of all devices over the system's life to make sure it everything a user needs is supported. For example, if an Xbox controller is added after installation, then **`modprobed-db`** will add the modules to be built next time the kernel is rebuilt. More on this topic can be found in the Modprobed-db (/wiki/Modprobed-db) article.

Manual process

This method allows a user to have full control of how their kernel is built with as minimal help from outside tools as they wish. Some could consider this as making it hard for the sake of it.

However, with this choice one thing is true: it is vital to know the system when a kernel is configured manually. Most information can be gathered by emerging `sys-apps/pciutils` (https://packages.gentoo.org/packages/sys-apps/pciutils) which contains the **`lspci`** command:

```
root # emerge --ask sys-apps/pciutils
```

Note
Inside the chroot, it is safe to ignore any pcilib warnings (like *pcilib: cannot open /sys/bus/pci/devices*) that **`lspci`** might throw out.

Another source of system information is to run **`lsmod`** to see what kernel modules the installation CD uses as it might provide a nice hint on what to enable.

Now go to the kernel source directory.

```
root # cd /usr/src/linux
```

The kernel has a method of autodetecting the modules currently being used on the installcd which will give a great starting point to allow a user to configure their own. This can be called by using:

```
root # make localmodconfig
```

It's now time to configure using **`nconfig`**:

```
root # make nconfig
```

The Linux kernel configuration has many, many sections. Let's first list some options that must be activated (otherwise Gentoo will not function, or not function properly without additional tweaks). We also have a Gentoo kernel configuration guide (/wiki/Kernel/Gentoo_Kernel_Configuration_Guide) on the Gentoo wiki that might help out further.

Enabling required options

When using `sys-kernel/gentoo-sources` (https://packages.gentoo.org/packages/sys-kernel/gentoo-sources), it is strongly recommend the Gentoo-specific configuration options be enabled. These ensure that a minimum of kernel features required for proper functioning is available:

KERNEL **Enabling Gentoo-specific options**

```
Gentoo Linux --->
  Generic Driver Options --->
    [*] Gentoo Linux support
    [*]   Linux dynamic and persistent device naming (userspace devfs) support
    [*]   Select options required by Portage features
        Support for init systems, system and service managers  --->
          [*] OpenRC, runit and other script based systems and managers
          [*] systemd
```

Naturally the choice in the last two lines depends on the selected init system (OpenRC (/wiki/OpenRC) vs. systemd (/wiki/Systemd)). It does not hurt to have support for both init systems enabled.

When using `sys-kernel/vanilla-sources` (https://packages.gentoo.org/packages/sys-kernel/vanilla-sources), the additional selections for init systems will be unavailable. Enabling support is possible, but goes beyond the scope of the handbook.

Enabling support for typical system components

Make sure that every driver that is vital to the booting of the system (such as SATA controllers, NVMe block device support, filesystem support, etc.) is compiled in the kernel and not as a module, otherwise the system may not be able to boot completely.

Next select the exact processor type. It is also recommended to enable MCE features (if available) so that users are able to be notified of any hardware problems. On some architectures (such as x86_64), these errors are not printed to **`dmesg`**, but to **`/dev/mcelog`**. This requires the `app-admin/mcelog` (https://packages.gentoo.org/packages/app-admin/mcelog) package.

Also select *Maintain a devtmpfs file system to mount at /dev* so that critical device files are already available early in the boot process (*CONFIG_DEVTMPFS* and *CONFIG_DEVTMPFS_MOUNT*):

KERNEL **Enabling devtmpfs support (*CONFIG_DEVTMPFS*)**

```
Device Drivers --->
  Generic Driver Options --->
    [*] Maintain a devtmpfs filesystem to mount at /dev
    [*]   Automount devtmpfs at /dev, after the kernel mounted the rootfs
```

Verify SCSI disk support has been activated (*CONFIG_BLK_DEV_SD*):

