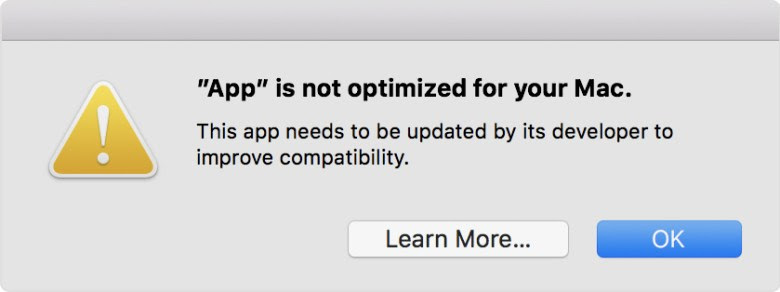
Apple has [run the death bell](https://support.apple.com/en-us/HT208436) on 32-bit macOS apps and, if you’re running a recent macOS version on your Mac (which you *should* so you can get security updates) you likely see this alert from time-to-time:

[](https://rud.is/b/2018/08/24/friday-rstats-twofer-finding-macos-32-bit-apps-processing-data-from-system-commands/macos-high-sierra-32-bit-app-alert/)

If you’re like me, you click through that and keep working but later ponder just how many of those apps you have. They are definitely going away, so knowing if your favourite app is on the chopping block is likely a good idea.

You can get this information via the “About This Mac”Í¢”System Report” app and sorting via one of the table fields.

R folks are data folks and we know we can do better than that. But, first we need to get the data. Thankfully, we can get this via the system\_profiler command-line utility since it can both display user-friendly information in the terminal and also generate an XML version of the information to work with. We won’t need to head to the terminal for this work, though, since there are many ways to execute the command from R and read the generated output.

System Profiler

system\_profiler

Report system hardware and software configuration.

Syntax

system\_profiler [-usage]

system\_profiler [-listDataTypes]

system\_profiler [-xml] *dataType1* ... *dataTypeN*

system\_profiler [-xml] [-detailLevel *level*]

Key:

-xml Generate a report in XML format. If the XML report

is redirected to a file with a ".spx" suffix that

file can be opened with System Profiler.app.

-listDataTypes List the available datatypes.

-detailLevel *level* The level of detail for the report:

mini report with no personal information

basic basic hardware and network information

full all available information

-usage Print usage info and examples.

system\_profiler is a replacement for /usr/sbin/AppleSystemProfiler.

**Examples**

Generate a text report with the standard detail level.  
$ system\_profiler

Generate a report of all 32 bit software and save to a text file on the desktop:  
$ system\_profiler SPApplicationsDataType | grep -B 6 -A 2 "(Intel): No" > ~/Desktop/non64bit.txt

Generate a short report containing no personal information.  
$ system\_profiler -detailLevel mini

Show a list of the available data types.  
$ system\_profiler -listDataTypes

Generate a text report containing only software and network data.  
$ system\_profiler SPSoftwareDataType SPNetworkDataType

Create an XML file which can be opened by System Profiler.app  
$ system\_profiler -xml > MyReport.spx

**Executing System Calls from R**

Base R provides two core methods for issuing a system call:

* system()
* system2()

Note that there are other functions provided with a base R installation that can also issue system commands and process the “piped” output, but we’ll focus on these deliberate invocation ones.

Functions in two other packages can also assist with this task and we’ll include a look at:

* processx::run()
* sys::exec\_internal()

as well.

Why leave base R for this task? Truthfully, we really don’t *need* to, but both sys and processx have other tasks which do make them handy tools in your package toolbox. Having said that, keep reading since we’re going to end up *not* choosing the built-in functions for this task.

