Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

System Firmware Updates Utilizing Sofware Repositories

OR: Two proprietary vendor firmware update packages walk into a dark alley, six RPMS in a yum repository walk out...

Matt Domsch
Dell
Matt_Domsch@dell.com

Michael Brown
Dell
Michael_E_Brown@dell.com

Abstract

Traditionally, hardware vendors don't make it easy to update the firmware (motherboard BIOSes, RAID controller firmware, systems management firmware, etc.) that's flashed into their systems. Most provide DOS-based tools to accomplish this, requiring a reboot into a DOS environment. In addition, some vendors release OS-specific, proprietary tools, in proprietary formats, to accomplish this. Examples include Dell Update Packages for BIOS and firmware, HP Online System Firmware Update Component for Linux, and IBM ServRAID BIOS and Firmware updates for Linux. These tools only work on select operating systems, are large because they carry all necessary prerequisite components in each package, and cannot easily be integrated into existing Linux change management frameworks such as YUM repositories, Debian repositories, Red Hat Network service, or Novell/SuSE YaST Online Update repositories.

We propose a new architecture that utilizes native Linux packaging formats (.rpm, .deb) and native Linux change management frameworks (yum, apt, etc.) for delivering and installing system firmware. This architecture is OS distribution, hardware vendor, device, and change management system agnostic.

The architecture is easy as PIE: splitting Payload, Inventory, and Executable components into separate packages, using package format Requires/Provides language to handle dependencies at a package installation level, and using matching Requires/Provides language to handle runtime dependency resolution and installation ordering.

The framework then provides unifying applications such as `inventory_firmware` and `apply_updates` that handle runtime ordering of inventory, execution, and conflict resolution/notification for all of the plug-ins. These are the commands a system administrator runs. Once all of the separate payload, inventory, and execution packages are in package manager format, and are put into package manager repositories, then standard tools can retreive, install, and execute them:

```
# yum install $(inventory_firmware -b)
# apply_updates
```

We present a proof-of-concept source code implementing the base of this system; web site

and repository containing Dell desktop, notebook, workstation, and server BIOS images; open source tools for flashing Dell BIOSes; and open source tools to build such a repository yourself.

1 Overview

The purpose of this paper is to describe a proposal and sample implementation to perform generic, vendor-neutral, firmware updates using a system that integrates cleanly into a normal Linux environment. Firmware includes things such as system BIOS; addon-card firmware, e.g. RAID cards; system Baseboard Management (BMC), hard drives, etc. The first concept of the proposal is the definition of a basic update framework and a plugin API to inventory and update the system. For this, we define some basic utilities upon which to base the update system. We also define a plug-in architecture and API so that different vendor tools can cleanly integrate into the system. The second critical piece of the update system is cleanly separating system inventory, execution of updates, and payload, i.e. individual firmware images. After defining the basic utilities to glue these functions together, we define how a package management system should package each function. Last, we define the interaction between the package manager and the repository manager to create a defined interface for searching a repository for applicable updates. This paper will cover each of these points. The proposal describes an implementation, called `firmware-tools` [1].

2 Infrastructure

This section will detail the basic components used by the firmware update system, and firmware-tools. The base infrastructure for this system consists of two components: inventory and execution. These are named `inventory_firmware` and `apply_updates`, respectively. These are currently command-line utilities, but it is anticipated that, after the major architectural issues are worked out and this has been more widely peer-reviewed, there will be GUI wrappers written.

The basic assumption is that, before you can update firmware on a device, you need several pieces of information.

- What is the existing firmware version?
- What are the available versions of firmware that are on-disk?
- How do you do a version comparison?
- How do I get the correct packages installed for the hardware I have? In other words, solve the bootstrap issue.

It is important to note that all of these questions are independent of exactly how the files and utilities get installed on the system. We have deliberately split out the behavior of the installed utilities from the specification of how these utilities are installed. This allows us flexibility in packaging the tools using the "best" method for the system. Packaging will be discussed in a section below. The specification of packaging and how it interacts with the repository layer is an important aspect of how the initial set of utilities get bootstrapped onto the system, as well as how payload upgrades are handled over time.

2.1 Existing Firmware Version

The answer to the question, "What is the existing firmware version?" is provided by the

inventory_firmware tool. The basic inventory_firmware tool has no capability to inventory anything; all inventory capability is provided by plugins. Plugins consist of a python module with a specific entry point, plus a configuration fragment to tell inventory_firmware about the plugin. Each plugin provides inventory capability for one device type. The plugin API is covered in the API section of this paper, below. It should be noted that, at this point, the plugin API is still open for suggestions and updates.

