Kernel Module Packages Manual
for CODE 11

June 25, 2013

1 Overview

SUSE Linux distributions use the RPM Package Manager for software management. As such, any SUSE/SLES external kernel modules (i.e., kernel modules not included in kernel packages) should be packaged in rpms. These rpms should be built in accordance with specific guidelines to ensure that the resulting Kernel Module Packages (KMPs) can be installed and updated appropriately, in sync with kernel updates.

This document specifies the requirements for rpm packages that contain kernel modules, and describes the processes surrounding those packages including building, signing, installing and upgrading. A complete example is given and explained.

2 Scope

This version of the Kernel Module Packages Manual applies to the SUSE CODE 11 code base, which includes openSUSE 11.1 and newer, SLES 11 (along with service packs), and all products based on SLES 11. Versions of this document for CODE 9 and CODE 10 are available as well [9].

This document's Appendix B highlights CODE 10 → CODE 11 changes as well as secure boot changes for SLE 11 SP3.

3 Background

The Linux kernel supports adding functionality at runtime through kernel loadable modules. It includes more than 1500 modules, about 75 percent of which are hardware drivers. These modules are shipped as part of the kernel packages. In some cases it is desirable to add additional modules or replace existing ones. (For example, a driver for a particular storage controller that was not available at the time of product release might be added later in order to support new hardware.)

Kernel modules interact with the kernel by the means of exported symbols, in a way similar to how user-space binaries use shared libraries.[1] To ensure that the kernel and modules refer to the same symbols, a version checksum (aka modversion) is added to each symbol. The checksum is computed from the symbol's type: in the case of function symbols, the checksum is determined by the function's parameters and return type.

When any of a function's parameters or the return type changes, the checksum changes as well. This includes all the data types involved recursively: if a function takes a struct task_struct as parameter and struct task_struct includes a field of type struct dentry, then a change to struct dentry will cause the symbol's version checksum to change as well. Symbol version checksums for different kernel flavors (e.g., kernel-pae vs. kernel-xen) will not match, and symbol versions of the same kernel package on different architectures (e.g., kernel-default on i386 vs. x86_64) will not match, either. This mechanism ensures that the kernel and kernel modules agree on the types of data structures that they use to communicate.

Unless symbol version checking is disabled, modules will load only if the checksums of the symbols they use match the checksums of the symbols that the kernel exports. The exported symbols and their version checksums comprise the kernel Application Binary Interface (kABI). When an updated kernel includes kABI changes, kernel modules that use any modified symbols must be updated as well.

1 The /proc/kallsyms file lists all symbols currently known to the kernel.

During its multi-year life cycle, products like SUSE Linux Enterprise Server (SLES) undergo continuous changes, and different kinds of updates like Service Packs (SPs), maintenance/security updates, and customer-specific updates (PTFs=Program Temporary Fixes) are released. The Application Binary Interface (ABI) between the kernel and kernel modules is volatile. Some kernel updates will change the kernel ABI (kABI) by adding or removing exported symbols, or existing symbol checksums may change in a kernel update because of changes in data structures they reference. We strive to keep the kernel ABI stable in maintenance/security and customer-specific updates, but sometimes we cannot avoid changes. In Service Packs, we reserve the right to introduce more intrusive changes, which increases the likelihood of ABI changes. We believe that the added flexibility outweighs the disadvantages of breaking older modules. Please see kABI Stability in SLES [1] and The Linux Kernel Driver Interface [2] for full discussions of this topic.

SUSE Linux operating systems include technology to ensure that kernel modules can be reused or updated in sync with kernel updates. To make use of this technology, kernel modules must be packaged into Kernel Module Packages (KMPs) as defined in this document.

4 Kernel Packages

All SUSE products based on 2.6.x and 3.0.x kernels contain a set of kernel packages that share the same version and release number; they are built from the same kernel sources. These packages are:

kernel-*flavor*

> A binary kernel package. Each architecture has its own set of kernel flavors (e.g., kernel-pae, kernel-default, kernel-xen, etc.) These are the packages that the kernel modules will be used with. Note: In CODE 11, the binary kernel is provided by two packages: kernel-*flavor*-base and kernel-*flavor*.

kernel-*source*

> The kernel source tree, generated by unpacking the vanilla kernel sources and applying all necessary patches. While the kernel-*flavor* packages technically are not built from the kernel-source package, they are built from the same source tree. This tree should be used for module building.

kernel-*syms*

> Kernel symbol version information for compiling external modules. This package is *required* for building external modules. If this package is not used, the resulting modules will be missing symbol version information, which will cause them to break during kernel updates. The kernel-source and kernel-syms packages used for compiling external modules must match each other exactly.
>
> Starting with SLE 11 SP1, the kernel-syms package is actually just a place-holder package which depends on the kernel-*flavor*-devel packages for all kernel flavors.

Please refer to Working With The SUSE 2.6.x Kernel Sources [3] for more information.

