SUSE SolidDriver Program

ABSTRACT
Installing third party kernel drivers on SUSE Linux Enterprise products gives customers the opportunity to enable the latest hardware and software technologies. But doing so can be a complicated and uneasy effort. This document covers the goals, standards and techniques provided by the SUSE SolidDriver Program that simplify the task of deploying kernel drivers and provide assurance to customers that do.

Version: 1.3 | June 13, 2018

Contents

Introduction

In today's fast-paced world of technology, organizations are constantly deploying new hardware into their data centers. Using new hardware often requires installing new, vendor-delivered kernel drivers, which causes anxiety for many customers.

Whether it's uncertainty around having the right driver for the right installation, understanding preconditions and knowledge are required for proper installation, ensuring backing by the hardware and OS vendors when support is required, or trust in deploying sensitive kernel code into a business critical environment, there are good reasons for customers to be concerned about installing third party delivered kernel modules.

At SUSE we understand the need to leverage new technology, as well as the requirements to install third party delivered kernel drivers to do so. At the same time we can identify with customer concerns when installing what are essentially components of the OS kernel provided by vendors other than SUSE.

SUSE has a long history of providing an enterprise level operating system software and has over time established many practices that make the installation and maintenance of the OS comfortable and dependable for customers.

The SUSE SolidDriver Program leverages that expertise to create a set of standards that facilitate third parties to provide kernel drivers in a uniform, consistent, proven and compatible manner that gives end users ease and confidence in identifying, installing, and using them in their SUSE Linux Enterprise environments.

This document describes in detail the goals and methods provided by the SUSE SolidDriver Program that benefit SUSE partners and customers alike.

Structure of This Document

This document begins by giving a quick overview or kernel modules (drivers), how they are used, and the way they work. Following that, a brief overview of the kernel modules delivered with SUSE Linux Enterprise kernels is given just before delving into the topic of third party kernel modules. At this point the benefits and risks of installing kernel drivers should be clear and sets us up for a description of the SUSE SolidDriver Program.

The later sections of the document introduces the goals of the SolidDriver Program, outlines some of the standards it establishes, and finally goes into detail about how all of this works.

Kernel Modules

The SUSE SolidDriver Program is focused on the tasks of packaging, delivering, deploying, and supporting kernel modules to be installed on SUSE Linux Enterprise OS's. Before looking at the details of how kernel modules are provided, it's important to have a basic understanding of what Linux kernel modules are and how they work, and this section will give a quick overview.

If you're familiar with Linux kernel modules, you can skip to the section Third Party Kernel Modules.

What are Kernel Modules

Kernel modules pieces of kernel object code that can dynamically loaded into a running instance of a Linux kernel. Not only can the modules be loaded when needed, they can also be unloaded and reloaded[1]. The ability to load sections of kernel code as needed allows for great flexibility when using Linux kernels. Because of the loadable nature they

[1]The reliability of unloading and reloading kernel modules depends on the design of the module code and possibly the hardware itself. Unloading should not be a common task in a production environment and when needed, should be done with care.

are sometimes referred to as Loadable Kernel Modules (LKM).

- The Linux kernel is an extensive program that supports a considerable amount features and functionalities from file-systems, to vast diversities of hardware components. The ability to load only those sections of kernel code that are required for a given installation or workload, allows the kernel to run with a minimal memory footprint.
- When new features, or technologies are to be used, new kernel modules can be loaded as needed without rebooting the system. This is a key part of device hot-plugging support in the Linux kernel where device drivers (kernel modules) are loaded as new devices are discovered.
- Kernel modules can be built separately from the base kernel code and loaded/tested dynamically and easily on existing kernel installations. This reduces the expensive process of building and deploying complete new kernels with each iteration.

Kernel *Modules* vs Kernel *Drivers*

The terms kernel *module* and kernel *driver* are essentially synonymous, and in this document both terms will be used interchangeably.

To be more precise, one can think of *kernel module* is a more general term for code that is dynamically loaded into a running kernel instance, and that *kernel drivers* are kernel modules built with the task of *driving* or otherwise providing interfaces to entities external to the kernel itself whether it's file systems, hardware devices, or software applications.

Power and Privilege of Kernel Modules

As explained in Anatomy of the Linux kernel[2] the GNU/Linux operating system consists of two major parts, the user space and the kernel. The kernel provides resource management functions to the user space, like device access and process and memory management. Everything running in the kernel has full access to the hardware of the entire machine; there is no protection between the different parts. The lack of protection in the kernel makes the code fast, but at the same time dangerous: a bug in one part can bring down the entire system.

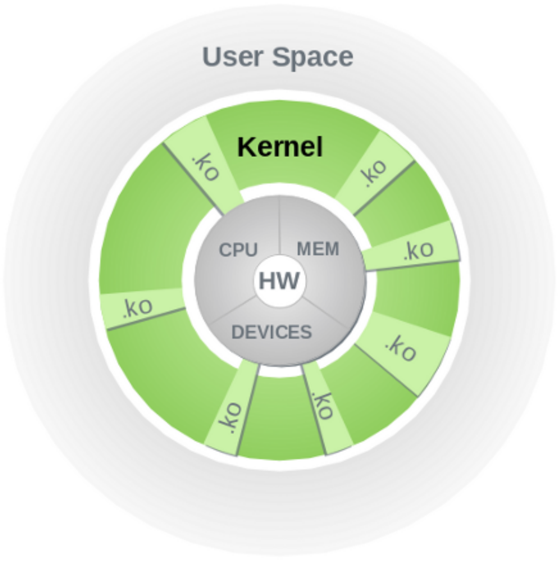

Figure 1. Kernel Modules

The illustration above shows the different layers of a running computer system with the hardware resources shown in the center, the user space on the outer ring with the kernel space in-between. The gray slices labeled *.ko* represent kernel modules. As can be seen in the diagram, there is no separation or layer of protection between kernel modules and the kernel or hardware resources.

The user space builds upon the abstractions provided by the kernel. Everything in the user space is running in the context of a process, and different processes are protected from each other. The main communication layer between the user space and the kernel is the system call interface.

In this document, we focus on the kernel space and the abstractions and mechanisms provided to extend the kerneli's functionality by adding support for new hardware devices or software features.

[2] M. Tim Jones, Anatomy of the Linux kernel: History and architectural decomposition, IBM developerWorks, http://www.ibm.com/developerworks/linux/library/l-linux-kernel/ (2007).

