In the “Changes on CRAN” section of the latest version of the The R Journal (Vol. 10/2, December 2018) had this short blurb entitled “CRAN mirror security”:

Currently, there are 100 official CRAN mirrors, 68 of which provide both secure downloads via ‘https’ and use secure mirroring from the CRAN master (via rsync through ssh tunnels). Since the R 3.4.0 release, chooseCRANmirror() offers these mirrors in preference to the others which are not fully secured (yet).

I would have linked to the R Journal section quoted above but *I can’t* because I’m blocked from accessing all resources at the IP address serving cran.r-project.org from my business-class internet connection likely due to me having a personal CRAN mirror (that was following the rules, which I also cannot link to since I can’t get to the site).

That word — “security” — is one of *the* most misunderstood and misused terms in modern times in many contexts. The context for the use here is cybersecurity and since CRAN (and others in the R community) seem to equate transport-layer uber-obfuscation with actual security/safety I thought it would be useful for R users in general to get a more complete picture of these so-called “secure” hosts. I also did this since I had to figure out another way to continue to have a CRAN mirror and needed to validate which nodes both supported + allowed mirroring and were at least somewhat trustworthy.

Unless there is something truly egregious in a given section I’m just going to present data with some commentary (I’m unamused abt being blocked so some commentary has an unusually sharp edge) and refrain from stating “X is |” since the goal is really to help **you** make the best decision of which mirror to use on your own.

The full Rproj supporting the snippets in this post (and including the data gathered by the post) can be found in my new [R blog projects](https://git.rud.is/r-blog-projects/cran-mirror-security).

We’re going to need a few supporting packages so let’s get those out of the way:

library(xml2)

library(httr)

library(curl)

library(stringi)

library(urltools)

library(ipinfo) # install.packages("ipinfo", repos = "https://cinc.rud.is/")

library(openssl)

library(furrr)

library(vershist) # install.packages("vershist", repos = "https://cinc.rud.is/")

library(ggalt)

library(ggbeeswarm)

library(hrbrthemes)

library(tidyverse)

**What Is “Secure”?**

As noted, CRAN folks seem to think encryption == security since the criteria for making that claim in the R Journal was transport-layer encryption for rsync (via ssh) mirroring from CRAN to a downstream mirror and a downstream mirror providing an https transport for shuffling package binaries and sources from said mirror to your local system(s). I find that equally as adorable as I do the rhetoric from the Let’s Encrypt cabal as this https gets you:

* *in theory* protection from person-in-the-middle attacks that could otherwise fiddle with the package bits in transport
* protection from your organization or ISP knowing what specific package you were grabbing; note that **unless you’ve got a setup where your DNS requests are also encrypted** the entity that controls your transport layer *does indeed know exactly where you’re going*.

and…that’s about it.

The soon-to-be-gone-and-formerly-green-in-most-browsers lock icon alone tells you *nothing* about the configuration of any site you’re connecting to and using rsync over ssh provides no assurance as to what else is on the CRAN mirror server(s), what else is using the mirror server(s), how many admins/users have shell access to those system(s) nor anything else about the cyber hygiene of those systems.

So, we’re going to look at (not necessarily in this order & non-exhaustively since this isn’t a penetration test and only lightweight introspection has been performed):

* how many servers are involved in a given mirror URL
* SSL certificate information including issuer, strength, and just how many other domains can use the cert
* the actual server SSL transport configuration to see just how many CRAN mirrors have HIGH or CRITICAL SSL configuration issues
* use (or lack thereof) HTTP “security” headers (I mean, the server is supposed to be “secure”, right?)
* how much other “junk” is running on a given CRAN mirror (the more running services the greater the attack surface)

We’ll use R for *most* of this, too (I’m likely never going to rewrite longstanding SSL testers in/for R).

Let’s dig in.

**Acquiring Most of the Metadata**

It can take a little while to run some of the data gathering steps so the project repo includes the already-gathered data. But, we’ll show the work on the first bit of reconnaissance which involves:

* Slurping the SSL certificate from the first server in each CRAN mirror entry (again, I can’t link to the mirror page because I literally can’t see CRAN or the main R site anymore)
* Performing an HTTP HEAD request (to minimize server bandwidth & CPU usage) of the full CRAN mirror URL (we have to since load balancers or proxies could re-route us to a completely different server otherwise)
* Getting an IP address for each CRAN mirror
* Getting metadata about that IP address

This all done below:

if (!file.exists(here::here("data/mir-dat.rds"))) {

mdoc <- xml2::read\_xml(here::here("data/mirrors.html"), as\_html = TRUE)

xml\_find\_all(mdoc, ".//td/a[contains(@href, 'https')]") %>%

xml\_attr("href") %>%

unique() -> ssl\_mirrors

plan(multiprocess)

# safety first

dl\_cert <- possibly(openssl::download\_ssl\_cert, NULL)

HEAD\_ <- possibly(httr::HEAD, NULL)

dig <- possibly(curl::nslookup, NULL)

query\_ip\_ <- possibly(ipinfo::query\_ip, NULL)

ssl\_mirrors %>%

future\_map(~{

host <- domain(.x)

ip <- dig(host, TRUE)

ip\_info <- if (length(ip)) query\_ip\_(ip) else NULL

list(

host = host,

cert = dl\_cert(host),

head = HEAD\_(.x),

ip = ip,

ip\_info = ip\_info

)

}) -> mir\_dat

saveRDS(mir\_dat, here::here("data/mir-dat.rds"))

} else {

mir\_dat <- readRDS(here::here("data/mir-dat.rds"))

}

# take a look

str(mir\_dat[1], 3)

