Deconstructing Linux udev Rules

A TundraWare Inc. Technical Note

Author: Tim Daneliuk (tundra@tundraware.com)

Version: $Id: Deconstructing_Linux_udev_Rules.rst,v 1.131 2015/02/23 16:05:40 tundra Exp $

Précis

udev is one of those pieces of Linux that is fairly well documented and not very well understood. This note isn't intended as a general introduction to writing udev rules, but, rather, a brief introduction to the topic by way of specific example. Most tutorials on this subject only provide you with templates along the lines of "Here's how to do <fill in the blank> with udev." The approach here is more about "here's why rules work the way they do". Hopefully, you'll find it useful.

> **Warning**
> The examples and descriptions below assume you are running as root. Most of the commands described will either return nothing or will not work at all unless you are root.
> Because you are root and are making device-level changes, you can wholly and completely Bugger Up your machine (it took me 6 years of graduate school to learn to use that term like a Real Computer Scientist (tm)).
> So ... do the smart thing. Don't practice this stuff on machines that matter. Better still, spin up some VMs and play with it there.
> You have been warned. We do provide tech support for this stuff. For ordinary work we charge the usual rates. For fixing things you screwed up, $10,000/hour ... prepaid.

Why Bother With udev?

There are many clever uses for udev documented on the Web, but the most common use is to ensure that when you connect a device - disk, tape, usb thumbdrive, camera... whatever - to a Linux system, that device shows up with the same name every time.

Original Unix derivatives had a static tree of devices the system could support. This was encoded in the /dev file tree hierarchy. This was pretty inflexible in the face of devices being added- and removed from the system as it ran. For this reason, modern device handling in Linux and most other Unix derivatives is *dynamic* - the content of /dev changes to reflect the actual state of the system as things get connected or disconnected. (Exactly how this is done is outside the purpose of this document, but if you care, investigate how the Linux /sys filesystem works.)

Our Example Problem

While the example below is "cooked", it is very much rooted in real world udev applications. We want to do the following things:

- Identify a specific disk no matter what name it was assigned name under /dev.
- Create a symbolic link to that disk so that - no matter what it's name under /dev/ might be at the moment - the symbolic link is always the same.
- Change the user and group ownership of that disk to something other than the default (root:disk).

- Set specific permissions for the disk.
- Create a corresponding "raw" character device under `/dev/raw` associated with our disk above.

Where Do `udev` Rules Live?

User created rules - well, created by `root`, actually - are found in `/dev/udev/rules.d`. If you look there, you'll see that the files there begin with numbers like `50` or `60`. `udev` reads rules in *lexical order*. That means it reads the `50...` file before the `60...` file before the `70...` file and so on. This is important because you have to be careful to insert your rule in early enough in the lexical order so that it can override any subsequent defaults.

Unfortunately, because of the way `udev` works, rules read later in the lexical order can also *override earlier rules* if we're not careful. We'll see an example of this below, and how to fix it.

In our case, we'll create our rules in the file `15-ExampleRules.rules` which should pretty much guarantee that our rules will be the first ones read.

How Does `udev` Read Rules?

When `udev` first starts, or any time it is informed that rules have been changed, it first reads a set of system-wide default rules in `/lib/udev/rules.d/`. Then it reads the rules in `/etc/udev/rules.d`. If you name your own rule file the same as one of system-wide rules, yours will take precedence. There is also a way to install "temporary" rules, but the location for such rules is distro-specific.

Ordinarily, the running `udev` daemon is automatically informed that a rule file has changed and it will reread them all again when this happens. You can also force a rule reload with:

```
udevadm control --reload-rules
```

Another way to do this is to restart the `udev` daemon or reboot to get the latest rules read in. Note that the daemon restart procedure is also distro-specific, so you'll have to figure out what works on your system. Rebooting is not distro-specific and can always be accomplished by removing all power sources. This is not a recommended best practice unless there is loud knocking at the door and you have to leave *really* fast.