KERNEL **Enabling SCSI disk support (*CONFIG_SCSI*, *CONFIG_BLK_DEV_SD*)**

```
Device Drivers --->
  SCSI device support  --->
    <*> SCSI device support
    <*> SCSI disk support
```

KERNEL **Enabling basic SATA and PATA support (*CONFIG_ATA_ACPI*, *CONFIG_SATA_PMP*, *CONFIG_SATA_AHCI*, *CONFIG_ATA_BMDMA*, *CONFIG_ATA_SFF*, *CONFIG_ATA_PIIX*)**

```
Device Drivers --->
  <*> Serial ATA and Parallel ATA drivers (libata)  --->
    [*] ATA ACPI Support
    [*] SATA Port Multiplier support
    <*> AHCI SATA support (ahci)
    [*] ATA BMDMA support
    [*] ATA SFF support (for legacy IDE and PATA)
    <*> Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support (ata_piix)
```

Verify basic NVMe support has been enabled:

KERNEL **Enable basic NVMe support for Linux 4.4.x (*CONFIG_BLK_DEV_NVME*)**

```
Device Drivers  --->
  <*> NVM Express block device
```

KERNEL **Enable basic NVMe support for Linux 5.x.x (*CONFIG_DEVTMPFS*)**

```
Device Drivers --->
  NVME Support --->
    <*> NVM Express block device
```

It does not hurt to enable the following additional NVMe support:

KERNEL **Enabling additional NVMe support (*CONFIG_NVME_MULTIPATH*, *CONFIG_NVME_MULTIPATH*, *CONFIG_NVME_HWMON*, *CONFIG_NVME_FC*, *CONFIG_NVME_TCP*, *CONFIG_NVME_TARGET*, *CONFIG_NVME_TARGET_PASSTHRU*, *CONFIG_NVME_TARGET_LOOP*, *CONFIG_NVME_TARGET_FC*, *CONFIG_NVME_TARGET_FCLOOP*, *CONFIG_NVME_TARGET_TCP***

```
[*] NVMe multipath support
[*] NVMe hardware monitoring
<M> NVM Express over Fabrics FC host driver
<M> NVM Express over Fabrics TCP host driver
<M> NVMe Target support
  [*]   NVMe Target Passthrough support
  <M>   NVMe loopback device support
  <M>   NVMe over Fabrics FC target driver
  < >     NVMe over Fabrics FC Transport Loopback Test driver (NEW)
  <M>   NVMe over Fabrics TCP target support
```

Now go to File Systems and select support for the filesystems that will be used by the system. Do not compile the file system that is used for the root filesystem as module, otherwise the system may not be able to mount the partition. Also select *Virtual memory* and */proc file system*. Select one or more of the following options as needed by the system:

KERNEL **Enable file system support (*CONFIG_EXT2_FS*, *CONFIG_EXT3_FS*, *CONFIG_EXT4_FS*, *CONFIG_BTRFS_FS*, *CONFIG_XFS_FS*, *CONFIG_MSDOS_FS*, *CONFIG_VFAT_FS*, *CONFIG_PROC_FS*, and *CONFIG_TMPFS*)**

```
File systems --->
  <*> Second extended fs support
  <*> The Extended 3 (ext3) filesystem
  <*> The Extended 4 (ext4) filesystem
  <*> Btrfs filesystem support
  <*> XFS filesystem support
  DOS/FAT/NT Filesystems  --->
    <*> MSDOS fs support
    <*> VFAT (Windows-95) fs support
  Pseudo Filesystems --->
    [*] /proc file system support
    [*] Tmpfs virtual memory file system support (former shm fs)
```

If PPPoE is used to connect to the Internet, or a dial-up modem, then enable the following options (*CONFIG_PPP*, *CONFIG_PPP_ASYNC*, and *CONFIG_PPP_SYNC_TTY*):

KERNEL **Enabling PPPoE support (*PPPoE*, *CONFIG_PPPOE*, *CONFIG_PPP_ASYNC*, *CONFIG_PPP_SYNC_TTY***

```
Device Drivers --->
  Network device support --->
    <*> PPP (point-to-point protocol) support
    <*> PPP over Ethernet
    <*> PPP support for async serial ports
    <*> PPP support for sync tty ports
```

The two compression options won't harm but are not definitely needed, neither does the PPP over Ethernet option, that might only be used by ppp when configured to do kernel mode PPPoE.