5 Kernel Modules

Documentation on general kernel module building can be found in abundance on the Internet [5,6]. SUSE

specific information is found in Working With The SUSE 2.6.x Kernel Sources [3].

Once built, kernel module binaries are installed below /lib/modules/*version-release-flavor* on the file system (e.g., /lib/modules/2.6.27.8-1-pae for the kernel-pae-2.6.27.8-1 package). Different kernels have different module directories, and will not usually see each other's modules.

Update modules are modules intended to replace or augment the modules that are provided in the kernel packages. Update modules must be stored below the /lib/modules/*version-release-flavor*/updates/ directory. Modules in the updates/ directory have precedence over other modules with the same name. Never replace modules from the kernel package by overwriting files: this would lead to inconsistencies between the file system and the rpm database.

Note that while modules intended to take precedence over in-kernel modules of the same name should be stored below /lib/modules/*version-release-flavor*/updates/, other add-on modules may be stored below /lib/modules/*version-release-flavor*/extra/.

Modules usually remain compatible with a range of kernel releases (i.e. package updates). To make such modules visible to other kernel releases, symbolic links to compatible modules are put in /lib/modules/*version-release-flavor*/weak-updates/ directories. Modules in the weak-updates/ directory have lower priority than modules in the updates/ directory, but higher priority than all other modules in /lib/modules/*version-release-flavor.* If more than one compatible module is available for a kernel, the module with the highest kernel release is chosen. Kernel Module Packages must never install modules into weak-updates/ directories.

Kernel modules must *never* be installed as individual files on a production system, and always as part of a Kernel Module Package.

6 Kernel Module Packages

SUSE has worked closely with the Linux Foundation Driver Backport Workgroup to establish a standard structure for building Kernel Module Packages for all RPM-based distributions. The information in this document includes the standards as appropriate.

Kernel Module Package spec files define a main package, and a sub-package for each kernel flavor supported. The kernel-flavor-specific sub-packages are defined with the %kernel_module_package rpm macro. The macro automatically determines for which kernel flavors to generate sub-packages. Several options are available to modify the macro's behavior, which are described below:

%kernel_module_package [-f filelist] [-p preamble] [-n name] [-v version] [-r release] [-t template] [-x flavor] [-b] [-c module-signing-certificate]

The main package of a Kernel Module Package can either contain no %files section, in which case rpm will not create a binary package with the main package's name, or it can be used for the user-space part associated with the kernel modules that end up in the kernel specific sub-packages. (The example Kernel Module Package in Appendix A has a main package without a %files section.)

Kernel Module Packages must adhere to the following rules:

- The package **Name** should consist of two components: a unique provider prefix, and a driver name. Hyphens are disallowed in the provider prefix, and allowed in the driver name. The provider prefix serves to create a

non-overlapping name space for all providers.

The sub-package names are composed of the main package name, followed by a dash, the string “kmp”, followed by another dash and the flavor of the supported kernel. The first component (main package name) can be overridden with a different value by using the **-n** option of the %kernel_module_package macro.

- The kernel module package **Version** can have an arbitrary value.

  The sub-package versions are composed of the main package version, followed by an underscore, and the version of the kernel source used during the build.

  Since sub-packages already include the supported kernel's flavor in their name, the flavor is not again included in the sub-package's version. Dashes in the kernel release are replaced by underscores. The first component (main package version) can be overridden with the **-v** option of the %kernel_module_package macro.

- The kernel module package **Release** can be assigned freely as required. It must be incremented at least once for each package release.

  The sub-package release numbers equal the main package's release number. It can be overridden with the **-r** option of the %kernel_module_package macro.

- The appropriate **Requires** and **Provides** tags are computed automatically by rpm as described in the RPM Provides and Requires section below. Requires and Provides tags in the spec file will only be effective for the main package.

- Kernel modules must be installed below /lib/modules/*version-release-flavor*/updates/.

- Packages must be signed with a public/private key pair, and the public key of the private/public key-pair used for signing must be made known to rpm. See the Signing Kernel Module Packages section below for details.

- If a kernel module package is intended to support UEFI Secure Boot, the modules in the package must be signed with a private key and the corresponding public key must be provided at package installation time. The macro's -c option provides a way to create a separate package that provides a UEFI certificate containing the required public key; other spec file changes can be included to actually sign the modules. The -c option is available starting with SLES 11 SP3.

The %description tag will be applied to both the main package and the sub-packages.

The %kernel_module_package macro uses a default sub-package template that should work for most Kernel Module Packages. This template can be overridden using the macro's -t option. The default template takes care of the following:

- When a Kernel Module Package package is installed, depmod is called to update module dependency information and various maps. Symlinks pointing at the new modules are created in other kernels' weak-modules/ directories for all compatible modules. Initial ramdisks used during booting are recreated automatically if they contain some of the added modules. Using the macro's -b option will force recreation

of the initial ramdisk regardless of whether or not the existing ramdisk contains modules with the same names as the modules being installed. (The -b option is available starting with SLES 11 SP2).