Our Example Rules

We need two rules to achieve our goals above. Notice that the rules below are broken across multiple lines to make them more readable. In the actual file, they entire rule is put on one line. It is possible to break rules across lines, but you have to ensure that you follow the syntax that `udev` expects. To keep things from mysteriously breaking, it's just easier to keep the entire rule on one line:

```
ACTION=="add", KERNEL=="sd*",
               PROGRAM=="/sbin/scsi_id --whitelisted /dev/$name",
               RESULT=="VBOX_HARDDISK_VB5f712327-2bb4be0c",
               SYMLINK+="my_fine_disk01",
               OWNER:="3009", GROUP:="421", MODE:="0600",
               RUN=="/bin/raw /dev/raw/raw1 /dev/$name"

ACTION=="add", KERNEL=="raw1",
               SYMLINK+="rmy_fine_disk01",
               OWNER:="3009", GROUP:="421", MODE:="0600"
```

What Does All This Mean?

Rules are made up of key-value pairs separated by an operator. These key-value pairs are separated by commas. The value (right side) is double quoted. Pay attention to the operators because they too mean something. For example, `=` (assignment) is not the same thing as `==` (checking for equality).

Key-value pairs either *match* or *assign*. Match key-value pairs check to see if a particular thing "matches" what we're looking for. Think of them as `if` statements in a programming language.

Assignment key-value statements take some sort of action *usually on the thing that was previously matched*. But, you're not restricted to this. It's entirely possible to write a rule that operates on something completely unrelated to the matched condition. For instance, you could write a rule that says, *reboot the computer every time my little brother plugs in his favorite thumbdrive*. This is, however, considered Very Bad Manners and may get you sent to your room without dinner. You may, however, have a career developing websites for US government healthcare initiatives.

Let's take each rule apart, one key-value pair at a time:

- `ACTION=="add", KERNEL=="sd*"`

  `KERNEL=="sd*"` matches any time the kernel emits a message with the string `sd` followed by anything. For example, the kernel sending messages about `sda`, `sdb`, `sdc` and so on would all match. But we only want to tune in to these messages when a drive is being *added*. We want to ignore other kernel messages with `sd*` in them when drives are removed or reporting an error. So, we also include the `ACTION="add"` key value pair.

  Effectively, this lets us look at every drive being added to the system, so we can spot the one we're looking for.

- `PROGRAM=="/sbin/scsi_id --whitelisted /dev/$name", RESULT=="VBOX_HARDDISK_VB5f712327-2bb4be0`

  This is an assignment or "action" key-value pair. Each time the matches above are true, the `scsi_id` program is then run to do further checking. `$name` in this case is the exact string the `KERNEL` matching triggered on. So, if it was `hdx`, then the command here would be:

  ```
  scsi_id --whitelisted /dev/hdx
  ```

  If `scsi_id` returns a string that matches `VBOX_HARDDISK....`, then the `RESULT` key-value match is also true. In other words, we're looking for a drive that's being added *that has a specific unique ID*. (On SAN-connected systems, this is called the drive's "World Wide ID" or just `wwid`.)

  Note

  When you have multiple match key-value pairs in a `udev` rule, *all* of them have to be true for the rule to be invoked. In order for our rule to be applied, the kernel has to report that something called `sd*` has been seen AND it's being *added* AND it's ID is the string `VBOX.....` If ANY of these conditions are not met, the rule is not applied.

  Unfortunately, the command to use to get the unique ID of a drive varies by distro. On `CentOS` and `Redhat` the command is `scsi_id`. On `Ubuntu` and `Mint` it's `scsidev -s` (you may have to install the `scsitools` package to get this utility). Other distros may have other ways of doing this.

Warning

If you're running `Linux` as a guest under `VMWare`, beware! By default, recent versions of `VMWare ESX, Workstation,` and `Player` do *not* enable unique disk identification like physical machines (and `Oracle VirtualBox`) do. You can read all about it here:

http://www.dizwell.com/wiki/doku.php?id=blog:the_case_of_vmware_and_the_missing_scsi_id&s%5B%5D=scsi

Basically, you have to set `disk.EnableUUID = "TRUE"` in your virtual machine's `.vmx` file.

