Thinking About Binary Compatibility and CentOS Stream

Pat Riehecky

CentOS Dojo May 2021

FERMILAB-SLIDES-21-020-SCD

14 May 2021

Disclaimers

Pat Riehecky is **not** speaking as official spokesperson for:

- Fermi Research Alliance
- Fermi National Accelerator Laboratory
- US Department of Energy

Neither the United States nor the United States Department of Energy, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any data, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

This is not an endorsement of any technology, software, service, or organization.

This presentation is an act of *research*, rather than a prescription of future practices.

The Road Ahead in this Presentation

Nothing is more boring than a presentation where someone reads the slides to you…

Except perhaps when you already know most of what is on each and every slide.

Alas, I couldn't think of a better way to ensure a uniform understanding of the topics ahead, simplify translation for non-english speakers, and provide a way to use the slide deck as quick reference when people are under time pressure.

For the Ops people, this presentation may help you put the 'Dev' in '*Dev*Ops'

For the Dev people, this presentation may help you put the 'Ops' in 'Dev*Ops*'

I've never worked at a place where folks complained about how thorough the communication is between Dev and Ops. Good communication is hard, *just like good coordination*.

Will my RHEL application run on CentOS Stream?

In practice, the answer to this is a function of the RHEL ABI/API guarantee.

https://access.redhat.com/articles/rhel8-abi-compatibility

When using items on the RHEL approved list: if it runs on RHEL it should run on CentOS Stream without any modifications. Any incompatibilities there are bugs that need to be addressed. The “stable” ABI/API is about consistency or dependability.

This is a presentation about **User Space**.

Note: the RHEL Kernel is a whole separate discussion with a whole separate document outlining its ABI/API guarantees, a nice presentation yesterday – also a blog post.

https://developers.redhat.com/blog/2018/03/28/analyzing-binary-interface-changes-linux-kernel/

The RHEL8 ABI/API guarantee

There are four levels outlined in the document:

Level 1: These are dependable within the lifetime of a major release and they are also dependable across the next two major releases (glibc, libxml2, etc)

Level 2: These are scoped within the lifetime of a single major release, but consistent within it (**this is the default**)

Level 3: Each component will specify a lifetime during which the ABI/API will be supported (AppStream/Modularity)

Level 4: No compatibility is guaranteed (gnome-desktop, libcgroup, libdrm, etc)

For the definitive rules and list of what binaries are in each group, you'll want to read the official documentation from Red Hat.

Will my CentOS Stream application run on RHEL?

In practice, this too is a function of the RHEL ABI/API consistency guarantee, but the implementation is a bit different.

https://access.redhat.com/articles/rhel8-abi-compatibility

In order to answer this, we need to look at how binaries actually run on a Linux system and understand what that means in a RHEL context.

Anatomy of an ELF binary as a package

For right now, we can treat an ELF binary as a type of runtime package. This should help us stay focused. After all, a binary RPM generally contains a collection of ELF binaries. We will be ignoring other binary types and ignoring ELF support/behavior on non Linux platforms.

The bits we care about at this time are the **binary payload** and the **library linkage/symbols**.

The **binary payload** contains the instructions to execute. If the CPU doesn't have the requested instructions, the binary will SEGFAULT once it hits an unknown instruction. RPM is not looking for these types of incompatibilities – I think that is a good thing. Microarchitectures (like `x86_64-v2`) are interesting, but out of scope today. Generally speaking, the '`x86_64`' in the package `arch` is considered sufficient to confine it to CPUs it can use. For this presentation that is good enough for what we plan to explore.

Library linkages and symbol exports are a bit more complex to summarize and will consume the bulk of this presentation.

Anatomy of an ELF binary linkage

There are a number of ways to review the dynamic libraries a binary uses and what various libraries provide.

- `ldd (-v)`
- `objdump (-p / -T)`
- `readelf (-r / -s)`

NOTE: the `ldd` command will potentially execute elements of the object to achieve symbol linkage and is thus less safe on untrusted binaries than the alternatives. The manpages on `ldd` have more information on this. It is fascinating! You should read it!

You can also use `abipkgdiff` (from `libabigail`) to look at changes between two RPMs or binary objects.

Anatomy of an ELF linkage: an example with xz

`xz` is a Compatibility Level Two package.

https://access.redhat.com/articles/rhel8-abi-compatibility

“Compatibility level 2

APIs and ABIs are stable within the lifetime of a single major release. Compatibility level 2 application interfaces will not change from minor release to minor release and can be relied upon by the application to be stable for the duration of the major release. Compatibility level 2 is the default for packages in Red Hat Enterprise Linux 8. Packages not identified as having another compatibility level may be considered compatibility level 2.”