- When a Kernel Module Package is removed, depmod is called to update module dependency information and various maps. The symlinks pointing to the modules being removed are removed as well. Initial ramdisks are recreated in case they did contain some of the removed modules.

By default, each kernel-specific sub-package will have the following list of files, which can separately be overridden with the -f option::

```
%defattr (-,root,root)
/lib/modules/%2-%1
```

Additional sub-package preamble lines such as Requires, Provides, and Obsoletes tags can be specified with the -p option. Filename arguments specified in -f, -p and -t should be given as absolute path names (e.g., %_sourcedir/file) and should be listed as Sources. The following substitutions are defined in those files:

```
%1      Flavor of the sub-package (e.g., pae).
%2      Kernel release string without flavor (e.g., 2.6.27.8-1).
%{-v*} The sub-package version.
%{-r*} The sub-package release.
```

Some Kernel Module Packages may make sense only for some of the kernel flavors a given architecture supports. A list of flavors to exclude from the build should be passed with the -x option to the %kernel_module_package macro. The sample Kernel Module Package spec file in Appendix A excludes the debug and trace flavors.

Appendix A contains an example Kernel Module Package spec file as well as the source code referenced by it. When this spec file and its accompanying source is built into an i586 rpm as described in section Building Kernel Module Packages below, the BuildRequires tag in the spec file will pull the module-init-tools, kernel-syms and kernel-source packages into the build root. (Note that the %kernel_module_package_buildreqs macro does not need to explicitly list “kernel-source” since the kernel-syms package has a dependency on the kernel-source package.) Let us assume that the required packages are available in version/release 2.6.27.8-1, and that the debug, default, pae, trace, and xen kernel flavors are available on that platform. RPM would then create the following packages:

```
suse-hello-kmp-default-1.0_2.6.27.8_1.1-0.i586.rpm
suse-hello-kmp-pae-1.0_2.6.27.8_1.1-0.i586.rpm
suse-hello-kmp-xen-1.0_2.6.27.8_1.1-0.i586.rpm
```

The generated packages would contain the following modules, and require and provide the following symbols:

Package	**Requires**	**Provides**	**Modules**
suse-hello-kmp-default	kernel(default:kernel) = 81ee021907bb81cf	suse-hello-kmp = 1.0_2.6.27.8_1.1 ksym(default:exported_function) = e52d5bcf suse-hello-kmp-default = 1.0_2.6.27.8_1.1-0	/lib/modules/2.6.27.8_1.0-default/updates/hello.ko

suse-hello-kmp-pae	kernel(pae:kernel) = f9a733a55e8da41f	suse-hello-kmp = 1.0_2.6.27.8_1.1 ksym(pae:exported_function) = e52d5bcf suse-hello-kmp-pae = 1.0_2.6.27.8_1.1-0	/lib/modules/2.6.27.8_1.0-pae/updates/hello.ko
suse-hello-kmp-xen	kernel(xen:kernel) = 449a592a4f3b46a5	suse-hello-kmp = 1.0_2.6.27.8_1.1 ksym(xen:exported_function) = e52d5bcf suse-hello-kmp-xen = 1.0_2.6.27.8_1.1-0	/lib/modules/2.6.27.8_1.0-xen/updates/hello.ko

7 RPM Provides and Requires

As stated in the introduction, kernels export symbols that kernel modules use. Symbols have version checksums attached, and the checksums of the exported kernel symbols must match the checksums of the used kernel symbols. These dependencies are mapped to symbols that the kernel packages provide and that Kernel Module Packages require at the rpm package level.

Typical packages only require and/or provide a handful of symbols, so package managers are not designed to handle hundreds and thousands of package dependencies. Therefore, the kernel and Kernel Module Packages do not provide/require each kernel symbol individually. Instead, symbols are grouped together into classes. The classes are computed when the kernel packages are built; the number of classes (about 200) is much smaller than the number of symbols. The packages then provide and require these symbol classes instead of individual symbols.

When modules in Kernel Module Packages export additional symbols, those symbols are not in an existing class. Such symbols are mapped to per-symbol provides of those packages. Modules in other Kernel Module Packages may require those symbols; they would also do so on a per-symbol basis.

As an example, assume that the kernel-pae-base package provides the following symbol (among others):

kernel(pae:kernel) = f9a733a55e8da41f

and that the kernel-pae package provides the following symbol (among others):

kernel(pae:drivers_net) = 9e3d2b248b3ec097

A matching Kernel Module Package which uses symbols from these two symbol groups would then require kernel(pae:kernel) and kernel(pae:drivers_net) in the same versions. The Kernel Module Package might provide exported functions as ksym(pae:exported_function) = e52d5bcf or similar.