Don't forget to include support in the kernel for the network (Ethernet or wireless) cards.

Most systems also have multiple cores at their disposal, so it is important to activate *Symmetric multi-processing support* (*CONFIG_SMP*):

KERNEL **Activating SMP support (*CONFIG_SMP*)**

```
Processor type and features  --->
  [*] Symmetric multi-processing support
```

Note
In multi-core systems, each core counts as one processor.

If USB input devices (like keyboard or mouse) or other USB devices will be used, do not forget to enable those as well:

KERNEL **Enable USB and human input device support (*CONFIG_HID_GENERIC*, *CONFIG_USB_HID*, *CONFIG_USB_SUPPORT*, *CONFIG_USB_XHCI_HCD*, *CONFIG_USB_EHCI_HCD*, *CONFIG_USB_OHCI_HCD*, *(CONFIG_HID_GENERIC, CONFIG_USB_HID, CONFIG_USB_SUPPORT, CONFIG_USB_XHCI_HCD, CONFIG_USB_EHCI_HCD, CONFIG_USB_OHCI_HCD, CONFIG_USB4)***

```
Device Drivers --->
  HID support  --->
    -*- HID bus support
    <*>   Generic HID driver
    [*]   Battery level reporting for HID devices
      USB HID support  --->
        <*> USB HID transport layer
  [*] USB support  --->
    <*>     xHCI HCD (USB 3.0) support
    <*>     EHCI HCD (USB 2.0) support
    <*>     OHCI HCD (USB 1.1) support
  <*> Unified support for USB4 and Thunderbolt  --->
```

Optional: Signed kernel modules

To automatically sign the kernel modules enable *CONFIG_MODULE_SIG_ALL*:

KERNEL **Sign kernel modules *CONFIG_MODULE_SIG_ALL***

```
[*] Enable loadable module support
  -*-   Module signature verification
    [*]     Automatically sign all modules
    Which hash algorithm should modules be signed with? (Sign modules with SHA-512) --->
```

Optionally change the hash algorithm if desired.

To enforce that all modules are signed with a valid signature, enable *CONFIG_MODULE_SIG_FORCE* as well:

KERNEL **Enforce signed kernel modules *CONFIG_MODULE_SIG_FORCE***

```
[*] Enable loadable module support
  -*-   Module signature verification
    [*]     Require modules to be validly signed
    [*]     Automatically sign all modules
    Which hash algorithm should modules be signed with? (Sign modules with SHA-512) --->
```

To use a custom key, specify the location of this key in *CONFIG_MODULE_SIG_KEY*. If unspecified, the kernel build system will generate a key. It is recommended to generate one manually instead. This can be done with:

```
root # openssl req -new -nodes -utf8 -sha256 -x509 -outform PEM -out kernel_key.pem -keyout kernel_key.pem
```

OpenSSL will ask some questions about the user generating the key, it is recommended to fill in these questions as detailed as possible.

Store the key in a safe location, at the very least the key should be readable only by the root user. Verify this with:

```
root # ls -l kernel_key.pem
```

```
 -r-------- 1 root root 3164 Jan  4 10:38 kernel_key.pem
```

If this outputs anything other then the above, correct the permissions with:

```
root # chown root:root kernel_key.pem
root # chmod 400 kernel_key.pem
```

KERNEL **Specify signing key *CONFIG_MODULE_SIG_KEY***

```
-*- Cryptographic API  --->
  Certificates for signature checking  --->
    (/path/to/kernel_key.pem) File name or PKCS#11 URI of module signing key
```