Anatomy of an ELF linkage: an example with xz

As part of its inventory/dependency system, RPM utilizes `objdump` to review the linkage of each binary.

```
$ objdump -p /usr/bin/xz  # from EL8
/usr/bin/xz:     file format elf64-x86-64
[snip]
Version References:
  required from libpthread.so.0:
    0x09691a75 0x00 05 GLIBC_2.2.5
  required from libc.so.6:
    0x0d696917 0x00 07 GLIBC_2.7
    0x06969197 0x00 06 GLIBC_2.17
    0x09691974 0x00 04 GLIBC_2.3.4
    0x09691a75 0x00 03 GLIBC_2.2.5
    0x096a2561 0x00 09 GLIBC_2.32
  required from liblzma.so.5:
    0x05e02812 0x00 08 XZ_5.2
    0x05e02810 0x00 02 XZ_5.0
```

The examples in this presentation will often switch back and forth between `objdump` and `readelf` style reporting.

I find a detailed investigation into the symbols is easier with `readelf`. Most system tools use `objdump`. You should use a tool you like and make sure to understand its limitations/risks.

Anatomy of an ELF linkage: an example with xz

With `readelf` we can easily see exactly which foreign functions (library calls) are listed and specific information about them.

```
$ readelf -r --wide /usr/bin/xz | awk '{print $5}' | sort -u | grep XZ_
[snip]
lzma_stream_encoder_mt_memusage@XZ_5.2
lzma_stream_flags_compare@XZ_5.0
lzma_stream_footer_decode@XZ_5.0
lzma_stream_header_decode@XZ_5.0
lzma_version_number@XZ_5.0
lzma_version_string@XZ_5.0
```

Anatomy of an ELF linkage: an example with xz

Putting together what we've uncovered about the `xz` binary on my test EL8 system:

- `liblzma.so.5` is used to resolve a number of symbols
- Some of those symbols are marked `XZ_5.0` and some as `XZ_5.2`
  - `lzma_stream_encoder@XZ_5.0`
  - `lzma_stream_encoder_mt@XZ_5.2`

Thankfully, `liblzma` follows the standard convention where the `.5` in `.so.5` corresponds to library version 5.

Any system that provides the required ABI symbols can run this binary.

Reminder: `liblzma` is a "Compatibility Level 2" library in RHEL 8.

Compatibility Level 2 in practice with liblzma

EL7.0 `xz` on an EL8.0 host:

```
$ ./xz
./xz: /lib64/liblzma.so.5: version `XZ_5.1.2alpha' not found
```

Specifically: `lzma_stream_encoder_mt@XZ_5.1.2alpha`

EL8.0 `xz` on an EL7.0 host:

```
$ ./xz
./xz: /lib64/libc.so.6: version `GLIBC_2.32' not found
./xz: /lib64/liblzma.so.5: version `XZ_5.2' not found
```

Specifically: `lzma_stream_encoder_mt@XZ_5.2`

There is nothing necessarily unexpected here. Between major releases the `liblzma` symbols are permitted to change. Furthermore, the addition of new symbols to `glibc` in EL8 is expected and generally a good thing. These are new features!

A short detour into Level 1 (backwards compatible)

`glibc` and `libpthread` are Level 1 libraries, so it is worth a brief detour to see what happened.

Opening up the `xz` binary linkage :
In EL8 we see the addition of `pthread_sigmask@GLIBC_2.32`
In EL7 the binary has `pthread_sigmask@GLIBC_2.2.5` instead

A look at the glibc changelog for `libpthread` shows:
* On Linux, the functions `pthread_attr_setsigmask_np` and `pthread_attr_getsigmask_np` have been added. They allow applications to specify the signal mask of a thread created with `pthread_create`.

Which in turn cascades down to `pthread_sigmask` causing the symbol addition.
The older symbol remains in the EL8 `glibc` (via `libpthread`), thus Level 1 rules are satisfied.

Compatibility Level 2 in practice with liblzma

Looking at `lzma_stream_encoder_mt`

In this instance the solution is fairly trivial. As I understand the `liblzma` changelog, the symbol introduced in `xz-5.1.2alpha` was stabilized in `xz-5.2` and saw no changes to the **arguments** or **return structure**.

In the end, the ABI of a binary is basically: the arguments, the return structure, the symbol name, and the namespace. In this instance three are the same - the namespace is what is different.

The code using this symbol name can be recompiled and relinked as is and should work exactly as expected.

But this isn't a conversation about porting software from EL7 to EL8.

This is a conversation about ABI consistency. To keep matters simple, lets stick with `liblzma` and look at EL7 where we have a lot of history to examine.