This is the command line we need to execute:

system\_profiler -xml -detailLevel full SPApplicationsDataType

Let’s load up all the packages we’ll be needing and execute this command-line all four ways, then briefly discuss the differences:

library(sys)

library(processx)

library(microbenchmark)

library(xml2)

library(tidyverse)

system(

command = "system\_profiler -xml -detailLevel full SPApplicationsDataType",

intern = TRUE

) -> apps\_system

str(apps\_system)

## chr [1:10665] "" ...

system2(

command = "system\_profiler",

args = c("-xml", "-detailLevel", "full", "SPApplicationsDataType"),

stdout = TRUE

) -> apps\_system2

str(apps\_system2)

## chr [1:10665] "" ...

processx::run(

command = "system\_profiler",

args = c("-xml", "-detailLevel", "full", "SPApplicationsDataType"),

spinner = TRUE

) -> apps\_processx\_run

str(apps\_processx\_run)

## List of 4

## $ status : int 0

## $ stdout : chr "XML STRING THAT prism.js won't let me show"

## $ stderr : chr ""

## $ timeout: logi FALSE

sys::exec\_internal(

cmd = "system\_profiler",

args = c("-xml", "-detailLevel", "full", "SPApplicationsDataType")

) -> apps\_sys\_exec\_internal

str(apps\_sys\_exec\_internal)

## List of 3

## $ status: int 0

## $ stdout: raw [1:331133] 3c 3f 78 6d ...

## $ stderr: raw(0)

The core difference between system() and the rest is that *you* need to shQuote() for system() whereas that’s taken care of for you by the others (so they’re a bit safer by default since you’re more than likely going to forget to shQuote()).

You can definitely notice the main differences in return objects. The built-in functions just give us the character data from the standard output stream (stdout) and the last two return a more structured object that provides more explicit information about the job we just executed. The base ones can provide this detail, but it’s a twisty maze of remembering which options do what vs the more (IMO) straightforward approach both processx and sys take.

You’ll also notice a difference in stdout between processx and sys with the latter giving us a raw vector vs a character vector. This gives us a great deal of power and flexibility. It also turns out to be a great choice for processing command-line-generated XML data. Here’s why:

microbenchmark(

sys = xml2::read\_xml(apps\_sys\_exec\_internal$stdout),

processx = xml2::read\_xml(apps\_processx\_run$stdout)

)

## Unit: milliseconds

## expr min lq mean median uq max neval

## sys 4.086492 4.60078 9.085143 5.508814 5.906942 207.6495 100

## processx 9.510356 10.98282 14.275499 12.054810 13.292234 163.9870 100

It turns out xml2::read\_xml() makes much quicker work of the raw vector data (though, I mean, *really*—are we really going to care about those ~5ms IRL?).

We’ll move on to the real reason for the post, but definitely explore both sys and processx since they are both super-handy packages.

***“Can we please just find the 32-bit apps already?”***

No problem. Well, actually, there is a minor annoyance. These are [property list XML files](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/PropertyLists/UnderstandXMLPlist/UnderstandXMLPlist.html) and I’ll confess that I truly hate this format. There are “dictionary arrays” of key and value nodes, but those nodes are siblings vs directly associated pairs. So, we have to use the sibling relationship to work with them. It’s not *hard*, per se, just (again, IMO) suboptimal.

Let’s take a look at it:

apps <- read\_xml(apps\_sys\_exec\_internal$stdout)

xml\_find\_all(apps, "//array/dict")