As an example, there is a dell-lsiflash package that provides a plugin to inventory firmware on LSI RAID adapters. The dell-lsiflash plugin package drops a configuration file fragment into the plugin directory /etc/firmware/firmware.d/ in order to activate the plugin. This configuration file fragment looks like this:

```
[delllsi]
# plugin that provides
# inventory for LSI RAID cards.
inventory_plugin=delllsi
```

This causes the inventory_plugin to load a python module named delllsi.py and use the entry points defined there to perform inventory on LSI RAID cards. The delllsi.py module is free to do the inventory any way it chooses. For example, there are vendor utilities that can sometimes be re-purposed to provide quick and easy inventory. In this specific case, we have written a small python extension module in C which calls a specific ioctl() in the LSI megaraid drivers to perform the inventory and works across all LSI hardware supported by the megaraid driver family. Note that while the framework is open source, the per-device inventory applications may choose their own licenses (of course, open source apps are strongly preferred).

2.2 Available Firmware Images

The next critical part of infrastructure lies in enumerating the payload files that are available on-disk. The main firmware-tools configuration file defines the top-level directory where firmware payloads are stored. The default location for firmware images is /usr/share/firmware/. This can be changed such that, for example, multiple systems network mount a central repository of firmware images. In general each type or class of firmware update will create a subdirectory under the main top-level directory, and each individual firmware payload will have another subdirectory under that.

Each individual firmware payload consists of two files: a binary data file of the firmware and a package.ini metatdata file used by the firmware-tools utilities. It specifies the modules to be used to apply the update and the version of the update, among other things.

2.3 Version Comparison

Another interesting problem lies in doing version comparison between different version strings to try to figure out which is newer, due to the multitude of version string formats used by different firmware types. For example, some firmware might have version strings such as A01, A02, etc., while other firmware has version strings such as 2.7.0-1234, 2.8.1-1532, etc. Each different system may have different precedence rules. For example, current Dell BIOS releases have version strings in sequence like A01, A02, etc. But non-release, beta BIOS have version strings like X01, X02, etc., and developer test BIOS have version strings like P01, P02, etc. This poses a problem because a naive string comparison would always rank beta "X-rev" BIOS as higher version than production BIOS, which is undesirable.

The solution to this problem is to allow plugins to define version comparison functions. These functions take two strings as input and output which one is newer. Each `package.ini` configuration file contains the payload version, plus the name of the plugin to use for version comparison.

2.4 Initial Package Installation—Bootstrap

The last interesting problem arises when you consider how to decide which packages to download from the package repository and install on the local machine. This is a critical problem to solve in order to drive usability of this solution. If the user has to know details of the machine to manually decide which packages to download, then the system will not be sucessful. Next to consider is that a centralized solution does not fit in well with the distributed nature of Linux, Linux development, and the many vendors we hope to support with this solution. We aim to provide a distributed solution where the packages themselves carry the necessary metadata such that a repository manager metadata query can provide an accurate list of which package is needed.

Normal package metadata relates to the software in the package, including files, libraries, virtual package names, etc. The firmware-tools concept extends this by defining "applicability metadata" and adding it to the payload packages. For example, we add
`Provides: pci_firmware(...)`
RPM tags to tell that the given RPM file is applicable to certain PCI cards. Details on packaging are in the next section, including specifications on package Provides that must be in each package.

We then provide a special "bootstrap inventory" mode for the inventory tool. In this mode, `inventory_firmware` outputs a standardized set of package Provides names, based upon the current system hardware configuration. By default, this list only includes `pci_firmware(...)`. Additional vendor-specific addon packs can add other, vendor-specific package names. For example, the Dell addon pack, firmware-addon-dell, adds `system_bios(...)` and `bmc_firmware(...)` standard packages to the list. We hope for wide vendor adoption in this area, where different vendors can provide addon packs for their standard systems. In this manner, the user need not know anything about their hardware, other than the manufacturer. They simply ask their repository manager to install the addon pack for their system. They then run bootstrap inventory to get a list of all other required packages. This list is fed to the OS repository manager, for example, yum, up2date, apt, etc. The repository manager will then search the repository for packages with matching Provides names. This package will normally be the firmware payload package. Through the use of Requires, the payload packages will then pull the execution and inventory packages into the transaction.