Support for different kinds of hardware lives in drivers, which can be compiled directly into the main kernel image or into kernel modules, which can be loaded at run-time. Likewise, features like file systems or network protocols can be part of the main kernel image or they can be implemented in modules. When a module is loaded, it becomes a part of the kernel and gains all the powers that kernel code has.

The Kernel Module Programming Interface

The programming interface between the kernel and kernel modules is similar to that of shared libraries[3] in user space: a shared library provides ("exports") a set of functions and variables. A user-space program that uses those functions and variables loads the shared library into its address space. The library loading code makes sure that the library is properly initialized ("constructed"), and that all references to functions and variables implemented in the library are properly resolved so that the program can use them. Similarly, the kernel exports functions and variables to kernel modules. When a kernel module is loaded, the kernel resolves all references to functions and variables implemented in the kernel so that the module can access them, and then it initializes the module. (In this analogy, the kernel module is in the role of the user-space program.)

For both shared libraries and kernel modules, the term symbol refers to a function or variable, and the term exported symbol refers to a symbol that a shared library or the kernel provides. Just as a shared library can depend on other shared libraries (meaning it uses symbols exported by those other libraries), kernel modules can depend on other kernel modules.

When discussing programming interfaces, the term Application Programming Interface (API) is often used to refer to an interface at source code level, and the term Application Binary Interface (ABI) is used to refer to the binary interface of an application. These two terms are also used in conjunction with kernel modules; there, the abbreviations kAPI and kABI (k standing for kernel) are also encountered.

[3]David A. Wheeler, Programming Library HOWTO, The Linux Documentation Project, Section 3: Shared Libraries, http://www.tldp.org/HOWTO/Program-Library-HOWTO/ (2003).

Kernel Module Compatibility

When loading kernel modules, it is essential to ensure that the kernel and modules are compatible. We can ensure this by building the kernel and all modules in the same build which insures all data structures involved are equal. When building the kernel modules separate from the core kernel build, another mechanism is needed to ensure that all the data structures involved are the same.

Historically, the kernel and its kernel modules were all built at once. Back then, it was sufficient to ensure that the version strings of the kernel and of all modules were the same; we would just need to ensure that the version string changed whenever an exported symbol became incompatible. When people started building their own kernel modules separately from the rest of the kernel, this weak form of compatibility checking often resulted in breakages. Modules would become incompatible and still load but crash the kernel upon first use of the module, corrupt other parts of the kernel and cause crashes there, or worse, even cause data corruption. Kernel modules built separately from the rest of the kernel could not be handled in a reasonable way with kernel version string comparisons.

To solve this problem, a mechanism called modversions was invented, which assigns a checksum to each exported symbol. The checksums are computed based on how a symbol is defined in the source code; they include the definitions of all data types reachable from each symbol. Instead of making sure that the kernel and module have the same version, the kernel makes sure that the checksums of all the symbols that the module uses match. Modules with checksum mismatches are rejected.

This ensures that a symbol or anything reachable from that symbol changes. However, it might also generate false positives, because a checksum might change even when a module is not actually affected by the change causing the checksum change.

Kernel Modules in SUSE Linux Enterprise

The kernel delivered with SUSE Linux Enterprise products is compiled with many of the components built as modules. This allows for the flexibility as mentioned in the previous section. Today, the SUSE kernels contain around 1800 modules in total, but since these modules are loaded as required based on functional needs of the end user, only a small portion of the total will be loaded at any given time. A typical server installation will have around 100 or even fewer modules loaded.

The ability to load parts of the kernel as needed allows SUSE Linux Enterprise to support a wide range of systems from notebooks to supercomputers. It also allows to support a range of file systems and I/O stacks which provides for suitability with diverse workloads.

The fact that SUSE provides kernels split into modules permits particular modules to be updated when needed to support new technology as is typically done when supporting new hardware releases. These *driver updates* are typically developed and provided by the hardware vendors that produce and sell the hardware products themselves. It's the existence, and use of drivers from these *third party* vendors that is the primary motivation for the SUSE SolidDriver Program which will be covered in the following sections.

Third Party Kernel Modules

As has been pointed out in the previous section, there is the option for *third party vendors* to deliver their own kernel modules that are built for and can be loaded with SUSE Kernels.

The ability to deliver custom or updated kernel drivers allows hardware and software vendors to bring compatibility of their products to SUSE Linux Enterprise OS's, but this capability doesn't come without some challenges. Those challenges are covered in this following sections.

Challenges with Deploying Third Party Kernel Modules

As mentioned in section Power and Privilege of Kernel Modules, kernel modules are first class kernel code that have the same privileges and access to system resources as the rest of the kernel. Kernel modules deployed without care can introduce great risk to the integrity and stability of the complete system. Loading modules of unknown origin or support commitment immediately leads to a running system that is, strictly speaking, unsupported by SUSE.

In addition to the risk of installing kernel modules of unknown origin or support status, kernel modules are tricky to install. Before the SUSE SolidDriver Program, vendors were delivering drivers in many differing and incompatible manners which increased the complexities and likelihood of user error.

Traditional 3rd Party Kernel Module Installation

When open source drivers are involved, a typical method of delivering kernel modules to end users is to simply provide the source code along with build and installation instructions. This requires users to have both a level of software building expertise as well as the proper kernel development environment installed. The common way of building and installing kernel modules as updates to those delivered with the base OS is to *overwrite* the original modules with the updates. This works fine until the next kernel update is installed which will happily overwrite the previously driver update with the original base version. Many times this leads to an un-bootable system.

A second typical delivery method is to provide kernel drivers in a pre-compiled bundle (some form of archive like tar or even rpm) along with an installation script to help make the deployment process *easy* for end users. Some of these scripts are quite extensive and try to manage all the subtleties of proper kernel module installation (and hopefully removal as well if the user ever decides to un-install). Unfortunately vendors tend to work in isolation and have developed varying methods (many of which make unsafe assumptions) to install drivers. The end user is then encountered with multitudes of different, vendor-specific installation processes. Most of the methods work fine with manual installation on single, locally accessed systems, but come short when customers need to perform mass installations, or automated installations of various drivers on mixes of hardware.

Supportability

An additional challenge with deploying third party kernel modules comes with requiring support from SUSE. If the kernel is misbehaving while a third party kernel module is installed, it's important that SUSE engineers can work closely with the vendor of the driver in order to remedy the situation. This is not only true of third party delivered drivers but also true of many of the *in-box* drivers delivered with SUSE Linux Enterprise Kernels where kernel drivers are supported in partnership between SUSE and the hardware vendors.

