# iCommands Userspace Packaging

Markus Kitsinger
github.com/SwooshyCueb
Software Engineer, iRODS Consortium

June 8-11, 2021
iRODS User Group Meeting 2021
Virtual Event

## What

- The iCommands you're already familiar with
- Packaged up for deployment in locked down environments
- Includes all needed libraries
- Works without a lot of fuss, just extract and run

## Why

It is common for shared systems (such as HPC environments) to be pretty locked down. Being able to use the iCommands in these environments without jumping through a bunch of hoops would be incredibly convenient, and is something that the community has asked for.

# Goals and Needs

- As portable as possible - extract and run with no fuss
  - iRODS runtime libraries bundled in
  - Dependency libraries bundled in
  - No need to set `LD_LIBRARY_PATH` or other environment variables
- We don't want to re-build the universe
  - Bundle our already built dependency libraries (`/opt/irods-externals`)
  - Plus any appropriate distro-provided dependencies
- As few extra build dependencies as possible

# Our Solution - Overview

- A Python 3 script (plus a few modules) invoked by a CMake build target
- Produces a tarball specific to the target distribution
  - for all currently supported distributions
- Minor surgery performed on the binaries
- Packed into a tarball named appropriately for the target platform

1. Tons of information is passed in from CMake.
2. The Python module search path is augmented to include the scripts directory from the iRODS source code.
3. Several tools that may be used during the packaging process are evaluated for availability and usability.
4. A workspace is prepared in the CMakeFiles directory.

**Other Details**

- New CMake install components for iCommands, manpages, and test data.
- New CMake build target that builds just the iCommands and does not create manpages.

# Our Solution - Step-by-step - Phase 1: Gathering iRODS components

1. The `cmake_install.cmake` script for the iCommands build is invoked directly with CMake variables set to install the iCommands to a temporary prefix.
2. The shell script iCommands are copied directly to the package bin directory.
3. The iCommands binaries are stripped and cleaned, and then copied into a staging area.
4. Using some information passed in from CMake, the iRODS client auth and network plugins are located, followed by the iRODS runtime libraries.
5. The plugins and libraries are stripped and cleaned, and then stored in the staging area.

# Our Solution - Step-by-step - Strip and clean process

- Three of the tools evaluated earlier were a standard `strip` tool, `ldd`, and the LIEF Python module.
  - If a `strip` tool is not found, this entire process is a no-op.
  - If a `strip` tool is found, but LIEF or `ldd` is not found (or not usable), the second and third steps of this process do not take place.

1. Unneeded symbols are stripped from the binaries using the `strip` tool.
2. `ldd` is used to determine which (if any) libraries that are imported by the binary are not used.
3. If any imported libraries are found to be unused, their entries are removed from the binary's dynamic header using LIEF.
   - This cuts down on the number of dependency libraries that are included in the final package.

# Our Solution - Step-by-step - Phase 2: Gathering external libraries

- There are two kinds of external libraries: iRODS-provided (`/opt/irods-externals`) and distro-provided.
- In order to help the packager identify which libraries should be included in the package, a set of directive files is included with the packager:
  - A file listing the names of iRODS-provided libraries known to potentially exist in the dependency tree.
  - A file for each supported distribution (plus one more common to all distributions) listing the full sonames of distro-provided libraries known to potentially exist in the dependency tree, along with a mark indicating whether the library is explicitly excluded from the package, or allowed to be included in the package.
- Should an external dependency be identified that is not listed in the directives, the packager will throw a warning and the library will be excluded from the package.

# Our Solution - Step-by-step - Phase 2: Gathering external libraries

1. `ldd` is used to get the locations of all dependency libraries of the gathered iRODS components.
   - This gives us the entire dependency tree, so indirect dependencies are included as well.
2. The direct dependencies of the prepared binaries are identified using a `readelf` tool (or LIEF, should one not be available).
   - Any already staged libraries are filtered out, and the results are condensed down to a list of unique sonames. This list is then filtered against the directives.
3. The results from step 1 are used to get the locations of the libraries from step 2. These libraries are stripped and cleaned, and then stored in the staging area.
4. Steps 2-4 are repeated for the newly staged libraries until no dependencies remain.