8 Building Kernel Module Packages

In addition to the C and kernel programming skills required for writing the kernel module source code in the first place, creating proper Kernel Module Packages requires some familiarity with rpm and with build environments. Those who are looking for more information on kernel module building may find the Linux Kernel Module Programming Guide [5] and the Linux Device Drivers book [6] interesting. Additional SUSE-specific kernel and kernel module information can be found in Working with the SUSE 2.6.x Kernel Sources [3]. We

recommend using the example package found in Appendix A as a template to reduce the complexities related to rpm. A lot of additional information on rpm can be found at http://www.rpm.org/, including an online version of the excellent Maximum RPM.

We strongly recommend using the kernel build infrastructure (kbuild) for building and installing the kernel modules, as done in the example package. Kbuild is documented in /usr/src/linux/Documentation/kbuild/ from the kernel-source package. Trying to emulate kbuild will lead to various problems including mis-compilations and missing or wrong symbol versions, and increased support load due to subtle breakages.

In order to achieve consistent and reproducible builds in a defined environment independent of the software installed on the system used for building, we recommend using the build script from the build.rpm package. This script sets up a build environment from the rpm packages the script is pointed at. The packages are then built in this environment using chroot (see the chroot(1) manual page). All SUSE packages are built using the same mechanism. When building Kernel Module Packages with build.rpm, the following options of the build script are particularly relevant:

--root *directory*

> Define the directory in which to set up the build environment. Defaults to the BUILD_ROOT environment variable, and to /var/tmp/build-root if unset.

--rpms *path1*[:*path2*:...]

> Define where build will look for packages for constructing the build environment. The directories are searched recursively. Packages found earlier in the path have precedence over packages found later, similar to how the PATH environment variable works. Defaults to the BUILD_RPMS environment variable, and to /media/dvd/suse if unset. The --rpms option must only be specified once.

--clean, --no-init

> Reconstruct the build environment entirely from scratch (--clean), or start the build without initializing the build environment (--no-init), which skips checking whether all packages in the build environment are up-to-date.

Build stores the created packages below usr/src/packages/ in the build environment.

On dual-architecture machines, packages for the other supported architecture can be built by running the build script inside an architecture selector. On x86_64, the selector is called linux32, on ppc64 this is ppc32, and on s390x the selector is called s390. The same build environment cannot be reused for different architectures unless it is reinitialized with build's --clean option.

See the build(1) manual page for further information.

> **Please Note:** For building external modules, you need to have both the kernel-source and kernel-<flavor>-devel packages installed in the build environment. The BuildRequires line in spec files takes care of this: the %kernel_module_package_buildreqs macro specifies the kernel-syms package, which pulls in the kernel-source package and the kernel-<flavor>-

devel packages due to its dependency on them. Note that without the kernel-syms the module build may still succeed depending on how you do the build, but the resulting modules will have module symbol versions disabled. Kernel Module Packages without module symbol versions will appear to match any kernel while in fact they do not. This can easily lead to very hard-to-diagnose system malfunctions.

9 Signing

Signing (as applied to a piece of software) is the process of digitally tagging the software to verify the author and guarantee that the software has not been altered since it was signed. SLES and SLED include utilities to sign and validate signatures on packages and repositories. In addition, SLES and SLED 11 SP3 include technology to sign and validate signatures on kernel modules.

The following sections describe how to sign packages and kernel modules. The topic of repository signing is beyond the scope of this document.

Signing Packages

All packages that are provided in SLES and SLED are digitally signed with the SUSE Build key. PLDP packages that are built on the PLDP Build Server (by the SUSE PLDP team or by a partner) are automatically signed with the SUSE PLDP key (see http://drivers.suse.com/doc/pldp-signing-key.html). Partners who build and/or provide their own packages are encouraged to sign them with their own keys.

To sign packages using GNU Privacy Guard (GPG):

In order to sign packages, a private/public key pair must be installed on the GPG keyring of the signing user (see the --gen-key option in the gpg(1) manual page). Then the following command can be used to sign a package (replace *build@suse.com* with the identity that identifies your signing key):

```
$ rpm --eval “%define _signature gpg” \
      --eval “%define _gpg_name build@suse.com” \
      --addsign package.rpm
```

Note that a package can only be signed once. Another --addsign operation will replace an existing old signature, and will add the new one.

The public key used for signing must then be exported into a file with:

```
$ gpg --armor --export build >build-pubkey.txt
```

Then, import the key into the rpm database with:

```
$ rpm --import build-pubkey.txt
```

You can verify that both package signing and key import have succeeded with rpm's --checksig option (note the “gpg” in the output):

```
$ rpm --checksig package.rpm
package.rpm: (sha1) dsa sha1 md5 gpg OK
```

The public key exported to build-pubkey.txt must be delivered to customers in a way that they will trust. It must be imported into the rpm database on systems on which the signed packages are to be installed.

Signing Module Object Files *(UEFI Secure Boot)*

While using signed packages and other OS security features can secure an installed and running system, they cannot prevent system subversion before the OS has booted. In order to address pre-OS security concerns, the UEFI 2.2 Secure Boot specification[9] details a protocol to prevent the loading of bootloaders or kernels (including drivers) that are not signed with an approved digital key stored in the system firmware.

