Building External Modules

Contents

Introduction

There may come time in life when one would find benefit in compiling kernel modules outside of the running kernel. This is known as compiling external or out of tree modules. In the case of `LiME`, compiling outside of the running kernel is a more forensically sound and secure method, as the kernel object is not compiled on the target system. Since there is no need to compile on the target systems, Admin's do not have to alter the production systems to include gcc, linux kernel headers, among other development tools. This link includes the kernel documentation on how to build an external kernel module.

NOTE This guide does not cover `cross compiling` external modules. If your architecture differs from your host machine you will need to cross compile your module.

How to

The following is a step-by-step guide, using Ubuntu, in order to compile your own external module. The steps will vary from each distribution. Some distribution specifics will be covered at the end of this document.

Required tools

You will need the following tools

- git
- build-essential package OS specific

Downloading the kernel source

The first task is to find and download the correct kernel source for your distribution and version. For this I will show you examples with the Ubuntu kernel. You can read Ubuntu's fancy guide for downloading source here.

But here is the TL;DR version

In order to determine the correct OS version of your target machine, you can run `cat /etc/os-release`.

```
$ cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.2 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.2 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

From the output above we can see that our release is `Xenial 16.04.2 LTS`.

Now we must go and clone the source using `git`. For our Ubuntu example the links are in the following format.

```
kernel.ubuntu.com/ubuntu/ubuntu-< release >.git
```

Following our Xenial example, we would clone the source by entering this

```
$git clone git://kernel.ubuntu.com/ubuntu/ubuntu-xenial.git
```

If you have firewall restrictions or other ridiculousness using the git protocol, you can clone via http.

```
$git clone http://kernel.ubuntu.com/git-repos/ubuntu/ubuntu-xenial.git
```

This will be a lot slower and you will not be able to set the history depth, therefore downloading far more data.

Choosing the correct kernel release

Once the repository has finished cloning, we will need to checkout the correct kernel release. To complete this task run `uname -r` on the target machine.

```
$uname -r
4.10.0-38-generic
```

This most important take-away of the kernel release is the string after the `sublevel` digit. The Linux kernel is versioned in the following format

```
version.patchlevel.sublevel-localversion
```

From the example above we can see that our local version needs to be `-38-generic`. Once you have determined the version that you need to build, change directory into your kernel source. From this location run

```
git tag -l
```

to list all the tags. Find the tag that matches you kernel version *version.patchlevel.sublevel* and checkout that the point in history.

```
git checkout < tag >
```

Following our Ubuntu guide you would run something like the following

```
git checkout Ubuntu-lts-4.10.0-9.11_16.04.2
```

Using an old kernel config

In order to build an external modules that will fit target running kernel, we need to know how your kernel was built. The kernel build process stores this information in a config file, storing that in `/boot/config-*`.
Copy the correct config file to your kernel working directory and then rename it to `.config`. In our Ubuntu example the correct config file is located/called

```
/boot/config-4.10.0-38-generic
```

Once you have renamed the config file `.config` run the following

```
$ make olddefconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
scripts/kconfig/conf  --olddefconfig Kconfig
#
# configuration written to .config
#
```

This make function will use the old kernel config and substitute the default values for options that differ in your kernel.

Setting the correct version

This is the most important part of the entire process. If the version does not match the running kernel, your module will most likely fail to install. This is due to a kernel safety measure, enabled by default, to prevent incompatible modules from loading.
Once the config completes, we need to make sure that all the versions match before we continue. Run the following make function

```
$ make kernelrelease
```

```
4.10.0+
```

Did make complete without error? Does that match the version you want? If so continue; else checkout a different tag with git.

Did you notice that our kernel release is missing the `localversion` string? Well, let's fix that using your favorite text editor.
Find the lines that say the following

```
#
# General setup
#
CONFIG_INIT_ENV_ARG_LIMIT=32
CONFIG_CROSS_COMPILE=""
# CONFIG_COMPILE_TEST is not set
CONFIG_LOCALVERSION=""
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_HAVE_KERNEL_GZIP=y
```

Change both `CONFIG_LOCALVERSION` and `# CONFIG_COMPILE_TEST is not set` to match the following example

```
CONFIG_LOCALVERSION="< localversion >"
CONFIG_LOCALVERSION_AUTO=n
```

In our Ubuntu example add `-38-generic` and don't forget the hyphen.

```
CONFIG_LOCALVERSION="-38-generic"
CONFIG_LOCALVERSION_AUTO=n
```

Now! run `make kernelrelease` again

```
$ make kernelrelease
4.10.0-38-generic+
```

Is your localversion correct? If so, continue
Note the `+` at the end of the localversion string. We need to remove this

```
touch .scmversion
```

to create and empty file. Now run `make kernelrelease` once more, this time the version should be an exact match.

```
$ make kernelrelease
4.10.0-38-generic
```

Prepare the source and compile

Now run

```
$ make modules_prepare
```

in order to prepare the kernel source tree for building external modules. We use this function in order to skip compiling an entire kernel, saving you some cycles. If this completes without error, one can proceed with compiling the module. We will use LiME as the example module. Change directory into your LiME source and run

```
make -C < path-src-tree > KVER=< kernel-version > M=$(pwd)
```

`path-to-src-tree` is the location where you cloned your kernel source. Again, following our Ubuntu example

```
make -C /home/kd8bny/ubuntu-xenial KVER=4.10.0-38-generic M=$(pwd)
```

And there you have it! A successfully compiled external kernel module. Now feel free to load this into the running kernel on your target machine.

OS specific resources

CentOS

CentOS and RHEL package source a little differently. The source is packaged as an RPM. This

is a semi-helpful link.
The source is located here. Browse to the following location and download.

```
http://vault.centos.org/< cent version >/os/Source/SPackages/kernel-3.10.0-123.el7.
```

Once you have downloaded the RPM extract it using `tar` or some other file archiving tool. Once extracted you will see another archive dubbed `linux-<version>`. This is your source, extract it. You can use the config files already found in this source. Continue as stated in the guide, ignoring the use of `git`.

Fedora

Fedora keeps kernel source off the main linux git tree. Clone it here

```
git://git.kernel.org/pub/scm/linux/kernel/git/jwboyer/fedora.git
```

Follow the same process in the guide.

RHEL

Follow the centOS section, as this is where the source is located for non-subscribers. If you are a subscriber, you can download the source from Red Hat.

Ubuntu

Follow as shown in guide.
kernel source