To also sign external kernel modules installed by other packages via `linux-mod-r1.eclass` , enable the `modules-sign (https://packages.gentoo.org/useflags/modules-sign)` (/wiki/USE_flag) USE flag globally:

FILE **`/etc/portage/make.conf`** **Enable module signing**

```
USE="modules-sign"

# Optionally, when using custom signing keys.
MODULES_SIGN_KEY="/path/to/kernel_key.pem"
MODULES_SIGN_CERT="/path/to/kernel_key.pem" # Only required if the MODULES_SIGN_KEY does not also contain the certificate
MODULES_SIGN_HASH="sha512" # Defaults to sha512
```

Note
`MODULES_SIGN_KEY` and **`MODULES_SIGN_CERT`** may point to different files. For this example, the pem file generated by OpenSSL includes both the key and the accompanying certificate, and thus both variables are set to the same value.

Optional: Signing the kernel image (Secure Boot)

When signing the kernel image (for use on systems with Secure Boot (/wiki/Secure_Boot) enabled) it is recommended to set the following kernel config options:

KERNEL **Lockdown for secureboot**

```
General setup  --->
  Kexec and crash features  --->
    [*] Enable kexec system call
    [*] Enable kexec file based system call
    [*]   Verify kernel signature during kexec_file_load() syscall
    [*]     Require a valid signature in kexec_file_load() syscall
    [*]     Enable ""image"" signature verification support

[*] Enable loadable module support
  -*-   Module signature verification
    [*]     Require modules to be validly signed
    [*]     Automatically sign all modules
    Which hash algorithm should modules be signed with? (Sign modules with SHA-512) --->

Security options  --->
[*] Integrity subsystem
  [*] Basic module for enforcing kernel lockdown
  [*]   Enable lockdown LSM early in init
        Kernel default lockdown mode (Integrity)  --->

[*]   Digital signature verification using multiple keyrings
  [*]     Enable asymmetric keys support
  -*-       Require all keys on the integrity keyrings be signed
  [*]       Provide keyring for platform/firmware trusted keys
  [*]       Provide a keyring to which Machine Owner Keys may be added
  [ ]         Enforce Machine Keyring CA Restrictions
```

Where ""image"" is a placeholder for the architecture specific image name. These options, from the top to the bottom: enforces that the kernel image in a kexec call must be signed (kexec allows replacing the kernel in-place), enforces that kernel modules are signed, enables lockdown *integrity* mode (prevents modifying the kernel at runtime), and enables various keychains.

On arches that do not natively support decompressing the kernel (e.g. **`arm64`** and **`riscv`**), the kernel must be built with its own decompressor (zboot):

KERNEL **zboot *CONFIG_EFI_ZBOOT***

```
Device Drivers --->
  Firmware Drivers --->
    EFI (Extensible Firmware Interface) Support --->
      [*] Enable the generic EFI decompressor
```

After compilation of the kernel, as explained in the next section, the kernel image must be signed. First install `app-crypt/sbsigntools` (https://packages.gentoo.org/packages/app-crypt/sbsigntools) and then sign the kernel image:

```
root # emerge --ask app-crypt/sbsigntools
root # sbsign /usr/src/linux-x.y.z/path/to/kernel-image --cert /path/to/kernel_key.pem --key /path/to
/kernel_key.pem --out /usr/src/linux-x.y.z/path/to/kernel-image
```

Note
For this example, the same key that was generated to sign the modules is used to sign the kernel image. It is also possible to generate and use a second separate key for signing the kernel image. The same OpenSSL command as in the previous section may be used again.

Then proceed with the installation.