3 plugin-api

The current `firmware-tools` provides only infrastructure. All actual work is done by writing plugins to do either inventory, bootstrap, or execution tasks. We expect that as new members join the firmware-tools project this API will evolve. The current API is very straightforward, consisting of a configuration file, two mandatory function calls, and one optional function call. It is implemented in python, but we anticipate that in the future we may add a C API, or something like a WBEM API. The strength of the current implementation is its simplicity.

3.1 Configuration

Plugins are expected to write a configuration file fragment into `/etc/firmware/firmware.d/`. This fragment should be named `modulename.conf`. It is an `INI`-format configuration file that is read with the python ConfigParser module. Each configuration fragment should have one section named the same as the plugin, for example, `[delllsi]`. At the moment, there are only two configuration directives that can be placed in this section. The first is, `bootstrap_inventory_plugin=` and the other is `inventory_plugin=`.

3.2 Bootstrap Inventory

When in bootstrap mode, `inventory_firmware` searches the configuration for `bootstrap_inventory_plugin=` directives. It then dynamically loads the specified python module. It then calls the `BootstrapGenerator()` function in that module. This function takes no arguments and is expected to be a python "generator" function [2]. This function yields, one-by-one, instances of the `package.InstalledPackage` class.

Figure 1 illustrates the Dell bootstrap generator for the `firmware-addon-dell` package.

This module is responsible for generating a list of all possible packages that could be applicable to Dell systems. As you can see, it outputs two standard packages, `system_bios(...)` and `bmc_firmware(...)`. It is also responsible for outputting a list of `pci_firmware(...)` packages with the system name appended. In the future, as more packages are added to the system, we anticipate that the bootstrap will also output package names for things such as external SCSI/SAS enclosures, system backplanes, etc.

3.3 System Inventory

When in system inventory mode, `inventory_firmware` searches the configuration for `inventory_plugin=` directives. It then dynamically loads the specified python module. It then calls the `InventoryGenerator()` function in that module. This function takes no arguments and is expected to be a python "generator" function. This function yields, one-by-one, instances of the `package.InstalledPackage` class. The difference here between this and bootstrap mode is that, in system inventory mode, the inventory function will populate `version` and `compareStrategy` fields of the `package.InstalledPackage` class.

Figure 2 illustrates the Dell inventory generator for the firmware-addon-dell package.

The inventory generator in this instance outputs only the BIOS inventory, with more detailed version information. It is also responsible for setting up the correct comparison function to use for version comparison purposes.

3.4 On-Disk Payload Repository

The on-disk payload repository is the toplevel directory where firmware payloads are stored. There is currently not a separate tools to generate an inventory of the repository, but, there is python module code in `repository.py` which will provide a list of available packages in the on-disk repository. The `repository.Repository` class handles the on-disk repository. The constructor should be given the top-level directory. After construction, the `iterPackages()` or `iterLatestPackages()` generator function methods can be called to get a list of packages in the repository. These generator functions output either all repository packages,

```
# standard entry point -- Bootstrap
def BootstrapGenerator():
  # standard function call to get Dell System ID
  sysId = biosHdr.getSystemId()

  # output packages for Dell BIOS and BMC
  for i in [ "system_bios(ven_0x1028_dev_0x%04x)", "bmc_firmware(ven_0x1028_dev_0x%04x)" ]:
    p = package.InstalledPackage(
        name = (i % sysId).lower()
        )
    yield p

  # output all normal PCI bootstrap packages with system-specific name appended.
  module = __import__("bootstrap_pci", globals(),  locals(), [])
  for pkg in module.BootstrapGenerator():
    pkg.name = "%s/%s" % (pkg.name, "system(ven_0x1028_dev_0x%04x)" % sysId)
    yield pkg
```

Figure 1: Dell bootstrap generator code

```
# standard entry point -- Inventory
def InventoryGenerator():
  sysId = biosHdr.getSystemId()
  biosVer = biosHdr.getSystemBiosVer()
  p = package.InstalledPackage(
    name = ("system_bios(ven_0x1028_dev_0x%04x)" % sysId).lower(),
    version = biosVer,
    compareStrategy = biosHdr.compareVersions,
    )
  yield p
```

Figure 2: Dell inventory generator code

or only latest packages, respectively. They read the `package.ini` file for each package and output an instance of `package.RepostitoryPackage`. The `package.ini` specifies the wrapper to use for each repository package object. The wrapper will override the `compareVersion()` and `install()` methods as appropriate.