The UEFI Secure Boot specification allows for variation in implementation. A simple way to implement secure boot is to ensure that the base system (as provided by the system vendor) contains all the keys that will be used by the bootloader, the OS, and any drivers. But having the system vendor simply place all needed keys into the firmware is not a full solution, as it does not give appropriate control to the system user/owner. SUSE's secure boot implementation addresses this control issue by extending the secure-boot-enabled EFI shim loader to accept keys that have been approved by the system owner. Thus, if there is a need to load a module with an unrecognized key, the key can be added to the “approved key” database (reboot and system-owner approval required).

Note: If a module has *no* signature, it cannot be loaded on a SLES/SLED 11 SP3 secure-boot-enabled system. SLES/SLED 11 SP3 users who wish to load unsigned modules must disable secure boot.

Creating a Key and Certificate

There are a number of ways to sign modules, but all methods require providing a key and certificate. Official keys and certificates may be provided by an organization's security team or by build services (such as the PLDP Build Service). Developers/packagers can also generate their own keys and certificates for testing purposes. To create a key and certificate using the “ssh req” command:

```
export USER="your company name"
openssl req -new -x509 -newkey rsa:2048 -sha256 -keyout key.asc -out cert.der \
        -outform der -nodes -days 4745 -subj "/CN=$USER/"
```

The above sequence of commands will create a key.asc key file and a cert.der x509 certificate in the current working directory. Note that the “4745” option generates a certificate which will be valid for 13 years.

Signing an Existing KMP

The SLES/SLED 11 SP3 pesign-obs-integration package provides a “modsign-repackage” utility that can be used to sign kernel modules in an existing KMP or other rpm. modsign-repackage unpacks the original rpm, signs any included modules, re-creates the rpm, and also creates a second <name>-ueficert rpm that installs the certificate and calls the “mokutil” utility to enroll the public key. The re-packaged rpm will have a dependency on the <name>-ueficert rpm, ensuring that the cert will be installed at the same time as the module(s).

To use modsign-repackage with the key.asc key and cert.der certificate created above to repackage a ./suse-hello-kmp-default-1.0_3.0.76_0.11.1-0.x86_64.rpm package:

```
modsign-repackage -c ./cert.der -k ./key.asc ./suse-hello-kmp-default-1.0-3.0.76_0.11.1-0.x86_64.rpm
```

The above command will create the following directories and files in the current working directory:

```
./RPMS/
   x86_64/
      suse-hello-kmp-default-1.0-3.0.76_0.11.1-0.x86_64.rpm
      suse-hello-ueficert-1.0-0.x86_64.rpm
```

Signing Modules During Packaging

Signing modules as part of the packaging process requires making a few changes to the KMP spec file template. The spec file template in Appendix A has been updated to include these changes along with conditionals to ensure that the changes will apply only when building for SLES/SLED 11 SP3. The changes are:

a) List the certificate file as a %Source file. The top-level directory of the build structure (where the spec file is located) should include a both a private key file and a certificate file. The spec file should list the certificate as a %Source file. The spec file should **not** list the key file (since the private key should **not** be included in the source KMP).

Note: In order to be recognized by the kernel Makefile, the key file must be named “signing_key.priv” and the certificate file must be named “signing_key.x509”. The example above describes how to use the “openssl req” command to create a key.asc key file and a cert.der certificate file; to use these files at packaging-time, they should be renamed to “signing_key.priv” and “signing_key.x509”.

b) Add code to determine the build target's service-pack level. This is done by adding the “sles-release” (or “sled-release”) package to %BuildRequires and then defining a %sle_version macro based on the contents of the /etc/SuSE-release file.

c) For SLES 11 SP3 builds, invoke the "%kernel_module_package macro with the "-c %_sourcedir/signing_key.x509" option to specify the certificate (and thus the key) to use in signing. Note that using the kernel_module_package “-c” option does not cause any module signing; it simply ensures the creation of a <name>-ueficert package which installs the certificate and calls the “mokutil” utility to enroll the public key. The actual module signing is handled in the %install section of the spec file.

c) For SLES 11 SP3 builds, add %install section code to ensure that the modules will get signed. This is done by setting the CONFIG_MODULE_SIG_ALL kernel configuration parameter. When CONFIG_MODULE_SIG_ALL is set, the “make modules_install” step automatically includes module signing.

Note: The Appendix A sample spec file is designed to be used by developers/packagers who provide their own keys and certificates. Partners who use the PLDP Build Service will not need to provide keys and certificates and thus should use the spec file in Appendix A.1.

Installation of Secure-boot-enabled KMPs

As discussed above, secure-boot-enabled KMPs include an additional <name>-ueficert package to install the

certificate and enroll the public key. The <name>-kmp-<flavor> packages require the <name>-ueficert package.

After the <name>-ueficert package is installed, the system must be rebooted and the newly-enrolled key approved by the system owner before the key (and thus the signed modules) can be used.