# Our Solution - Step-by-step - Phase 2: Gathering external libraries - Alternate

If LIEF was not found or is not usable, an alternate process is used:

1. `ldd` is used to get the locations of all dependency libraries of the gathered iRODS components.
    - This gives us the entire dependency tree, so indirect dependencies are included as well.
2. The list of dependency libraries is filtered against the directives.
3. The filtered libraries are stripped and cleaned, and then stored in the staging area.

# Our Solution - Step-by-step - Phase 3: Setting `RUNPATH`/`RPATH`

In order to eliminate the need to set the `LD_LIBRARY_PATH` environment variable, we need to set the `RUNPATH` or `RPATH` in as many of the binaries as possible.

1. For each staged binary, the relative path from their directory in the package to the lib directory in the package is derived.
   - If this path is derived to be `.` or empty, the packager attempts to set the binary's `RUNPATH`/`RPATH` to `$ORIGIN`*.
   - Otherwise, the packager attempts to set the binary's `RUNPATH`/`RPATH` to `$ORIGIN/`* + the derived relative path
2. The binaries are stored in their package directories.

* `$ORIGIN` is a substitution string in `RUNPATHs` and `RPATHs` for the directory in which the library or executable resides.

# Our Solution - Step-by-step - Phase 3: Setting `RUNPATH`/`RPATH` - The Tools

- The packager has three methods at its disposal for setting the `RUNPATH`/`RPATH` of a binary:
  - LIEF can set the `DF_ORIGIN` flag, which helps keep the runtime linker from misbehaving. It can also add a `RUNPATH` to a binary that does not arleady have an `RUNPATH` or `RPATH`. However, it can also cause the entire packager to segfault, and has the potential to bungle up the resultant binaries.
  - `chrpath` can change the `RPATH` dynamic entry to a `RUNPATH` dynamic entry.
  - CMake is guaranteed to be available.
- The packager will try each available method until one succeeds.
  - If all fail (or none are avialable), the binary is used as-is.
- `chrpath` is prioritized over CMake. Whether LIEF is given top or bottom priority depends on a number of variables.

# Our Solution - Step-by-step - Phase 4: Creating Final Package

- At this stage, everything that is to be included in the package has been placed in a package directory with the final directory structure that is to appear in the package. The only thing left to do is to actually make the tarball.
- In order to support the most amount of compression formats possible, the packager will look for a `tar` tool that automatically compresses based on output filename.
  - The package filename and output path can be specified using a CMake variable.
  - The default compression is gzip, as it is widely supported.
- A few extra flags are passed to the tar tool to ensure the tarball is suitable for a relocatable package.

# Our Solution - Other fun details

- It's multithreaded!
  - It uses the default number of threads for a python ThreadPoolExecutor, as getting the parallelization flag passed to the native buildsystem is a super complicated problem that is way outside the scope of the packager.
- It's handy!
  - The `strip_util` and `runpath_util` modules double as functional command line utilities that can be executed directly if needed.

**Other solutions we considered**

- Conda
- Build in container against Alpine Linux or Ubuntu Trusty

# Challenges

- LIEF
- `RUNPATH`/`RPATH` shenanigans and the `DF_ORIGIN` flag
- External dependencies
  - symlinks and linker script files
  - soname differences between Centos and Ubuntu
  - OpenSSL
  - Symbol versioning and GLIBC
  - Knowing what to bundle and what to exclude
- Invocation
- Bad path assumptions in iRODS runtime
- Python 3.5

# TODO

- Decrease reliability on LIEF.
- Prevent LIEF's segfaults from crashing the whole packager.
- Clean out unused library imports with symbol version references.
- Move logging instrumentation code into iRODS scripts folder.
- Use `objdump` as an alternative to `readelf`.
- Fix setting `RUNPATH`/`RPATH` with CMake on Centos.
- Use `readelf`/`objdump` to make stripping and `RUNPATH`/`RPATH` setting smarter.
- Implement support for musl's `ldd`.
- Make library directive syntax rolling-release-friendly.
- Linker script file handling.