Just as we have established partnerships and joint support processes setup to help support our in-box kernel drivers, it's important that any third party delivered drivers are accompanied with a proper technical relationship and support process between SUSE and the vendor providing the kernel driver. The SUSE SolidDriver Program provides processes and resources specifically for those tasks. See the Joint Support Agreements section later in this document for more details.

Mitigating Complications

The SUSE SolidDriver Program was established and developed to address the challenges identified with delivering, deploying, and supporting third party kernel modules. Before moving on to describe the program in detail let's take a moment to address a common question...

Why Not Deliver All Required Kernel Modules with the SUSE Kernels?

With all the risks and challenges associated with using third party delivered kernel modules, why not avoid the situation all together by having SUSE deliver all needed drivers with the SUSE Linux Enterprise kernels?

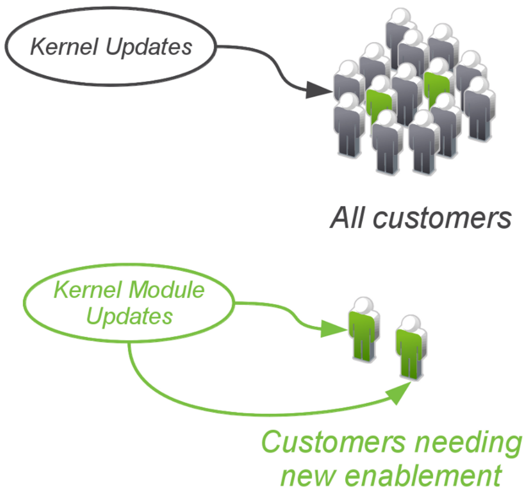

Figure 2. Delivering new kernel functionality

That would indeed take care of many of the challenges. The drivers would simply be installed with our official kernels and kernel updates thereby removing the installation complications, support issues, and issues with trust. It sounds like an ideal solution, but it's not reasonable given the following reasons:

- Additional quality assurance expense
- Additional risk to users that receive no benefit
- Additional delays in delivering the needed support
- Licensing restrictions

Before going into detail on each of these points lets briefly look at the figure below.

The top portion represents standard kernel updates that go to all SUSE customers while the bottom portion represents an individual kernel module update going to those customers requiring the new functionality. In the former case, the exposure of the update is much greater than the latter. The customers with the gray color are customers that would receive no benefit from the updated, new functionality. Only the green customers get the benefit, and further more, once the green customers are up and running with their new feature support they become *gray* customers when the next new features or functional support are introduced with kernel module updates. Keep this image in mind while reading the following sections.

Additional Expense of Quality Assurance

SUSE Linux Enterprise kernels go through a great amount of testing and quality assurance before being delivered to customers. The development of a service pack update entails around 6 months of testing and hardening by SUSE and SUSE's partners. The combined testing efforts span many companies and customers and having the widest and most thorough testing coverage possible in the available time frame. Anyone familiar with the efforts and costs of any kind of product testing can appreciate the amount of expense this calls for. It wouldn't make much sense to invest so much into testing the initial release of a product or a product service pack and not put the same level of attention to subsequent kernel updates. Indeed, SUSE kernel updates are done with care and testing and the amount of code changed for any update is kept to the bare minimum in order to achieve the specific objectives of the update while keeping risk of recessions and required testing to a minimum. These minimal, specific updates require weeks of testing before release.

Of course, kernel driver updates would not require the months of testing that a full service pack update encompasses, but could require weeks of testing by SUSE and SUSE partners in order to ensure no regressions are introduced for the existing install base (the gray customers). The frequency of new technology introductions and required driver updates would quickly create a very expensive process for all involved only to ensure that those customers, who otherwise receive no additional benefit, don't encounter issues when deploying our standard updates. It should be clear that the cost vs risk would be a questionable investment.

Additional Risk

Delivering any kind of update introduces risk of regression. As outlined above, extensive quality assurance is required to reduce the risk of regression to a comfortable level. There is never 100% coverage and some level of risk is always present. Many regressions will unfortunately be initially identified by customers when they deploy the updates. While SUSE commits to fixing such regressions with the highest priority, we would rather avoid them all together. Therefore it's better to restrict kernel module updates to those specific customers that require them and not expose the rest of the customers to unnecessary risk.

Additional Delay

We could avoid the *additional* QA and risk associated with delivering kernel module updates by integrating the updates into the service pack releases. That is exactly what happens with the standard upstream drivers that are included with the SUSE kernels. Typically all commonly used drivers are updated to the latest available version at time of code freeze for a new service pack. The challenge with this approach is that the time between code freeze, and first customer ship spans many months. Given the speed of development in the IT industry, it's not uncommon that at time of first customer ship of a new SUSE Linux Enterprise major release or service pack, many newly released hardware components are not supported by the newly released SUSE product.

It's of value to have, without delay, support in SUSE Linux Enterprise for the latest technologies as they are released to the market. For this reason, use of third party kernel modules is equally valuable and important for our customers.

Licensing Restrictions

Last, but not least, licensing restrictions make it difficult if not impossible for SUSE to deliver some pieces of software including some kernel modules.

Embracing Third Party Kernel Modules

As we have learned so far, the flexibility that loadable kernel modules bring to a Linux based system is a valuable asset while the same time safely applying third party kernel modules to production systems requires care and attention to details. SUSE embraces the delivery and use of third parties with SUSE Linux Enterprise products when done responsibly. The SUSE SolidDriver Program has been designed to address the complications and risks associated with deploying such module updates and provide customers a well integrated, supported, and dependable experience in the process.

Goals of the SUSE SolidDriver Program

The fundamental goal of the SUSE SolidDriver Program can be summed up in the following sentence:

> *Ensure customers can deploy necessary 3rd party kernel drivers with ease and confidence.*

The program achieves this by establishing standards to be followed by vendors who packaging and delivering kernel drivers that are used with SUSE Linux Enterprise OS's. These standards are set out to ensure that kernel drivers are:

- Easy to deploy
- Compatible with SUSE Linux Enterprise products
- Supported by SUSE and the driver vendor
- Trusted by the end user
- Usable with SUSE YES certifications

Easy to Deploy

Given the impact to system functionality that many kernel drivers bring, it's important that they are installed properly. The first step in improving the reliability of kernel module installations is to remove complexity in the operation. We can remove complexity by removing the *need to compile* kernel modules, as well as using *standard* methods and tools to install them.