To automatically sign EFI executables installed by other packages, enable the `secureboot (https://packages.gentoo.org/useflags/secureboot)` `(/wiki/USE_flag)` USE flag globally:

FILE **`/etc/portage/make.conf`** **Enable Secure Boot**

```
USE="modules-sign secureboot"

# Optionally, to use custom signing keys.
MODULES_SIGN_KEY="/path/to/kernel_key.pem"
MODULES_SIGN_CERT="/path/to/kernel_key.pem" # Only required if the MODULES_SIGN_KEY does not also contain the
certificate.
MODULES_SIGN_HASH="sha512" # Defaults to sha512

# Optionally, to boot with secureboot enabled, may be the same or different signing key.
SECUREBOOT_SIGN_KEY="/path/to/kernel_key.pem"
SECUREBOOT_SIGN_CERT="/path/to/kernel_key.pem"
```

Note

SECUREBOOT_SIGN_KEY and **SECUREBOOT_SIGN_CERT** may point to different files. For this example, the pem file generated by OpenSSL includes both the key and the accompanying certificate, and thus both variables are set to the same value.

Note
When generating an Unified Kernel Image (/wiki/Unified_Kernel_Image) with systemd's `ukify` the kernel image will be signed automatically before inclusion in the unified kernel image and it is not necessary to sign it manually.

Architecture specific kernel configuration

Make sure to select IA32 Emulation and 32-bit time_t if 32-bit programs should be supported (*CONFIG_IA32_EMULATION* and *CONFIG_COMPAT_32BIT_TIME*). Gentoo installs a multilib system (mixed 32-bit/64-bit computing) by default, so unless a no-multilib profile is used, these options are required.

KERNEL **Selecting processor types and features**

```
Processor type and features  --->
   [ ] Machine Check / overheating reporting
   [ ]   Intel MCE Features
   [ ]   AMD MCE Features
   Processor family (AMD-Opteron/Athlon64)  --->
      ( ) Opteron/Athlon64/Hammer/K8
      ( ) Intel P4 / older Netburst based Xeon
      ( ) Core 2/newer Xeon
      ( ) Intel Atom
      ( ) Generic-x86-64
Binary Emulations --->
   [*] IA32 Emulation
General architecture-dependent options  --->
   [*] Provide system calls for 32-bit time_t
```

Enable GPT partition label support if that was used previously when partitioning the disk (*CONFIG_PARTITION_ADVANCED* and *CONFIG_EFI_PARTITION*):

KERNEL **Enable support for GPT**

```
-*- Enable the block layer --->
   Partition Types --->
      [*] Advanced partition selection
      [*] EFI GUID Partition support
```

Enable EFI stub support, EFI variables and EFI Framebuffer in the Linux kernel if UEFI is used to boot the system (*CONFIG_EFI*, *CONFIG_EFI_STUB*, *CONFIG_EFI_MIXED*, *CONFIG_EFI_VARS*, and *CONFIG_FB_EFI*):

KERNEL **Enable support for UEFI**

```
Processor type and features  --->
    [*] EFI runtime service support
    [*]   EFI stub support
    [*]     EFI mixed-mode support

Device Drivers
    Graphics support  --->
        Frame buffer Devices  --->
            <*> Support for frame buffer devices  --->
                [*]   EFI-based Framebuffer Support

File Systems
    Pseudo filesystems  --->
        <*> EFI Variable filesystem
```

To enable the Kernel options for the use of SOF Firmware (/wiki/Handbook:AMD64/Installation/Kernel#SOF_Firmware) covered earlier:

KERNEL **Enabling SOF Firmware support (*CONFIG_SND_SOC_SOF_TOPLEVEL*, *CONFIG_SND_SOC_SOF_PCI*, *CONFIG_SND_SOC_SOF_ACPI*, *CONFIG_SND_SOC_SOF_AMD_TOPLEVEL*, *CONFIG_SND_SOC_SOF_INTEL_TOPLEVEL*)**

```
Device Drivers --->
  Sound card support --->
    Advanced Linux Sound Architecture --->
      <M> ALSA for SoC audio support --->
        [*] Sound Open Firmware Support --->
            <M> SOF PCI enumeration support
            <M> SOF ACPI enumeration support
            <M> SOF support for AMD audio DSPs
            [*] SOF support for Intel audio DSPs
```

Compiling and installing

With the configuration now done, it is time to compile and install the kernel. Exit the configuration and start the compilation process:

```
root # make && make modules_install
```

Note
It is possible to enable parallel builds using **`make -jX`** with `X` being an integer number of parallel tasks that the build process is allowed to launch. This is similar to the instructions about **`/etc/portage/make.conf`** earlier, with the *MAKEOPTS* variable.