Why are we bothering with all this? When disks are added and/or removed from a system, *they are NOT guaranteed to be assigned to the same device node in* `/dev/`. Your drive could show up as `/dev/sdh` one time and `/dev/sdx` the next. This is the price we pay for having a dynamic device system that allows hot swapping USB devices on your laptop or live presentation of SAN storage to a server. We thus have to use something that *uniquely* identifies the drive every time.

Many of the tutorial examples of this on the World Weird Web show this "uniqueness" test using the major- and minor device numbers. This is an Officially Bad Idea (tm) because there is no guarantee that the same device will get the same major/minor device numbers every time. These numbers are generated by the kernel and are based on the order of driver/device loading, the next free number available to the kernel, and the cosmic ray count in Lower Goscratchistan. Only the `wwid` (or an equivalent `UUID`) is guaranteed to be unique, so that's what we used here.

If we got this far, it means that all our matching tests were successful: we've found the drive we're looking for. Now we can do what we set out to do in the first place:

- `SYMLINK+="my_fine_disk01"`

  This creates a symbolic link to whatever device mountpoint ended up being assigned to our disk by the operating system. In other words, if the mount point changed from `/dev/sdd` to `/dev/sdl` after a reboot, our rule would figure it out and point the link named `my_fine_disk01` at that mountpoint.

  Notice the use of the `+=` operator here instead of the more usual `=`. The use of the `+=` operator means, *"Add another symbolic link to this mountpoint."*

  The real purpose of this is to provide a *consistent device name* for software to use when referencing this disk. Say I'm writing an application. We don't have to know where the disk is mounted. We just have to always refer to `/dev/my_fine_disk01` and let all this `udev` magic do the hard stuff. Remember, it's our job as Snotty Systems Engineers (tm) to relieve Applications Programmers of as much thinking as possible. Really, it is. Look it up in the job description.

Note

Why use a symlink? Why not just rename the mountpoint. It *is* possible to do this with `udev` with the `NAME+=...` key-value construct. It's best not to do this because you lose visibility into the underlying device name when you do this.

It's handy to know that the actual device name is, say, `sdk`. For example, ejecting SAN-attached storage requires you to send things to `/sys/block/sdk/device/delete`. If you overwrite `/dev/sdk` with `my_fine_disk01`, it's not immediately clear what the underlying device actually is. A symbolic link covers both bases.

The only reason to not use a symlink and to actually rename the node is if you happen to have software that does not properly deal with symlinks. People that write such awful software are called "n00bs", "sloppy", "bozos" or, possibly just, "programmers at Computer Associates".

- `OWNER:="3009", GROUP:="421", MODE:="0600`

  These key-values pairs change the owner, group, and permissions on whatever mount-point our drive ended up on. That way, we're assured that, no matter where the drive gets mounted, it will have ownership and permissions we expect.

Here the use of the `+=` operator means something different. It means, *"I am the final rule in this matter. No subsequent rule can change this setting."* That's how we prevent rules that are read after us (ones with higher numbers in their name) from overriding what we want. For example, we've seen instances of systems using `GROUP=` that then got overriden by a later default filesystem rule. This then reset group ownership to `disk`. Using `:=` instead, fixed this.

One other thing here: Notice the use of numeric values for `UID` and `GID`. You *could* use the actual user- and group names here. In fact, most `udev` tutorials show it this way. It is a *bad idea*, especially in large, high complexity datacenters. When a machine boots that uses remote authentication like `ldap`, you cannot guarantee you'll have access to the authentication server at the time `udev` wants to set these ownership and permissions values. This can happen when you have slow, crufty networks that take a long time to nail up a connection between a server and its `ldap` authority. The numeric values are always right (unless some genius is in the habit of changing them often in `ldap`). You'll see the proper user- and group names on your mountpoint when `ldap` connectivity is established. If you want to know what the numeric values for user- and group are, do this:

```
id username
```

- `RUN=="/bin/raw /dev/raw/raw1 /dev/$name"`

  Now we're ready to create a raw character device associated with our matching drive. If you don't know what this is, you probably don't need it. If you ever work with database servers, you'll find out soon enough :)

  Basically, the command above magically creates a raw character device of `/dev/raw/raw1` associated with /dev/sd... (our mountpoint).