Note that the “mokutil” utility can also be used on its own to view and manage keys in the key database.

10 Deploying Kernel Module Packages

Kernel Module Packages may be distributed on Driver Update Disks, as Add-on Products, or simply as standalone rpms. If a Kernel Module Package's driver is required in order to boot an installation kernel, the Kernel Module Package should be provided on a Driver Update Disk (DUD). Otherwise, we recommend providing Kernel Module Packages as Add-on Products complete with URL(s) for functioning update sites.

11 System Installation and Kernel Module Packages

Initial system installation is carried out by YaST from some installation media (CDs or DVDs, network locations, etc.) As noted above, support for additional hardware that the installation media do not provide can be added with Driver Update Disks. This is most important to enable hardware needed for booting, such as storage controllers.

Update media (aka Driver Update Disks) provide two kinds of modules: those which the kernel that runs the installation uses, and those which are installed onto the final target system. Both types of modules are provided by including Kernel Module Packages on the update media. In addition, update media may contain scripts which are run at specific times during the installation. The Update Media HOWTO [4] describes in more detail what SUSE Update Media must contain in order to work.

After the initial YaST installation, additional driver packages can be installed using any of the mechanisms for installing rpm packages (YaST Add-On Products, YaST Software Management, YaST Online Update, the rpm command, etc.) Note that the Add-on Product format supports the ability to register the system for an update site.

Please note that any drivers required for getting to and accessing the root file system must be part of the initial ramdisk (initrd). YaST will automatically include necessary kernel modules in the initrd created during installation, but when Kernel Module Packages are installed by hand or updated, this needs to be taken care of. Such drivers may also need to be added to the INITRD_MODULES variable in /etc/sysconfig/kernel.

12 Kernel Updates and Kernel Module Packages

After all software repositories that should be checked for updates have been added, the package manager will automatically detect when new kernel packages as well as new Kernel Module Packages become available. The dependencies between those packages will ensure that the installed kernel packages match the installed Kernel Module Packages.

13 Futures

Currently, SUSE Linux kernel packages provide checksums for about 200 symbol groups. Each group corresponds to some portion of exported kernel symbols. As noted in section RPM Provides and Requires

above, the kernel packages provide checksums for symbol groups rather actual symbols because, so far, it has been infeasible to dependency check the 7000 or so actual kernel symbols at an rpm level. However, SUSE is looking at options for how to increase the granularity of the symbol group checksums. The subject is also under discussion at a cross-vendor level in the LF Driver Backport Workgroup[8].

Appendix A: Sample Source for suse-hello Kernel Module Package

The following sample is described in the Kernel Module Packages section above. The spec file includes conditional code that will build secure-boot-enabled packages for SLES 11 SP3 and non-secure-boot-enabled packages for SLES 11 – SLES 11 SP2. The conditional code is shown in bold and may be removed if there is no need to build secure-boot-enabled packages (even for SLES 11 SP3).

When using the sample to build secure-boot-enabled packages, the build structure must also include signing_key.priv and signing_key.x509 files as described in the "Signing Modules During Packaging" section above. The signing_key.* files must be located in the same directory as the spec file.

A sample package similar to this one is available in the Open Build Service:
https://build.opensuse.org/package/show?package=modsign-example-1&project=home%3Amichal-m%3Amodsign-examples

suse-hello.spec

```
# norootforbuild

Name:           suse-hello
BuildRequires:  %kernel_module_package_buildreqs
# Required to support secure-boot:  Include sles-release in order to determine
# service-pack version
BuildRequires:  sles-release
License:        GPL
Group:          System/Kernel
Summary:        Sample Kernel Module Package
Version:        1.0
Release:        0
Source0:        %name-%version.tar.bz2
# Required to support secure-boot:  Include certificate named "signing_key.x509"
# Build structure should also include a private key named "signing_key.priv"
# Private key should not be listed as a source file
Source1:        signing_key.x509
BuildRoot:      %{_tmppath}/%{name}-%{version}-build

# Required to support secure-boot:  Determine service-pack level
%define sle_version %(tr '\\n' ' ' < /etc/SuSE-release | sed -rn 's/^SUSE Linux
Enterprise ([A-z]+) ([0-9]+).*PATCHLEVEL = ([0-9]+).*$/\\2\\3/p')

# Required to support secure-boot:  The -c option tells the macro to generate a
# suse-hello-ueficert supackage that enrolls the certificate
%if 0%{?sle_version} > 112
%kernel_module_package -x debug -x trace -c %_sourcedir/signing_key.x509
%else
%kernel_module_package -x debug -x trace
%endif

%description
This package contains the hello.ko module.
```

```
%prep
%setup
%if 0%{?sle_version} > 112
cp %_sourcedir/signing_key.* .
%endif
set -- *
mkdir source
mv "$@" source/
mkdir obj

%build
for flavor in %flavors_to_build; do
      rm -rf obj/$flavor
      cp -r source obj/$flavor
      make -C %{kernel_source $flavor} modules M=$PWD/obj/$flavor
done

%install
export INSTALL_MOD_PATH=$RPM_BUILD_ROOT
export INSTALL_MOD_DIR=updates
for flavor in %flavors_to_build; do
      # Required to support secure-boot:  By default, kernel modules are not
      # signed by make. The CONFIG_MODULE_SIG_ALL=y setting overrides this for
      # flavors with module signing enabled.
      unset CONFIG_MODULE_SIG_ALL
      if grep '^CONFIG_MODULE_SIG=y' %{kernel_source $flavor}/.config; then
            export CONFIG_MODULE_SIG_ALL=y
      fi
      make -C %{kernel_source $flavor} modules_install M=$PWD/obj/$flavor
done

%changelog
* Wed Apr 24 2013 - mmarek@suse.cz
- Sign the module by a supplied keypair.
* Tue Dec 22 2008 - andavis@suse.com
- Updated to reflect CODE 11 changes and LF standard spec file work.
* Sat Jan 28 2006 - agruen@suse.de
- Initial package.
```