When the kernel has finished compiling, copy the kernel image to **`/boot/`**. This is handled by the **`make install`** command:

```
root # make install
```

This command will copy the kernel image to **`/boot`**. If `sys-kernel/installkernel` (https://packages.gentoo.org/packages/sys-kernel/installkernel) is installed it will call **`/sbin/installkernel`** instead and delegate the kernel installation. Instead of simply copying the kernel to **`/boot`**, Installkernel (/wiki/Installkernel) installs each kernel with its version number in the file name. Additionally, installkernel provides a framework for automatically accomplishing various tasks relating to kernel installation, such as: generating an initramfs (/wiki/Initramfs), building an Unified Kernel Image (/wiki/Unified_Kernel_Image), and updating the bootloader (/wiki/Bootloader) configuration.

Deprecated: Genkernel

Genkernel should only be considered by users with a required need that only Genkernel can meet. For others, it is recommended to use the Distribution kernel or manually compile their own as it will make maintaining a Gentoo system a lot more simple. An example of why **`genkernel`** is more difficult to manage is the lack of integration with `sys-kernel/installkernel` (https://packages.gentoo.org/packages/sys-kernel/installkernel). This means a user will not get the same level of automation as provided by the other methods; for example, Unified Kernel Images will need to be created manually when using Genkernel.

Users still wishing to use Genkernel should see the Genkernel (/wiki/Genkernel) article for more information.

Kernel modules

Listing available kernel modules

Note
Hardware modules are optional to be listed manually. **`udev`** will normally load all hardware modules that are detected to be connected in most cases. However, it is not harmful for modules that will be automatically loaded to be listed. Modules cannot be loaded twice; they are either loaded or unloaded. Sometimes exotic hardware requires help to load their drivers.

The modules that need to be loaded during each boot in can be added to **`/etc/modules-load.d/*.conf`** files in the format of one module per line. When extra options are needed for the modules, they should be set in **`/etc/modprobe.d/*.conf`** files instead.

To view all modules available for a specific kernel version, issue the following **`find`** command. Do not forget to substitute "<kernel version>" with the appropriate version of the kernel to search:

```
root # find /lib/modules/<kernel version>/ -type f -iname '*.o' -or -iname '*.ko' | less
```

Force loading particular kernel modules

To force load the kernel to load the **`3c59x.ko`** module (which is the driver for a specific 3Com network card family), edit the **`/etc/modules-load.d/network.conf`** file and enter the module name within it.

```
root # mkdir -p /etc/modules-load.d
root # nano -w /etc/modules-load.d/network.conf
```

Note that the module's **`.ko`** file suffix is insignificant to the loading mechanism and left out of the configuration file:

FILE **`/etc/modules-load.d/network.conf`** **Force loading 3c59x module**

```
3c59x
```

Continue the installation with Configuring the system (/wiki/Handbook:AMD64/Installation/System).

← Installing base system (/wiki/Handbook:AMD64/Installation/Base) Home (/wiki/Handbook:AMD64) Configuring the system → (/wiki/Handbook:AMD6

Retrieved from "https://wiki.gentoo.org/index.php?title=Handbook:AMD64/Installation/Kernel&oldid=212420 (https://wiki.gentoo.org /index.php?title=Handbook:AMD64/Installation/Kernel&oldid=212420)"

- This page was last edited on 2 January 2015, at 00:03.
- Privacy policy (/wiki/Gentoo_Wiki:Privacy_policy)
- About Gentoo Wiki (/wiki/Gentoo_Wiki:About)
- Disclaimers (/wiki/Gentoo_Wiki:General_disclaimer)