Compatibility Level 2 in practice with liblzma on EL7

In 2016 `liblzma` was rebased from `5.1.2-12alpha` to version `5.2.2` in EL7.

The upstream source code has the symbols tagged as `XZ_5.2`. However, the EL7 source is patching the code to instead keep those symbols (`lzma_stream_encoder_mt` and `lzma_stream_encoder_mt_memusage`) marked with the `XZ_5.1.2alpha` namespace.

This rebase also adds some new symbols which it is tagging as a part of the `XZ_5.2.2` namespace (which matches the version of `liblzma` where they were introduced into EL7).

Compatibility Level 2 in practice with liblzma on EL7

xz-5.2.2-compat-libs.patch →

this patch is a concise (and near perfect) example of Level 2 compatibility (author Pavel Raiskup?)

`liblzma/api/lzma/container.h` in `5.1.2alpha` lists the `lzma_stream_encoder_mt` and `lzma_stream_encoder_mt_memusage` as a part of their 'UNSTABLE' symbol namespace.

What does that mean? For `liblzma` upstream, These symbols are part of the `5.2` namespace. For EL7, they have to stay in `5.1.2alpha` for Level 2 compatibility to be honored.

This slide is the centerpiece of this presentation

```
We provided two 5.1.2alpha symbols (lzma_stream_encoder_mt and
lzma_stream_encoder_mt_memusage) before we updated to xz-5.2.2-1 in RHEL7.3.

Those symbols did not change ABI in 5.2.2 so it should be safe to provide
(except for 5.0 and 5.2 symbols) also the two 5.1.2alpha symbols and
use 5.1.2alpha symbol version as parent for 5.2.

For better reasoning look at container.h in 5.1.2alpha -- those two symbols
were for testing purposes only, and thus not considered to be API/ABI.

diff --git a/src/liblzma/liblzma.map b/src/liblzma/liblzma.map
index f53a4ea..9c3002a 100644
--- a/src/liblzma/liblzma.map
+++ b/src/liblzma/liblzma.map
@@ -95,7 +95,13 @@ global:
        lzma_vli_size;
 };

-XZ_5.2 {
+XZ_5.1.2alpha {
+global:
+       lzma_stream_encoder_mt;
+       lzma_stream_encoder_mt_memusage;
+} XZ_5.0;
+
+XZ_5.2.2 {
 global:
        lzma_block_uncomp_encode;
        lzma_cputhreads;
@@ -105,4 +111,4 @@ global:
 local:
        *;
-} XZ_5.0;
+} XZ_5.1.2alpha;
```

Compatibility Level 2 in practice with liblzma on EL7

The RPM dependencies for the `xz` rpm are similarly interesting.

```
rpm -qp --requires xz-5.1.2-8alpha.el7.x86_64.rpm |grep liblzma
liblzma.so.5()(64bit)
liblzma.so.5(XZ_5.0)(64bit)
liblzma.so.5(XZ_5.1.2alpha)(64bit)

rpm -qp --requires xz-5.2.2-1.el7.x86_64.rpm |grep liblzma
liblzma.so.5()(64bit)
liblzma.so.5(XZ_5.0)(64bit)
liblzma.so.5(XZ_5.1.2alpha)(64bit)
liblzma.so.5(XZ_5.2.2)(64bit)
```

The new RPM correctly determines it needs the new symbols **without the need to set an explicit version of `liblzma`** under `Require`.

Compatibility Level 2 with liblzma in conclusion

We've seen the source get rebased and have symbols added. The `liblzma` maintainer was required to honor the previous symbol names/namespaces and patched the old names back in.

We've seen that RPM correctly identifies the new symbols when used by new binaries.

We've also seen new binaries get the stable symbol names when built against the new libraries.

And we've seen that Level 2 libraries can in fact break between major releases.

The ABI/API guarantee is part of RHEL, and it worked as expected.

The ABI/API guarantee in CentOS Stream

I think of CentOS Stream as the continuous delivery repository for RHEL. RHEL updates are published in bundles/batches/point releases on scheduled intervals. Stream is published now.

The packages in CentOS Stream are headed into RHEL. To my mind this means they **must** be suitable for running on RHEL.

Packages in CentOS Stream that violate the *RHEL* ABI/API guarantee are **not suitable** for running on RHEL.

Therefore, breakage of the *RHEL* ABI/API shouldn't happen in Stream.

If a CentOS Stream package violates the *RHEL* ABI/API guarantee the maintainer must fix it. This is not an optional step or something to be considered as an afterthought. If it doesn't get fixed, it changes in RHEL and *if it changes in RHEL it violates RHEL contracts*.