## {xml\_nodeset (476)}

## [1] \n \_SPCommandLineArguments\n \n ...

## [2] \n \_name\n Sublime Text\n h ...

## [3] \n \_name\n System Preferences\n ...

## [4] \n \_name\n Google Chrome Canary\n ...

## [5] \n \_name\n Google Chrome\n ...

## [6] \n \_name\n Dropbox\n has64B ...

## [7] \n \_name\n Keypad\n has64Bi ...

## [8] \n \_name\n Garmin WebUpdater\n < ...

## [9] \n \_name\n LaTeXiT\n has64B ...

## [10] \n \_name\n CocoaPacketAnalyzer\n ...

## [11] \n \_name\n Janetter\n has64 ...

## [12] \n \_name\n VMware Fusion\n ...

## [13] \n \_name\n Photo Library Migration Utility ...

## [14] \n \_name\n Setup Assistant\n \n \_name\n Siri\n has64BitI ...

## [16] \n \_name\n Software Update\n \n \_name\n Spotlight\n has6 ...

## [18] \n \_name\n Stocks\n has64Bi ...

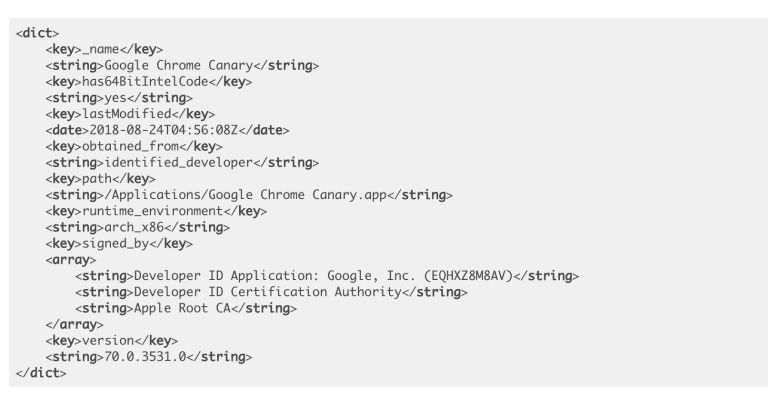
## [19] \n \_name\n SystemUIServer\n \n \_name\n UniversalAccessControl ...

## ...

Let’s look at a sample record using xml\_view() from the htmltidy package:

xml\_find\_all(apps, "//array/dict[key='\_name']")[1] %>%

htmltidy::xml\_view()

[](https://rud.is/b/2018/08/24/friday-rstats-twofer-finding-macos-32-bit-apps-processing-data-from-system-commands/screen-shot-2018-08-24-at-4-58-41-am/)

Be wary of using xml\_view() on giant XML structures since it’ll freeze up RStudio for a bit and even slows down Chrome since the resultant, composed DOM object can get ginormous.

Now we know we can use has64BitIntelCode for filtering once we get to the data. Let’s read in all the apps, cherry-picking the fields and then just look at the 32-bit apps:

xml\_find\_all(apps, "//array/dict[key='\_name']") %>%

map\_df(~{

list(

name = xml\_find\_first(.x, ".//string") %>% xml\_text(),

path = xml\_find\_first(.x, ".//key[.='path']/following-sibling::string") %>% xml\_text(),

is\_64bit = xml\_find\_first(.x, ".//key[.='has64BitIntelCode']/following-sibling::string") %>% xml\_text()

)

}) %>%

filter(is\_64bit == "no") %>%

arrange(name) %>%

select(-is\_64bit)

## # A tibble: 30 x 2

## name path

##

## 1 AAM Registration Notifier /Applications/Utilities/Adobe Application Manaâ€¦

## 2 AAM Registration Notifier /Applications/Utilities/Adobe Application Manaâ€¦

## 3 AAM Updates Notifier /Applications/Utilities/Adobe Application Manaâ€¦

## 4 AAMLauncherUtil /Applications/Utilities/Adobe Application Manaâ€¦

## 5 ACR\_9\_10 /Library/Application Support/Adobe/Uninstall/Aâ€¦

## 6 Adobe Application Manager /Applications/Utilities/Adobe Application Manaâ€¦

## 7 adobe\_licutil /Applications/Utilities/Adobe Application Manaâ€¦

## 8 Audacity /Applications/Audacity.app

## 9 COCM\_1\_0\_32 /Library/Application Support/Adobe/Uninstall/Câ€¦

## 10 COPS\_1\_0\_32 /Library/Application Support/Adobe/Uninstall/Câ€¦

## # ... with 20 more rows

The Adobe helper apps are longstanding 32-bit “offenders”. Many of these death-row apps fall into the “helper” category and will hopefully get some attention by their developers. I do find it amusing that Apple *kinda* wants **us** to prod the developers to get their collective acts together.

**FIN**

This exhaustive search finds all of the 32-bit apps residing on your system. If you just want to see the one’s you’ve *executed* and macOS has kept track of, you can drop to a command-line and do:

sudo Rscript -e 'knitr::kable((dplyr::select(dplyr::tbl(dplyr::src\_sqlite("/var/db/SystemPolicyConfiguration/ExecPolicy"), "legacy\_exec\_history\_v3"), responsible\_path)))'

You need elevated privileges to access those files, so please *read that whole line* to make sure I’m not having you rm -rf /.

Remember to take some time to explore the sys and processx packages and perhaps bundle up the salient bits of this post into a script so you can occasionally check to see the 32-bit eradication progress.