The SolidDriver program recommends that modules are provided to customers in standard RPM packages that contain pre-compiled kernel module binaries which are compatible with the SUSE Linux Enterprise kernels. The program utilizes a specific *kernel module package* RPM packaging standard that provides for automatic and correct installation relieving the end user of detailed knowledge around building and installing kernel drivers. This packaging standard will be covered in more depth in the section Kernel Module Packages later in this document.

Compatible with SUSE Linux Enterprise

There are two aspects of compatibility that the SolidDriver program is focused on. The first is related to kernel module updates properly co-existing with standard system files. This means that kernel module installations must *preserve* system files and system package installations must *preserve* kernel module files. As was described in the section on third party kernel modules, traditional installation involves installing the module object file in the location reserved for standard kernel modules - sometimes overwriting the default system files in the process. This leads to complications when installing SUSE kernel updates which will may remove or overwrite the files installed by the third party kernel module. In such cases, the third party kernel module will need to be re-installed after the kernel update before reboot in order to ensure the proper driver is available for system functionality. It's up to the system admin to remember this crucial step - failing to do so can result in an un-bootable system that can be tricky to restore. The kernel module package standard defined by the SUSE SolidDriver program places driver updates in a pre-defined, SUSE standard location that is reserved for externally provided kernel modules. This allows for third party kernel modules to be installed in parallel with SUSE kernels without the complications of file collisions.

The second aspect of compatibility is concerned with the kABI (see The Kernel Module Programming Interface). If the kABI required by the kernel module does not match the kABI of the running kernel, the module will not load. Once again the kernel module package standard gives the solution by handling kernel ABI requirements at the package dependency level. This protects users from installing incompatible kernel modules that could lead to systems not working. How this is managed is discussed in detail in the section [Kernel ABI Compatibility] later in this document.

Supportable by SUSE and 3rd Party Vendor

By using kernel module packages, the process of installation of kernel modules is fully supported by SUSE. In addition SUSE SolidDriver compliant drivers ensure that joint support agreements are established between SUSE and the third party vendor so that proper support can be given to customers having any issues related to the SUSE kernels. See Joint Support Agreements for more information.

Trusted by the End User

Kernel modules are proper pieces of kernel code and have the same access and privileges as the rest of the kernel. There is no added protection between the kernel module and the hardware, data, and complete runtime environment to protect against unwanted behavior.

Because of that, users need to have a sufficient level of trust in any kernel drivers that they install on their systems. How does the user establish that trust?

The SUSE SolidDriver Program helps the user establish trust by using a combination of proper package meta data (like package descriptions and vendor identification) along with digital signatures that the user can verify with the vendors he or she trusts. The verified digital signature ensures that the contents of the package indeed originate from the identified vendor and has not been tampered with by external forces.

Usable with SUSE YES Certifications

Hardware vendors will want to utilize their kernel drivers when running SUSE YES Certification tests. Doing so requires that the environment under test is available to end users in the exact configuration as has been certified. This means that any kernel drivers used must be available to the customers in the exact binary form as used for testing and should remain available for the life of the hardware product or SUSE Linux Enterprise OS that the certification is based on.

To help vendors maintain this availability, the SUSE SolidDriver Program offers the opportunity to have their drivers hosted at drivers.suse.com. SUSE will make sure the drivers used in certifications remain available for the required amount of time.

How It Works

Up to this point, the complications related to installing third party kernel modules have been identified, and the goal of the SUSE SolidDriver Program to reduce these complexities and set standards to help do so have been illustrated.

The following sections will go into some detail on technical implementations that meet the standards and help achieve the goal.

- Kernel Module Packages The heart of the SUSE SolidDriver Program standards. This packaging standard allows us to install kernel modules with ease and confidence.
- SUSE Kernel ABI Stability To help keep users running with installed third party kernel modules, SUSE has a policy to keep the kernel ABI stable with kernel updates provided to customers.
- Joint Support Agreements In order to effectively support customers that are running SUSE Linux Enterprise kernels with third party kernel modules, we set into place cooperative support proceses with the third party vendors.
- Driver Kits Driver kits allow a vendor to bundle all software required to enable a product into one location that is easily integrated and installed on SUSE Linux Enterprise operating systems.

Kernel Module Packages

SUSE Linux Enterprise products use the RPM software packaging format for software delivery, installation, and maintenance The package installation stack within SUSE Linux Enterprise is designed to consume and process RPM packages for installation.

Because of this, it's only natural that the SUSE SolidDriver program standardizes on RPM as the recommended way to package kernel drivers. In addition to simply utilizing the RPM format, SUSE has developed an approach specifically designed for packaging kernel modules. This packaging scheme is known as the *Kernel Module Package* or *KMP*.

The KMP defines the location where kernel module object files are installed on the system. This insures compatibility at the file system level between the KMP and other installed packages - specifically the kernel packages.

The SUSE Linux Enterprise products come with a variety of kernels to choose from depending on use case. These kernels are divided by *architecture* and *flavor*. The architecture would be the platform CPU architecture (e.g x86_64, i586, ppc64, ia64 etc.) while the flavor matches a specific kernel configuration like *pae*, *xen*, *trace* or the most common, *default*. Each of these architecture and flavor combinations provides a different, and incompatible kABI which requires separate kernel module object files to be compiled for each target kernel that a driver is developed to work with.

The KMP standard will build, from a single source, kernel modules for each kernel. The build process handles the details behind the scenes which makes the procedure easy for developers. The result is separate *sub-packages* - one for each kernel type.

In addition to the proper location of the installed files, and the intrinsic handling of different kernel flavors, the KMP brings two other capabilities unique to kernel modules:

- *modalias* hints
- kABI dependency hints

The former allows for automated installation of the proper KMPs that matches the underlying hardware while the later ensures that KMPs and kernels are compatible with each other before they are installed.

Automatic Installation via *modaliases*

Understanding exactly which kernel drivers are required for a given hardware product can be a difficult and baffling routine. Kernel drivers have arcane and sometimes outdated nomenclature like: *tg3*, *bnx2x*, *ixgbe*, *qla2xxx* etc. Kernel module packages usually carry the same naming as the kernel modules themselves and since many of them are used generically across different releases and models of hardware, there is no easy way to indicate which exact driver and version is tested and supported with specific hardware products. To complicate matter further, KMPs come in sets of RPM *sub-packages* - one for each

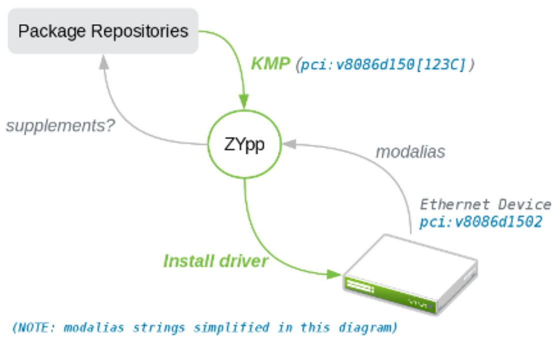

Figure 3. Auto KMP selection

possible kernel configuration. A customer typically only requires one or two of the sub-packages and knowing *which* one might not be easily known.