Packages in CentOS Stream that break the *RHEL* ABI/API are serious bugs.

The ABI/API guarantee in CentOS Stream: New symbols

Maintenance of the existing ABI/API guarantees is well understood.

New symbols/features/ABI/API/etc are where things get complex.

Symbols not on the RHEL guarantee list are not guaranteed, but this doesn't mean they *will* change before going into RHEL – just that they *can* change. But how?

Changes to the arguments or return types don't feel like something you'd change from upstream. Similarly, the names of the symbols are selected by the upstream project. The namespace seems to be our only option for change here.

Use of new symbols presents a risk, but there are several mitigation strategies.

For developers they all boil down to "know what you are using".

For operations it is a question of how to provide extra libraries in a clean manner.

Mitigation: Developers

What should developers do and think about?

Mitigation: Developers

Step 1 is know what you are using.

This is already true for RHEL8 AppStream/Modularity.

If your application links against MariaDB/PostgreSQL you already must keep track of your version usage.

Similarly with `gcc`, `llvm`, `python`, `perl`, `rust`, `swig`, and more and more…..

Big questions:

Are the library features you selected “new” (ie not yet in RHEL)?

If they are, do you *need* them? They answer can be ‘yes’, but know why.

Mitigation: Developers : libabigail

Consider using `abipkgdiff` (from `libabigail`) to track your package ABI changes over time. The output is even more detailed with the debuginfo installed!

```
$ abidiff 5.1.2-alpha/usr/lib64/liblzma.so.5  5.2.2/usr/lib64/liblzma.so.5
Functions changes summary: 0 Removed, 0 Changed, 0 Added function
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable
Function symbols changes summary: 0 Removed, 3 Added function symbols not referenced by debug info
Variable symbols changes summary: 0 Removed, 0 Added variable symbol not referenced by debug info
 3 Added function symbols not referenced by debug info:
  [A] lzma_block_uncomp_encode@@XZ_5.2.2
  [A] lzma_cputhreads@@XZ_5.2.2
  [A] lzma_get_progress@@XZ_5.2.2
```

Additionally, `libabigail` provides the `abicompat` command for further review of a library's dependencies. **The reports can be super helpful** for determining *compatibility* and *tracking changes* over time.

Mitigation: Developers : symbols you choose

How do you tell if a feature is “new”?

Look at the documentation for your library

- Is the feature you’re using listed as coming with a newer version of the library than you’ve got installed? If so, this feature was backported to the EL source. When though?
- Was the library rebased “recently” in Stream? What does “recently” mean here?

Easy way : test your binary with RHEL UBI

The RHEL Universal Base Image (ebook link) is real RHEL. UBI is free to download and redistribute. “No subscription, login, or even registration is required for the UBI images.”

If you picked a set of software behaviors for your application,
know why you picked them.

Mitigation: Developers : RPM symbols

Let RPM decide your binary dependencies.

Look at our `xz` rpms again,

```
rpm -qp --requires xz-5.2.2-1.el7.x86_64.rpm |grep liblzma
liblzma.so.5()(64bit)
liblzma.so.5(XZ_5.0)(64bit)
liblzma.so.5(XZ_5.1.2alpha)(64bit)
liblzma.so.5(XZ_5.2.2)(64bit)
```

RPM knows what we need. Setting a specific version of a binary package to required for a binary RPM is counterproductive. If you recompile the Source RPM it should generate the dependencies it needs. If you hard code a specific version of a binary you need, you are reducing the functionality. Do you need `liblzma-5.2.2` or the symbols it provides?

Mitigation: Developers “surprise symbols”

Looking back at our `xz` binary and the `pthread_sigmask` symbol selection. The `liblzma` library didn’t specifically choose the new or old symbol. It just used the `pthread` headers on the system.

How do you force a specific symbol when multiple versions exist?

In GCC you can use this:

```
__asm__(".symver pthread_sigmask,pthread_sigmask@GLIBC_2.2.5");
```

to select a specific version rather than let the compiler/linker/library choose.

https://developers.redhat.com/blog/2019/08/01/how-the-gnu-c-library-handles-backward-compatibility/

Mitigation: Developers “surprise symbols”

The `glibc` library is probably the most likely to be a source of ‘surprise symbols’.

We can gather some useful stats from EL7 (in year 7 now) and security errata only maintenance. New features in EL7 at this point would be a surprise.

The `/lib64/libc.so.6` library from `glibc-2.17-323.el7_9.x86_64` contains **2125** symbols which are scoped to multiple namespaces. Some of them are ‘unscoped’ versions of the ‘scoped’ version.