The following two files should be compressed to form the suse-hello-1.0.tar.bz2 tarball referenced as Source0 in the suse-hello.spec file above.

suse-hello-1.0/Kbuild

```
obj-m       := hello.o
hello-y     += main.o
```

suse-hello-1.0/main.c

```
/*
 * main.c - A demo kernel module.
 *
```

```
 * Copyright (C) 2003, 2004, 2005, 2006
 * Andreas Gruenbacher <agruen@suse.de>, SUSE Labs
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation.
 *
 * A copy of the GNU General Public License can be obtained from
 * http://www.gnu.org/.
 */

#include <linux/module.h>
#include <linux/init.h>

MODULE_AUTHOR("Andreas Gruenbacher <agruen@suse.de>");
MODULE_DESCRIPTION("Hello world module");
MODULE_LICENSE("GPL");

int param;

module_param(param, int, 0);
MODULE_PARM_DESC(param, "Example parameter");

void exported_function(void)
{
        printk(KERN_INFO "Exported function called.\n");
}
EXPORT_SYMBOL_GPL(exported_function);

int __init init_hello(void)
{
        printk(KERN_INFO "Hello world.\n");
        return 0;
}

void __exit exit_hello(void)
{
        printk(KERN_INFO "Goodbye world.\n");
}

module_init(init_hello);
module_exit(exit_hello);
```

Appendix A.1: Sample spec file for use with SUSE Build Services

Packagers who use the Open Build Service (https://build.opensuse.org/) or the SUSE PLDP Build Service (https://partnerbuild.suse.com/) to create KMPs can take advantage of the fact that these build services provide keys and certificates for package and module signing. The following spec file can be used in the SUSE build services to build secure-boot-enabled KMPs for SLES/SLED 11 SP3 and non-secure-boot-enabled KMPs for SLES/SLED 11 – SLES/SLED SP2. The secure-boot conditional code is highlighted in bold.

A sample package similar to this one is available in the Open Build Service:
https://build.opensuse.org/package/show?package=modsign-example-2&project=home%3Amichal-m%3Amodsign-examples

suse-hello.spec

```
# The following line tells the buildservice to save the project certificate as
# %_sourcedir/_projectcert.crt

# needssslcertforbuild

Name:           suse-hello
BuildRequires:  %kernel_module_package_buildreqs
# Required to support secure-boot:  Include sles-release in order to determine
# service-pack version
BuildRequires:  sles-release
BuildRequires:  %kernel_module_package_buildreqs sles-release
License:        GPL
Group:          System/Kernel
Summary:        Sample Kernel Module Package
Version:        1.0
Release:        0
Source0:        %name-%version.tar.bz2
BuildRoot:      %{_tmppath}/%{name}-%{version}-build

# Required to support secure-boot:  Determine service-pack level
%define sle_version %(tr '\\n' ' ' < /etc/SuSE-release | sed -rn 's/^SUSE Linux
Enterprise ([A-z]+) ([0-9]+).*PATCHLEVEL = ([0-9]+).*$/\\2\\3/p')

# Required to support secure-boot:  The -c option tells the macro to generate a
# suse-hello-ueficert supackage that enrolls the certificate
# The _projectcert.crt certificate is provided by the build service
%if 0%{?sle_version} > 112
%kernel_module_package -x debug -x trace -c %_sourcedir/_projectcert.crt
%else
%kernel_module_package -x debug -x trace
%endif

%description
This package contains the hello.ko module.

%prep
%setup
```

```
%_sourcedir/signing_key.* .
set -- *
mkdir source
mv "$@" source/
mkdir obj

%build
for flavor in %flavors_to_build; do
      rm -rf obj/$flavor
      cp -r source obj/$flavor
      make -C %{kernel_source $flavor} modules M=$PWD/obj/$flavor
done

%install
export INSTALL_MOD_PATH=$RPM_BUILD_ROOT
export INSTALL_MOD_DIR=updates
for flavor in %flavors_to_build; do
      make -C %{kernel_source $flavor} modules_install M=$PWD/obj/$flavor
done

# The BRP_PESIGN_FILES variable must be set to a space separated list of
# directories or patterns matching files that need to be signed.  E.g., packages
# that include firmware files would set BRP_PESIGN_FILES='*.ko /lib/firmware'
export BRP_PESIGN_FILES='*.ko'

%changelog
* Wed Apr 24 2013 - mmarek@suse.cz
- Sign the module by a supplied keypair.
* Tue Dec 22 2008 - andavis@suse.com
- Updated to reflect CODE 11 changes and LF standard spec file work.
* Sat Jan 28 2006 - agruen@suse.de
- Initial package.
```