When kernel drivers are delivered in properly built KMPs, the SUSE installer can pick an choose the right packages to install on a given system. The installer will query the underlying hardware for a list of components, which are identified using unique PCI or USB IDs. The ID strings are stored in what are called *modalias* strings and are the same ones used by the system to determine which kernel modules to load upon hardware detection.

Each KMP in turn provides a list of IDS that it *supplements*. The SUSE installer will query all enabled package repositories for packages supplementing the underlying hardware and when a match is found, the package(s) are selected for installation. The installer also uses information provided by the KMP to install the one built for the installed kernel.

Properly built kernel module packages streamline the kernel driver deployment by allowing customers to simply place the packages in repositories accessed by the SUSE installer. The intelligence built into the installer handles the rest.

Kernel ABI Compatiblity

It's important that structures kernel modules to interface with the kernel are compatible with the installed kernel. This interface is called the kernel ABI or kABI. See the section "Kernel Modules" at the beginning of this document for more information on kABI and kernel module compatibility.

Kernel modules that do not use a compatible

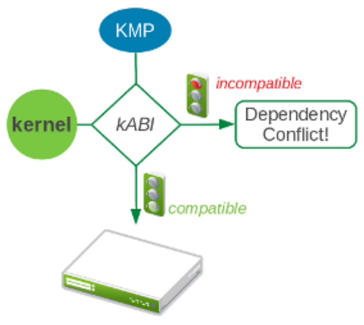

Figure 4. kABI Checks

kABI will fail to load into the running kernel. Installing a kernel module update that fails to load can result in missing functionality crucial for system operation and can even lead to a non-bootable system. Therefore it's important to ensure that only kernel modules that are compatible with the system kernel are installed. Equally important is to only install kernel updates that are compatible with installed, third party kernel modules.

By leveraging the package inter-dependency mechanisms of the KMP standard, the SUSE SolidDriver program helps protect users from unknowingly creating a state of incapability when installing kernel drivers or kernel updates.

The SUSE Linux Enterprise kernel packages are built containing a list of kABI symbol sets that they provide. In turn, KMPs are built containing a list of symbol sets that they require. During installation of either kernel or kernel module packages, the SUSE package installer will verify that the symbol sets match. If not, a package dependency error is thrown, and the package is not installed. In this way, we ensure that incompatible kernels or kernel modules are not installed and save the user from potential complications with system operation.

Putting the Pieces Together

The *KMP Pictorial View* illustration above shows how all the pieces of the kernel module package fit together. On the left hand side are the components of the package including the actual package files (in rounded boxes) and package meta data (dog-eared rectangles). In the middle is the KMP itself where all the components are packaged. To the

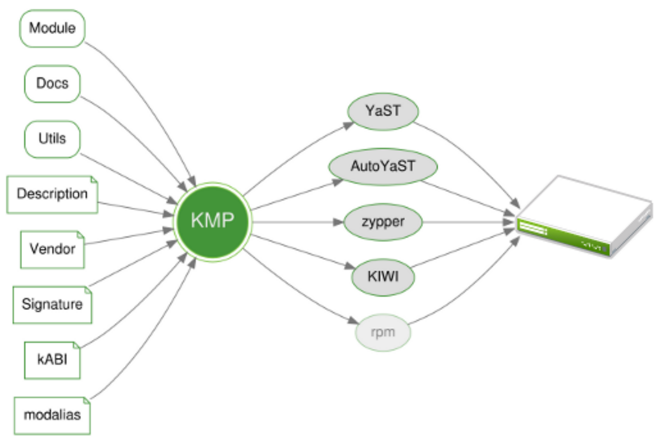

Figure 5. KMP Pictorial View

right of the KMP extend arrows to ellipses designating the different modes of installing the KMP onto the target system at the far right of the figure.

Below is a brief description of each of the KMP components:

Module This is the actual kernel module object file or files.

Docs Package documentation.

Utils Optional, supplemental utilities programs and scripts to be used in conjunction with the kernel module.

Description Package text that briefly describes the contents of the package. This includes both the RPM *Summary* and RPM *Description*.

Vendor String indicating the vendor producing and supporting the package.

Signature The public part of a digital signature of the package that ensures that the package has not been altered by anyone other than that owning the private key of the signature.

kABI The kABI symbol set dependencies of the kernel module.

modalias The modalias list linking this package to the specific hardware that the kernel module supports.

These pieces are packaged into the KMP which is installable onto the target system using **YaST**, **AutoYaST**, or **zypper** (SUSE's native package installation tools). In addition KMPs can be integrated into appliances using SUSE's KIWI tool set. Lastly, since

KMPs are standard RPM packages, they can be installed using the plain `rpm` command. The `rpm` bubble is shown in lightly subdued because it's considered by SUSE to be a low level tool that does not provide the richness of features that the SUSE installers provide.

SUSE Kernel ABI Stability

It would be great if OS kernels had support for all features now and in the future, had no bugs, and no known security holes. Then we could deploy our systems and not be concerned with updates and potential regressions The fact is, kernels do have bugs and security holes that must be addressed. In addition, new features and functionality are constantly being developed, enhanced and optimized. This is true for the SUSE Linux Enterprise kernels, and customers need to deal with the updates that come their way.

These updates, which contain kernel code changes, can lead to incompatibilities with other parts of the system if precautions are not taken. In this document we are concerned with the compatibility at the kernel ABI (kABI) level. This compatibility is sometimes termed *kABI Stability* and only applies to kernel level code (i.e. kernel modules). For more information on the kernel programing interfaces see the The Kernel Module Programming Interface section earlier in this document.

When updating kernels, SUSE does take precautions with regard to the stability of the kernel ABI and has the following policy for SUSE Linux Enterprise products:

- Keep the kernel ABI stable for the lifetime of each service pack. Kernel module packages built for a specific service pack will remain compatible with all update kernels to that service pack. The same policy holds for kernels updates between the initial release of the product and the first service pack.

- With service pack or major version updates, there is no guarantee of kernel ABI compatibility with the previous releases[4]. Kernel module packages maintained for several service pack releases will need to be rebuilt for each service pack.