  "But why", you may ask, "are you using the `RUN==` construct? Isn't that what `PROGRAM==` does?" Not exactly, Grasshopper. `PROGRAM==` *always* runs regardless of prior matching. That's because `PROGRAM==` is itself a *matching* key-value construct. It's used to *figure out* whether a match has taken place (by means of it populating `RESULTS`). It thus has to run every time. `RUN==`, on the other hand, *only* runs if all prior matching has been successful.

  Why is that important here? Say we boot the system, and the kernel discovers drives `/dev/sdh, /dev/sdi,` and `/dev/sdj` and let's suppose that the first one has the matching `wwid`. With `RUN==` the raw character device will only be created when the full set of matching occurs - i.e., When the kernel reports the addition of `/dev/sdh`. But if you use `PROGRAM==`, the raw device will be associated *every time the kernel reports a new "/dev/sd*"*. The last one to be reported will "win". In this case, that means `/dev/raw/raw1` will be associated with `/dev/sdj` - not what we want here.

With that under our belts, the second rule should be pretty simple to understand:

- `ACTION=="add", KERNEL=="raw1"`

  We want to match any time the kernel reports something called `raw1` being added to the system. Oh, wait, we just did that at the end of the previous rule.

- `SYMLINK+="rmy_fine_disk01"`

  Let's symlink `/dev/raw/raw1` to `/dev/rmy_fine_disk01`. It is a time honored `Unix` convention that the raw device name be the same as the actual device with an `r` prepended to it. You don't have to do this, but if you don't, people will hate you, your dog will probably run away, and your ex-wife will show up again ... and no one wants that.

  The DBAs can then configure their database engines to look for the symlink name and never worry about what the underlying node mapping is for the raw device. Just as with Applications Programmers, we Snotty Systems Engineers (tm) are required - by

law - to make things as easy as possible for DBAs. This one isn't actually in the job description, it's just an act of kindness.

- `OWNER:="3009", GROUP:="421", MODE:="0600`

  And finally, as before, we set ownership and permission, this time for the raw character device, not the associated block device (which we took care of in the first rule).

Final Thoughts

Obviously, you'd have to have another pair of rules for each additional disk you want to manage this way. Adding another disk would be a matter of changing the unique ID for the `RESULT` field of the first rule. You'd also have to change any references to `my_fine_disk01` and `raw1`.

You'll also have to change the rules above to the program used to check for a unique `wwid` or `UUID` on your particular distro.

If you want to know the current state of what raw devices exist do this:

```
raw -qa
```

For reasons that are not entirely clear (to me anyway), the `raw` command only knows how to create raw devices whose names begin with `raw`, go figure.

Another way to get a unique ID for a device is to tail your system log (`tail -f /var/log/messages` or `tail -f /var/log/syslog`) and watch what happens when you plug your device into, say, a USB port.

If you want to know all the attributes `udev` knows about a particular device, use this, substituting your device for `/dev/sdd`

```
udevadm info --query=all --name /dev/sdd 2>&1| less
```

The output of this command can be helpful in figuring out just which attributes and values you need to get to a running rule.

Finally, you can test your rules to see what is matching, again substituting for `/block/sdd`:

```
udevadm test /block/sdd 2>&1| less
```

Most of what is described above applies analagously for non-disk devices like cameras and scanners. The principles are pretty much the same.

Copyright And Licensing

Document Information

You can find the latest version of this document at:

http://www.tundraware.com/TechnicalNotes/Deconstructing-Linux-udev-Rules

A pdf version of this document can be downloaded at:

http://www.tundraware.com/TechnicalNotes/Deconstructing-Linux-udev-Rules/Deconstructing_Linux_udev_Rules.pdf

This document produced with `emacs`, `RestructuredText`, and `TeXLive`.