Appendix B – Code 10 → Code 11 Notes for Kernel Module Packagers

Following is a brief list of CODE 10->CODE 11 changes which may affect kernel module packagers:

Kernel packages:

- The CODE 11 kernel is provided via two packages: kernel-*flavor*-base and kernel-*flavor*. Kernel-*flavor* is dependent on kernel-*flavor*-base. Both packages are required to provide checksums for all the kernel symbol groups.

- CODE 11 automatically installs the -pae kernel flavor for 32-bit pae-enabled systems.

- The CODE 11 kernel-source package does not include the sample Kernel Module Package source provided in Appendix A.

Kernel module packages:

- Starting with SLES/SLED 11 SP3, kernel module packages may contain modules that have been signed in order to support UEFI Secure Boot.

Package and file locations:

- In CODE 11, a number of packages have been moved from the base SLES product to the SLES SDK. Developers building Kernel Module Packages on CODE 11 will need to include the SDK in their build environment in order to ensure that all build-time package dependencies are resolved.

- In CODE 11, the rpm macros used to build Kernel Module Packages are provided in the kernel-source package (instead of the rpm package) and installed under /etc/rpm as well as /usr/lib/rpm.

rpm macro changes:

- CODE 11 includes rpm macros to facilitate creating Kernel Module Package build structures that support multiple RPM-based distributions. The spec file template in Appendix A includes these macros:

  - %kernel_module_package_buildreqs - used for “BuildRequires”.
  - %kernel_module_package - used instead of %suse_kernel_module_package (note that the options are slightly different: with %kernel_module_package, “-x” is used to *exclude* flavors from the build, and “-t” replaces “-s” as the option to override the default sub-package template. See section Kernel Module Packages above for a complete description of all %kernel_module_package options.)
  - %kernel_source – used with the “$flavor” argument to specify the location of the top-level kernel Makefile.

- Starting with SLES 11 SP2, the %kernel_module_package macro includes a “-b” option that can be used to force initial ramdisk creation when a KMP is installed.

- Starting with SLES 11 SP3, the %kernel_module_package macro includes a “-c <module-signing-certificate>” option that can be used when building secure-boot-enabled packages.

Changes

December 22, 2008 – andavis@suse.com

- Initial Version: Update the CODE 10 Kernel Module Packages Manual [7], and adapt it to the CODE 11 process.

August 13, 2009 – andavis@suse.com

- Update to correctly reflect behavior of -f option to kernel_module_package macro.

June 27, 2013 – andavis@suse.com

- Update to change Novell->SUSE.
- Update to add -b option to kernel_module_package macro.
- Update to support building of secure-boot-enabled packages.

References

[1] Kurt Garloff et al.: kABI Stability in SLES, http://www.suse.de/~agruen/kabi (Temporary location; please contact Kurt Garloff <garloff@suse.de> in case this URL has become unavailable.)

[2] Greg Kroah-Hartman: The Linux Kernel Driver Interface, http://www.kroah.com/log/linux/stable_api_nonsense.html , also provided as stable_api_nonsense.txt in the upstream kernel source tree.

[3] Andreas Gruenbacher: Working With The SUSE 2.6.x Kernel Sources, provided as README.SUSE in SUSE kernel-source packages.

[4] Update Media HOWTO, ftp://ftp.suse.com/pub/people/hvogel/Update-Media-HOWTO .

[5] Peter Jay Salzman, Michael Burian, Ori Pomerantz: The Linux Kernel Module Programming Guide, http://www.tldp.org/LDP/lkmpg/2.6/html/index.html .

[6] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman: Linux Device Drivers, Third Edition, February 2005, http://www.oreilly.com/catalog/linuxdrive3/ . Also available online at http://lwn.net/Kernel/LDD3/ .

[7] Andreas Gruenbacher: Kernel Module Packages Manual for CODE 10, http://www.novell.com/developer/kernel_module_packages_manuals.html .

[8] Documentation of The Linux Foundation Driver Backport Workgroup, http://www.linuxfoundation.org/en/Driver_Backport .

[9] UEFI Specification, http://www.uefi.org/specs .