Given the life of general support of an individual service pack being around two years and the option to extend that by an addition three years using *Long Term Service Pack Support*[5], the policy provides for a fairly long stable period with compatibility changes happening at predefined and well planned intervals. That allows for the introduction of substantial improvements over the life of the product, resulting in an overall better offering to our customers without sacrificing longer term stability and compatibility.

What does it mean for vendors?

If a vendor provides modules that are maintained in the upstream kernel and supported in the SUSE Linux Enterprise product, most likely kernel ABI compatibility will not be an issue. It's important to work with SUSE to ensure the module is considered supported in the SUSE products, and gets updated to the latest upstream version at each service pack release.

If the modules are not part of the upstream kernel, they will need to be rebuilt for each service pack and possibly require source code changes to match the updated kernel interfaces. SUSE offers a beta program allowing for partners to test and prepare for any kernel module adjustments well in advance of the service pack release.

For further information on the SUSE beta program or ensuring that your upstream drivers are up to date and supported in SUSE Linux Enterprise products, contact your PartnerNet representative[6].

What does it mean for users?

The most prominent use of third party kernel modules is updating the standard drivers delivered with SUSE Linux Enterprise products to enable new products or features. Such updates are typically maintained upstream,

[4]Later service packs (e.g. SUSE Linux Enterprise 10 SP4) tend to have smaller changes to the kernel, and in these cases we strive to keep the kernel ABI compatible with the previous service pack release where possible.

[5]https://www.suse.com/support/programs/long-term-service-pack-support.html

[6]https://www.suse.com/partners/

and as a result are included in the next service pack release removing the need for the updates going forward. In those cases, the kernel ABI changes coming with service pack updates will not affect end users.

If the modules are not maintained in the upstream kernel, then the user will need to take care when deploying service pack updates to ensure that new, compatible versions of the kernel modules are also applied. Users are advised to contact the vendor of the kernel modules for assistance with updating to new service packs.

Why not maintain compatibility across service packs?

SUSE Linux Enterprise is based on the community developed Linux kernel, and by that, leverages the advancements and code quality gained from the community development process[7]. To help customers best take advantage of the ongoing community work, we strive to keep the SUSE Linux Enterprise kernels as close as possible to the upstream community code base. Since the ABI of the community kernel is continually changing, maintaining compatibility over time would require diverging more and more from the community. As this divergence widens, the benefits that we and our customers realize by being close to and part of the general kernel community diminish.

So, while keeping the kernel ABI compatible for the life of a SUSE product is preferred by vendors that deliver their own drivers, the resulting divergence from upstream is not the best option for maintaining code quality over the long term. On the other hand, breaking the kernel ABI compatibility several times a year is neither desirable nor imperative.

SUSE's policy for kernel ABI stability strikes a balance between compatibility and advancement that allows us and our customers to take advantage of new upstream development during the life of SUSE Linux Enterprise while minimizing divergence from general kernel community.

[7]Ibrahim Haddad (Ph.D.) and Brian Warner, Upstreaming: Strengthening Open Source Development, The Linux Foundation, Section 3: Benefits of Upstreaming, http://www.linuxfoundation.org/publications/linux-foundation/upstreaming-strengthening-open-source-development

Does SUSE provide a kernel ABI symbol whitelist?

One tactic to maintaining a stable kernel ABI over a long period of time is to define a subset of the complete kernel ABI guaranteed to remain compatible. This subset is often termed a "whitelist" referring to only the symbols that are "safe" to use in external kernel modules. SUSE does not follow this route and instead keep virtually all[8] symbols unchanged, but for a shorter period of time. We have found this policy to be better suited to most vendors maintaining kernel modules as it provides freedom and flexibility in leveraging the kernel interfaces they require to support their products.

Compatibility With User Space Applications

We have focused on the interfaces at the kernel level only (i.e. between the kernel and kernel modules); the topic of user space interfaces is beyond the scope of this document. It will simply be stated that the formal programming interfaces between kernel and user space are kept compatible for the life of a major release of SUSE Linux Enterprise Server and Desktop products. Indeed, most of the interfaces are compatible across major releases as well.

Joint Support Agreements

A running kernel that has had third party kernel module loaded can not be supported by SUSE engineering without cooperation of the vendor of the kernel module code. In order to ensure supportability to SUSE Linux Enterprise customers, SUSE requires a support process agreement to be established between vendor's of third party kernel modules and SUSE.

Process Agreement

The joint support agreement establishes a process for handling customer support of SUSE Linux Enterprise kernels running kernel modules supplied by partners. With the agreement, SUSE technical support engineers, can handle initial triage and data

[8]We do allow rare exceptions for a few symbols that clearly make no sense for an external kernel module to use.

collection from the customer. Once it is determined that the partner's kernel module is suspect to the customer's problem, or if there is no way to rule out the partner's kernel module as root cause, SUSE will involve the partner's support contacts as outlined in the support process agreement.

Through the joint support agreement, SUSE and the partner can work together on resolving the issue while maintaining a consistent interface to the end customer.

Without a Joint Support Agreement

Without a joint support agreement in place, SUSE can't do root-cause analysis of kernel issues when third party kernel modules have been loaded. Customers will be instructed to contact the vendor of the third party kernel module for additional support and analysis, in which case the vendor will need to either fix the issue in the kernel module code, or root cause and provide evidence of a valid bug in the SUSE Linux Enterprise kernel for SUSE for remedy.

Kernel Support Taint Flag

Modules can be built with a special flag that indicates the support status of the module. The module can be queried for the mode of support flag using the `modinfo` command. There are two possible modes:

- **Supported: yes** = SUSE supported
- **Supported: external** = supported by driver vendor and SUSE.

If neither of these modes are indicated, the supportability of the kernel module, and complete kernel, is unknown and should be assumed as *unsupportable.*

When loading modules The SUSE Linux Enterprise kernels will identify the supportability of the module code and set the appropriate kernel *taint* flag if an unsupported or externally supported module is loaded. The kernel taint flags are one mechanism to help customers and SUSE support technicians identify the supportability of a running kernel environment.

In default and recommended configuration, SUSE Linux Enterprise Server will refuse to load kernel modules that cannot be identified as supported by SUSE or an external party. Running unknown or unsupported code in kernel space invalidates any support commitments between SUSE and the end user.

Kernel Taint States and Support Levels

When the kernel issues an error message it will indicate the taint state of the kernel at that time. The information is encoded through single-character flags in the string following "Tainted:" in a kernel error message.