The original `/lib64/libc.so.6` published with EL7.0 (`glibc-2.17-55.el7.x86_64`) contains **2100**.

Testing Methodology (better methodology: `abidiff --no-default-suppression`):

```
readelf -s --wide ./lib64/libc.so.6 | awk '{print $8}' | sort -u | cut -d '@' -f1
```

Mitigation: Developers things you “can’t” do - extraction

You “cannot” directly extract a specific symbol from a binary library. A binary library is a blob of instructions, not a file system directory of symbols or `tar` archive. This is something I know but often forget.

The binary has been optimized and the related jumps are relative to the location within the blob. If you disassemble it, it may be possible to reassemble a stub library with just a subset.

But if you’ve gotten this far down the chain a better question is, “Is this feature worth it?” or “Can I just ship the new library?” If you need a new feature, shipping the whole library makes a lot more sense since upstream probably tested it that way.

If you want to build a new binary that just contains the symbol you want, you can always edit the source and compile what you desire. And that is probably a lot less work than using a disassembler on an existing binary.

Mitigation: Developers - DANGEROUS THINGS

ELF objects support an `rpath` value that you can use to tweak the symbol resolution. If you need this, read up on `DT_RPATH` vs `DT_RUNPATH`.

In RHEL, RPM will not let you package binaries with encoded `rpath` values! This is a good thing! Packages should either ship the libraries they need and put them in a rational place, or depend on the libraries already in a rational place.

If you are using `rpath`, you claim to know better than the system linker.

Are you sure that is true? Spack does this because it knows where the libs go.

You can use `patchelf` to mess about with `rpath` settings on existing binaries.

Never ever do this on a file owned by a package manager.

Updates to the package will lose the change.

The installation is no longer repeatable by the package manager.

Mitigation: Operations

What should operations do and think about?

Mitigation: Operations

Use technologies that are linkage aware: ie package managers – not `make install`.

RPM uses `objdump` to evaluate what you need. It is tracking at "namespace/version" level, not each individual binary symbol. A badly behaving library may add symbols to an existing namespace. An older/existing copy might have that namespace, but not the new symbols. Adding a feature and not changing the version is a bad practice. File bugs on this if it happens!

The fastest way to diagnose this is probably via `ldd` (see note on safety in manpage)

```
$ ldd -r xz  # el7 binary on el8
[snip]
undefined symbol: lzma_get_progress, version XZ_5.2.2
undefined symbol: lzma_stream_encoder_mt, version XZ_5.1.2alpha
undefined symbol: lzma_stream_encoder_mt_memusage, version XZ_5.1.2alpha
undefined symbol: lzma_cputhreads, version XZ_5.2.2
```

Mitigation: Operations workarounds

If you find yourself needing these workarounds, **you've hit a bug** that should be filed with the package owner or person who gave you the binaries.

For applications run as non-root but a "non-daemon" user, the `~/.local/lib` directory is a place users can drop their own libraries and add to `LD_LIBRARY_PATH`. See also `~/.local/bin` and the XDG Base Directory Spec documentation.

For daemons running out of `systemd`, you can add another directory to the service's defined `LD_LIBRARY_PATH` as a drop-in.

If the application doesn't fit under these or within a container, I'd consider generating a `flatpak` to ensure any extra libraries are walled off from the rest of the system and that it can be trivially moved to a new box down the line. A "one time install" is never one time in my experience.

It is tempting to drop a config into `/etc/ld.so.conf.d`, but this pollutes the global linker space.

It is tempting to use `rpath` to just fix things, but this is invisible and will get forgotten.

Concluding Thoughts

What have we learned and how can we apply it?

Areas of Improvement

Existing self tests are pretty good, but **with your help** they could be great! See `%check` in your favorite package! Or ask about next steps on centos-devel. Functional testing workloads are also welcome! This is how we help **assure** the **quality** of the released packages.

A Koji plugin to run `abidiff` against the artifacts from the last build would produce an amazing record of changes!

Different output formats for libabigail would be wonderful! Annotated sources? JSON? Something else? Make this data more digestible and more usable!

So, Will my CentOS Stream application run on RHEL?

In practice the answer to this is a function of the RHEL ABI/API guarantee that we've been talking about and the mindset of the code developer.

Something built on CentOS Stream should run on RHEL if it was built with compatibility in mind.

If a binary has special requirements, there are a number of ways to work around the linker until the symbols are considered part of the consistency guarantee.

So, will my CentOS Stream application run on RHEL? I believe the answer is clearly "yes, this shouldn't be a problem".

Notice of Production

Work supported by the Fermi National Accelerator Laboratory, managed and operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

Questions?