3.5 Execution

Execution is handled by calling the `install()` on a package object returned from the repository inventory. The `install()` method is set up by a type-specific wrapper, as specified in the `package.ini` file. Figure 3 shows a typical wrapper class.

The wrapper constructor is passed a package object. The wrapper will then set up methods in the package object for install and version compare. Typical installation function is a simple call to a vendor command line tool. In this example, it uses the open-source `dell_rbu` kernel driver and the open-source libsmbios [3] `dellBiosUpdate` application to perform the update.

4 Packaging

The goal of packaging is to make it as easy as possible to integrate firmware update applications and payloads into existing OS deployments. This means following a standards-based packaging format. For Linux, this is the Linux Standard Base-specified Red Hat Package Manager (RPM) format, though we don't preclude native Debian or Gentoo package formats. The concepts are equally applicable; implementation is left as an exercise for the reader.

Base infrastructure components are in the `firmware-tools` package, detailed previously. Individual updates for specific device classes are split into two (or more) packages: an Inventory and Execution package, and a Payload package. The goal is to be able to provide newer payloads (the data being written into the flash memory parts) separate from providing newer inventory and execution components. In an ideal world, once you get the relatively simple inventory and execution components right, they would rarely have to change. However, one would expect the payloads to change regularly to add features and fix bugs in the product itself.

4.1 RPM Dependencies

Payload packages have a one-way (optionally versioned) RPM dependency on the related Inventory and Execution package. This allows tools to request the payload package, and the related Inventory and Execution package is downloaded as well. Should there be a compelling reason to do so, the Inventory and Execution components may be packaged separately, though most often they're done by the same tool.

Payload packages further Provide various tags, again to simplify automated download tools.

Lets look at the details, using BIOS package for Dell PowerEdge 6850 as an example. The actual BIOS firmware image is packaged in an RPM called `system_bios_PE6850-a02-12.3.noarch.rpm`. This package has RPM version-release `a02-12.3`, and is a `noarch` rpm because it does not contain any CPU architecture-specific executable content.

This package Provides:

```
class BiosPackageWrapper(object):
  def __init__(self, package):
    package.installFunction = self.installFunction
    package.compareStrategy = biosHdr.compareVersions
    package.type = self

  def installFunction(self, package):
    ret = os.system("/sbin/modprobe dell_rbu")
    if ret:
        out = ("Could not load Dell RBU kernel driver (dell_rbu).\n"
               " This kernel driver is included in Linux kernel 2.6.14 and later.\n"
               " For earlier releases, you can download the dell_rbu dkms module.\n\n"
               " Cannot continue, exiting...\n")
        return (0, out)
    status, output = commands.getstatusoutput("""dellBiosUpdate -u -f %s""" %
                     os.path.join(package.path, "bios.hdr"))
    if status:
        raise package.InstallError(output)
    return 1
```

Figure 3: Example wrapper class

```
system_bios(ven_0x1028
_dev_0x0170) = a02-12.3
system_bios_PE6850 = a02-12.3
```

Let's look at these one at a time.

```
system_bios(ven_0x1028
_dev_0x0170) = a02-12.3
```

This can be parsed as denoting a system BIOS, from a vendor with PCI SIG Vendor ID number of 0x1028 (Dell). For each vendor, there will be a vendor-specific system type numbering scheme which we care nothing about except to consume. In this example, 0x0170 is the software ID number of the PowerEdge 6850 server type. The BIOS version, again using a vendor-specific versioning scheme, is A02. All of the data in these fields can be determined programatically, so is suitable for automated tools.

Most systems and devices will have prettier, marketing names. Whenever possible, we want to use those, rather than the ID numbers, when interacting with the sysadmin. So this package also provides the same version information, only now using the marketing short name `PE6850`.

```
system_bios_PE6850 = a02-12.3
```

Presumably the marketing short names, though per-vendor, will not conflict in this flat namespace. The BIOS version, A02, is seen here again, as well as a release field (12.3) which can be used to indicate the version of the various tools used to produce this payload package. This version-release value matches that of the RPM package.

The firmware-addon-dell package provides an ID-to-shortname mapping config appropriate for Dell-branded systems. It is anticipated that other vendors will provide equivalent functionality for their packages. Users generating their own content for systems not in the list can accept the auto-generated name or add their system ID to the mapping config.

Epochs are used to account for version scheme changes, such as Dell's conversion from the Axx format to the `x.y.z` format.

To account for various types of firmware that may be present on the system, we have come up with a list for RPM Provides tags seen in Figure 4. We anticipate adding new entries to this list as firmware updates for new types of devices are added to the system.