The descriptions below only apply to environments running official SUSE Linux Enterprise kernels.

Untainted kernel = SUSE Supported
When a kernel is not tainted with an 'X' or 'N' flag, all loaded kernel code is fully supported by SUSE and SUSE engineering has expertise to debug and fix the code. Only code that has been built, tested, and shipped by SUSE loads without tainting the kernel.

'X' tainted kernel = e'X'ternally Supported
Some code shipped with SUSE Linux Enterprise kernels, and all code built and shipped by SUSE partners, requires the partner's assistance in supporting the code. If any such code is loaded by the kernel, the 'X' taint flag will be set indicating that partner assistance is necessary to provide support. Only official partners who have support agreements established with SUSE are allowed to identify their kernel module binaries as externally supported.

'N' tainted kernel = u'N'supported If at any time, a running kernel instance loads a kernel module that cannot be identified as supported by SUSE or supported by a SUSE partner, the 'N' taint flag will be set. Such an environment is not considered supported by SUSE and looses guarantees of any efforts by SUSE towards resolution to customer issues encountered. To restore the SUSE support guarantees, the issue must be reproduced in an environment without the 'N' kernel taint flag set.

The kernel maintains list of taint flags in addition to the ones related to support. For the full list and descriptions, refer to `/usr/src/linux/Documentation/sysctl/kernel.txt`

Assistance from SUSE Partners

As stated above, in order to fully resolve customer issues, code built and shipped by SUSE partners as well as some code shipped with SUSE Linux Enterprise products might require assistance by SUSE partners. SUSE provides support commitments to such code based on strong partnerships and established agreements between between partners and SUSE for the specific code in question. For the SUSE Linux Enterprise customer, the support process will be transparent and all partner assistance handled behind the scenes between SUSE and the partners. In most cases, the customer will not notice the difference between support coming solely from SUSE or with the cooperation of SUSE partners.

Driver Kits

When enabling products such as server or desktop systems, vendors commonly provide to customers a handful of "packages" to be installed on the target OS. These packages can be anything from firmware updates, device drivers, and documentation that directly enable the hardware, to utilities and programs that enhance the user experience.

When providing packages to enable hardware products for use with SUSE Linux Enterprise products, the SUSE SolidDriver Program recommends delivering all packages for a given product, or even product *family* on a single medium so that customers have the opportunity to obtain from a single source everything that they require.

Driver Kit Add-ons

The medium of choice when providing bundled software is the *add-On Product* as described in the SUSE Linux Enterprise documentation:

"Add-on products are system extensions. You can install a third party add-on product or a special system extension of SUSE Linux Enterprise Server (for example, a CD with support for additional languages or a CD with binary drivers). To install a new add-on, start YaST and select Software+Add-On Products. You can select various types of product media, like CD, FTP, USB mass storage devices (such as USB flash drives or disks) or a local directory. You can work also directly with ISO files. To add an add-on as ISO file media, select Local ISO Image then enter the Path to ISO Image. The Repository Name is arbitrary."[9]

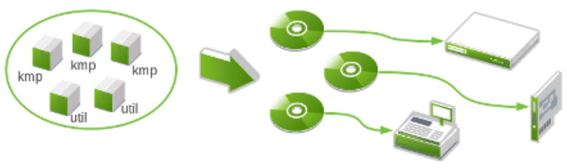

Figure 6. Driver Kits Enable Products

A *Driver Kit* is an add-on that contains kernel drivers to be installed on a given release of SUSE Linux Enterprise product. The driver kit add-on can be installed during the initial installation of the SUSE Enterprise product by checking Include Add-On Products from Separate Media on the Installation Mode screen or installed afterwards using the YaST2 Add-on Products module. For further information refer to the SUSE Linux Enterprise Server Deployment Guide

Driver Kits Target Products

Proper preparation and documentation of driver kits is important for customer usability. The customer should be expected to know two things:

- Hardware *product* to be enabled
- SUSE Linux Enterprise *product* that is to be used with the hardware

Therefore, a driver kit should be documented and delivered based on the products it enables, rather than on the drivers it contains. It's also important that a driver kit indicates the SUSE Linux Enterprise product that it's designed to be installed on.

A driver kit that is documented as

[9]"Installing Add-On Products" in SUSE Linux Enterprise Server Deployment Guide.

> Enables ACME ProServer S3000 server with SUSE Linux Enterprise Server 11

is much more useful to the end user than a driver kit described as

> Provides updates to the igb and mpt2sas drivers

The names of drivers should not be of primary concern to the end user and the user should not need to figure out if these drivers apply to his product. In addition to proper description of the products that a driver kit enables, a list of drivers or other packages contained in the driver kit is useful as *supplementary* information.

It should assumed that the driver kit has been tested by the vendor for the products it's indicated to work with.

These details go a long way toward usability and confidence for the end user and is another aspect of how the SUSE SolidDriver Program helps ensure the best user experience when using third party kernel modules with SUSE Linux Enterprise products.

Bootable Driver Kits

There are times when kernel drivers are required to boot a system in order to install the OS. There are even times when an updated kernel is required to boot a system. Since the bits contained in the SUSE Linux Enterprise product are frozen when released to customers, and do not get updated until the next service pack, the *Bootable Driver Kit* was developed by the SUSE SolidDriver Program to easily enable bringing up systems using updated kernels or kernel modules so that installation of the OS can be achieved.

A Bootable Driver Kit is an add-on product image that contains a boot loader section in addition to the standard driver kit add-on repository. The boot loader contains isolinux and UEFI loaders as well as a kernel and initrd containing the linuxrc[10] program which in turn initiates the SUSE Linux Enterprise installation process. Only the components of this first stage of the installation boot process are contained on the bootable driver kit, which is used to kick-off the installation that then continues using the standard SUSE Linux Enterprise product media or repositories.

[10] http://en.opensuse.org/SDB:Linuxrc

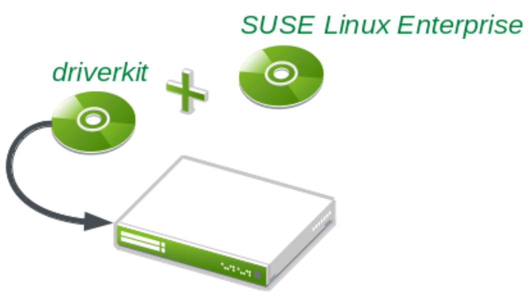

Figure 7. Installing with Optical Media

One of the other benefits that the bootable driver kit brings over the standard driver kit is that it *primes* the installation process by automatically adding the driver kit add-on to the selection of repositories used for installation. It can be designed to automatically install the additional packages that it provides or even provide user choice via *patterns*. This results in eliminating any extra steps that the end user for to do during the installation process to ensure that the packages provided by the driver kit get installed together with the OS. The installation of the driver it is *seamless*.

Installation with Bootable Driver Kits

To illustrate the flexibility and ease of using bootable driver kits, we will briefly describe the workflow of four installation methods:

- Installing with standard DVD/CDs
- Installing over network
- Installing using PXE boot
- Installing directly from repository on the Internet

Traditional Optical Media Installation

First we will show how the bootable driver kit is used to install SUSE Linux Enterprise using optical media (DVDs).

Workflow

1. Burn the driver kit and SUSE products to DVD media

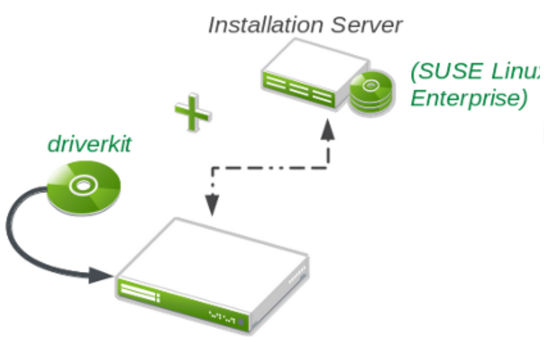

Figure 8. Network based installation

2. Boot machine using driver kit media
3. First stage installer instructs user to install the SUSE Linux Enterprise product media
4. Installation continues as normal

Except for the fact that the driver kit media is used to boot the machine, and the SUSE product media is to be inserted as a secondary step, there is virtually no difference in to the installation experience compared to the default SUSE Linux Enterprise process. The initial boot loader menu can be used to add installer command line options or even instruct the system to install from network which we will illustrate next.

Network Based Installation

Installing a system from a network installation server is covered next. In this method the driver kit media is still used to boot the system before initiating the network based install.

Workflow

1. Burn driver kit image to DVD media
2. Boot machine using driver kit media
3. Using the `install=` command line option, instruct the installer to pull the SUSE Linux Enterprise product from a server over the network
4. Installation continues as normal

As can be seen, installing from a network server is just as easy as installing completely from optical media. In this example optical media was still utilized to boot the system. To avoid that, and initial complete remote based installations, a PXE boot environment is commonly utilized.

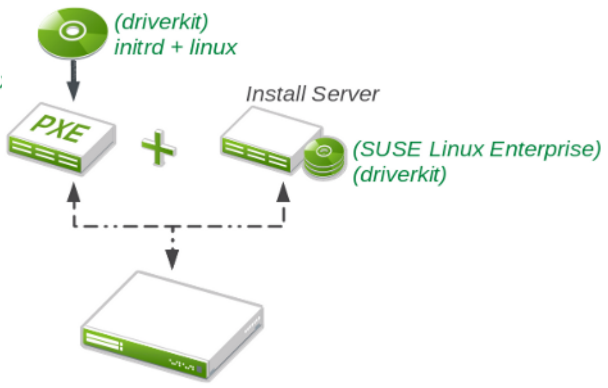

Figure 9. PXE Boot Network installation

PXE Boot Installation

Setting up a PXE boot environment for doing full network based installs requires pulling the kernel and initrd images from the base media and hosting them on the PXE server. Leveraging a bootable driver kit in the process is basically the same.

NOTE: the instructions below are only for reference and leave out many details For comprehensive instructions on setting up a PXE environment refer to the deployment guide of the SUSE Linux Enterprise product.

Workflow

1. Host SUSE Linux Enterprise product as well as the driver kit on an installation server.
2. Setup the PXE boot server to use the kernel and initrd images from the bootable driver kit.
3. Add to the pxelinux config an `install=` boot option that points to the URL of the SUSE Linux Enterprise product repository
4. Add an `addon=` option the pxelinux config file that points to the URL of the driver kit repository
5. Boot system using the systems network PXE boot source
6. Continue installation as usual

Of the steps listed above, only the fourth step is unique compared to standard PXE installation of SUSE Linux Enterprise products. Since the driver kit is an additional software repository, we need a way to instruct the installer to use it during installation. That is

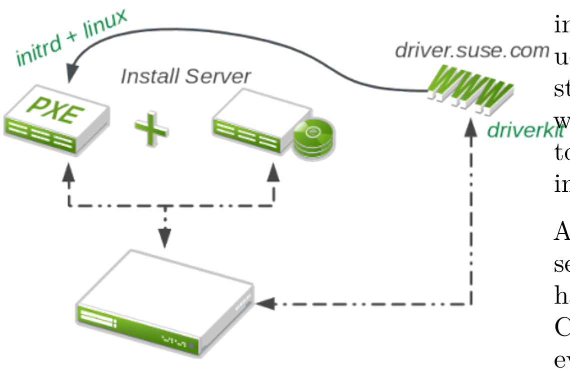

Figure 10. Installing from drivers.suse.com

accomplished using the addon= command line option.

Installing directly from drivers.suse.com

In the PXE Boot example above, we learned how the addon= command line option allows us to point the installer at a network installation source to pull in an extra, add-on repository. If the system to be installed has Internet connectivity, the driver kit can be installed directly from a vendor hosted repository. In this example we illustrate pulling the driver kit directly from drivers.suse.com during installation. The workflow is identical to the PXE Boot workflow described above - only the add-on URL is changed.

Summary

Kernel drivers are required to enable hardware or software products and features and new products often requires new or updated drivers to function properly. Hardware and software vendors alike are able to deliver the needed kernel drivers for enabling their products with SUSE Linux Enterprise OSs and this is usually the quickest way to get new support to the customers. In the end, many SUSE customers will be confronted with the task of deploying and utilizing third party delivered kernel modules in their IT environments. Care must be taken as any form of kernel code, including kernel modules, can impact the integrity and stability of the complete system.

The SUSE SolidDriver Program has established standards and best practices for packaging and delivering kernel drivers to be installed on SUSE Linux Enterprise products. These standards ensure ease of installation, compatibility, supportability, as well as trust and integrity that provide customers with confidence and assurance when installing third party kernel drivers.

An overview of the standard and techniques set forth by the SUSE SolidDriver Program has been presented in this document. Customers can use this information to help evaluate the completeness and suitability of drivers that are given to them. SUSE partners can use the general advice that has been provided to consider how they can better deliver their drivers in a standard, compatible and user satisfactory manner.