The combination pci_firmware/system entries

```
system_bios(ven_VEN_dev_ID)
pci_firmware(ven_VEN_dev_DEV)
pci_firmware(ven_VEN_dev_DEV_subven_SUBVEN_subdev_SUBDEV)
pci_firmware(ven_VEN_dev_DEV_subven_SUBVEN_subdev_SUBDEV)/system(ven_VEN_dev_ID)
bmc_firmware(ven_VEN_dev_ID)

system_bios_SHORTNAME
pci_firmware_SHORTNAME
pci_firmware_SHORTNAME/system_SHORTNAME
bmc_firmware_SHORTNAME
```

Figure 4: Package Manager Provides lines in payload packages

are to address strange cases where a given payload is applicable to a given device in a given system only, where the PCI `ven/dev/subven/subdev` values aren't enough to disambiguate this. It's very rare, and should be used with extreme caution, if at all.

These can be expanded to add additional firmware types, such as SCSI backplanes, hot plug power supply backplanes, disks, etc. as the need arises. These names were chosen to avoid conflicts with existing RPM packages Provides.

4.2 Payload Package Contents

Continuing our BIOS example, the toplevel firmware storage directory is `/usr/share/firmware`. BIOS has its own subdirectory under the toplevel, at `/usr/share/firmware/bios/`, representing the top-level BIOS directory. The BIOS RPM payload packages install their files into subdirectories of the BIOS toplevel directory. Figure 5 shows this layout.

This allows multiple versions of each payload to be present on the file system, which may be handy for downrev'ing. It also allows an entire set of packages to be installed once on a file server and shared out to client servers.

In this example, the actual data being written to the flash is in the file `bios.hdr`. The `package.ini` file contains metadata about the payload described above and consumed by the framework apps. The `package.xml` file listed here was copied from the original vendor package. It contains additional metadata, and may be used by vendor-specific tools. The `firmware-addon-dell` package uses the information in this file to only attempt installing the payload onto the system type for which it was made (e.g. to avoid trying to flash a desktop system with a server BIOS image).

4.3 Obtaining Payload Content

We've described the format of the packages, but what if the existing update tools aren't already in the proper format? For example, as detailed at the beginning of this paper, most vendors release their content in proprietary formats. The solution is to write a tool that will take the existing proprietary formats and repackage them into the `firmware-tools` format.

The `fwupdate-tools` package provides a script, `mkbiosrepo.sh`, which can download files from `support.dell.com`, extract and unpack the relevant payloads from them, and re-package them into packages as we've described here. This allows a graceful transition from an existing packaging format to this new format with little impact to existing business processes. The script can be extended to do likewise for other proprietary vendor package formats.

```
# rpm -qpl system_bios_PE6850-a02-12.3.noarch.rpm
/usr/share/firmware/bios
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02/bios.hdr
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02/package.ini
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02/package.xml
```

Figure 5: Example Package Manager file layout

If this format proves to be popular, it is hoped that vendors will start to release packages in native firmware-tools format. The authors of this paper are already working internally to Dell to push for this change, although there is currently no ETA nor guarantee of official Dell support. We are working on the open-source firmware-tools project to prototype the solution and to get peer review on this concept from other industry experts in this area.

5 Repositories

We recognize that each OS distribution has its own model for making packages avaialble in an online repository. Red Hat Enterprise Linux customers use Red Hat Network, or RHN Satellite Server, to host packages. Fedora and CentOS use Yellow dog Updater, Modified (YUM) repositories. SuSE uses Novell ZenWorks, YaST Online Update (YOU) repositories, and newer SuSE releases can use YUM repositories too. Debian uses FTP archives. Other third party package managers have their own systems and tools. The list goes on and on. In general, you can put RPMs or debs into any of these, and they "just work."

As an optimization, you can package RPMs in a single directory, and provide the multiple forms of metadata that each require in that same location, letting one set of packages, and one repository, be easily used by all of the system types. The `mkbiosrepo.sh` script manages metadata for both YUM and YOU tools. Creation of channels in Red Hat Network Satellite Server is, unfortunately, a manual process at present; uploading content into channels is easily done using RHN tools. Providing packages in other repository formats is another exercise left to the reader.

6 System Administrator Use

Up to this point, everything has focused on creating and publishing packages in a format for system administration tools to consume. So how does this all look from the sysadmin perspective?

6.1 Pulling from a Repository

First, you must configure your target systems to be able to pull files from the online repositories. How you do that is update system specific, but it probably involves editing a configuration file (`/etc/yum.repos.d/`, `/usr/sysconfig/rhn/sources`, ...) to point at the repository, configure GPG keys, and the like. Nothing here is specific to updating firmware.

The first tool you need is one that will match your system vendor, which pulls in the framework packages, which provides the `inventory_firmware` tool.

```
# yum install firmware-addon-dell
```

6.2 Bootstrapping from a Repository

Now it's time to request from the repository all the packages that might match your target system. `inventory_firmware`, in bootstrap mode, provides the list of packages that could exist. Figure 6 shows an example.

We pass this value to yum or up2date, as such:

```
# yum install $(inventory_firmware
-b)
```

or

```
# up2date -i $(inventory_firmware
-b -u)
```

This causes each of the possible firmware Payload packages, if they exist in any of the repositories we have configured to use, to be retreived and installed into the local file system. Because the Payload packages have RPM dependencies on their Inventory and Execution packages, those are downloaded and installed also.

Subsequent update runs, such as the nightly yum or up2date run will then pick up any newer packages, using the list of packages actually on our target system. If packages for new device types are released into the repository (e.g. someone adds disk firmware update capability), then the sysadmin will have to run the above commands again to download those new packages.

6.3 Applying Firmware Updates

`apply_updates` will perform the actual flash part update using the inventory and execution tools and payloads for each respective device type.

```
# apply_updates
```

`apply_updates` can be configured to run automatically at RPM package installation time, though its more likely to be run as a scheduled downtime activity.

7 Proof of Concept Payload Repository

Using the above tool set, we've created a proof-of-concept payload repository [4], containing the latest Dell system BIOS for over 200 system types, and containing Dell PERC RAID controller firmware for current generation controllers. It provides YUM and YOU metadata in support of target systems running Fedora Core 3, 4, and 5, Red Hat Enterprise Linux 3 and 4 (and its clones like CentOS), and Novell/SuSE Linux Enterprise Server 9 and 10. New device types and distributions will be added in the future.

8 Future Directions

We believe that this model for automatically downloading firmware can also be used for other purposes. For example, we could tag DKMS [5] driver RPMS with tags and have the inventory system output `pci_driver(...)` lines to be fed into yum or up2date. A proposal has been sent to the dkms mailing list with subsequent commentary and discussion. This model could also be used for things like Intel ipw2x00 firmware, which typically is downloaded separately from the kernel and must match the kernel driver version.

9 Conclusion

While most sysadmins only update their BIOS and firmware when they have to, the process

```
# inventory\_firmware -b
system_bios(ven_0x1028_dev_0x0170)
bmc_firmware(ven_0x1028_dev_0x0170)
pci_firmware(ven_0x8086_dev_0x3595)/system(ven_0x1028_dev_0x0170)
pci_firmware(ven_0x8086_dev_0x3596)/system(ven_0x1028_dev_0x0170)
pci_firmware(ven_0x8086_dev_0x3597)/system(ven_0x1028_dev_0x0170)
...
```

Figure 6: Running `inventory_firmware -b`

should be as easy as possible. By utilizing OS tools already present, BIOS and firmware change management becomes just as easy as other software change management. We've developed this to be Linux distribution, hardware manufacturer, system manufacturer, and update mechanism agnostic, and have demonstrated its capability with Dell BIOS and PERC Firmware on a number of Linux distributions and versions. We encourage additional expansion of the types of devices handled, types of OSs, and types of update systems, and would welcome patches that provide this functionality.

10 Glossary

Package: OS standard package (`.rpm`/`.deb`)

Package Manager: OS standard package manager (`rpm`/`dpkg`)

Repository Manager: OS standard repository solution (`yum`/`apt`)

References

[1] Firmware-tools Project
Home page: `http://linux.dell.com/firmware-tools/`
Mailing list: `http://lists.us.dell.com/mailman/listinfo/firmware-tools-devel`

[2] Python Generator documentation
`http://www.python.org/dev/peps/pep-0255/`

[3] Libsmbios Project
Home Page: `http://linux.dell.com/libsmbios`
Mailing list: `http://lists.us.dell.com/mailman/listinfo/libsmbios-devel`

[4] Proof of Concept Payload Repository
Home Page: `http://fwupdate.com`

[5] DKMS Project
Home Page:
`http://linux.dell.com/dkms`
Mailing list:
`http://lists.us.dell.com/mailman/listinfo/dkms-devel`