## List of 1

## $ :List of 5

## ..$ host : chr "cloud.r-project.org"

## ..$ cert :List of 4

## .. ..$ :List of 8

## .. ..$ :List of 8

## .. ..$ :List of 8

## .. ..$ :List of 8

## ..$ head :List of 10

## .. ..$ url : chr "https://cloud.r-project.org/"

## .. ..$ status\_code: int 200

## .. ..$ headers :List of 13

## .. .. ..- attr(\*, "class")= chr [1:2] "insensitive" "list"

## .. ..$ all\_headers:List of 1

## .. ..$ cookies :'data.frame': 0 obs. of 7 variables:

## .. ..$ content : raw(0)

## .. ..$ date : POSIXct[1:1], format: "2018-11-29 09:41:27"

## .. ..$ times : Named num [1:6] 0 0.0507 0.0512 0.0666 0.0796 ...

## .. .. ..- attr(\*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...

## .. ..$ request :List of 7

## .. .. ..- attr(\*, "class")= chr "request"

## .. ..$ handle :Class 'curl\_handle'

## .. ..- attr(\*, "class")= chr "response"

## ..$ ip : chr "52.85.89.62"

## ..$ ip\_info:List of 8

## .. ..$ ip : chr "52.85.89.62"

## .. ..$ hostname: chr "server-52-85-89-62.jfk6.r.cloudfront.net"

## .. ..$ city : chr "Seattle"

## .. ..$ region : chr "Washington"

## .. ..$ country : chr "US"

## .. ..$ loc : chr "47.6348,-122.3450"

## .. ..$ postal : chr "98109"

## .. ..$ org : chr "AS16509 Amazon.com, Inc."

Note that two sites failed to respond so they were excluded from all analyses.

**A Gratuitous Map of “Secure” CRAN Servers**

Since [ipinfo.io](https://ipinfo.io/)‘s API returns lat/lng geolocation information why not start with a map (since that’s going to be the kindest section of this post):

maps::map("world", ".", exact = FALSE, plot = FALSE, fill = TRUE) %>%

fortify() %>%

filter(region != "Antarctica") -> world

map\_chr(mir\_dat, ~.x$ip\_info$loc) %>%

stri\_split\_fixed(pattern = ",", n = 2, simplify = TRUE) %>%

as.data.frame(stringsAsFactors = FALSE) %>%

as\_tibble() %>%

mutate\_all(list(as.numeric)) -> wheres\_cran

ggplot() +

ggalt::geom\_cartogram(

data = world, map = world, aes(long, lat, map\_id=region),

color = ft\_cols$gray, size = 0.125

) +

geom\_point(

data = wheres\_cran, aes(V2, V1), size = 2,

color = ft\_cols$slate, fill = alpha(ft\_cols$yellow, 3/4), shape = 21

) +

ggalt::coord\_proj("+proj=wintri") +

labs(

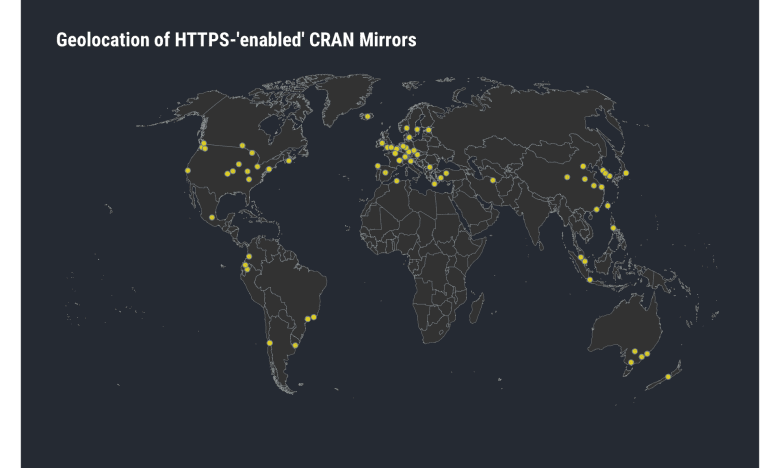
x = NULL, y = NULL,

title = "Geolocation of HTTPS-'enabled' CRAN Mirrors"

) +

theme\_ft\_rc(grid="") +

theme(axis.text = element\_blank())

[](https://i0.wp.com/rud.is/b/wp-content/uploads/2019/03/cran-map-1.png?ssl=1)

**Shakesperian Security**

*What’s in a [Subject Alternative] name? That which we call a site secure. By using dozens of other names would smell as not really secure at all?* —Hackmeyo & Pwndmeyet (II, ii, 1-2)

The average internet user likely has no idea that one SSL certificate can front a gazillion sites. I’m not just talking a wildcard cert (e.g. using \*.rud.is for all rud.is subdomains which I try not to do for many reasons), I’m talking dozens of [subject alternative names](https://tools.ietf.org/html/rfc5280#section-4.2.1.6). Let’s examine some data since an example is better than blathering:

# extract some of the gathered metadata into a data frame

map\_df(mir\_dat, ~{

tibble(

host = .x$host,

s\_issuer = .x$cert[[1]]$issuer %||% NA\_character\_,

i\_issuer = .x$cert[[2]]$issuer %||% NA\_character\_,

algo = .x$cert[[1]]$algorithm %||% NA\_character\_,

names = .x$cert[[1]]$alt\_names %||% NA\_character\_,

nm\_ct = length(.x$cert[[1]]$alt\_names),

key\_size = .x$cert[[1]]$pubkey$size %||% NA\_integer\_

)

}) -> certs

certs <- filter(certs, complete.cases(certs))

count(certs, host, sort=TRUE) %>%

ggplot() +

geom\_quasirandom(

aes("", n), size = 2,

color = ft\_cols$slate, fill = alpha(ft\_cols$yellow, 3/4), shape = 21

) +

scale\_y\_comma() +

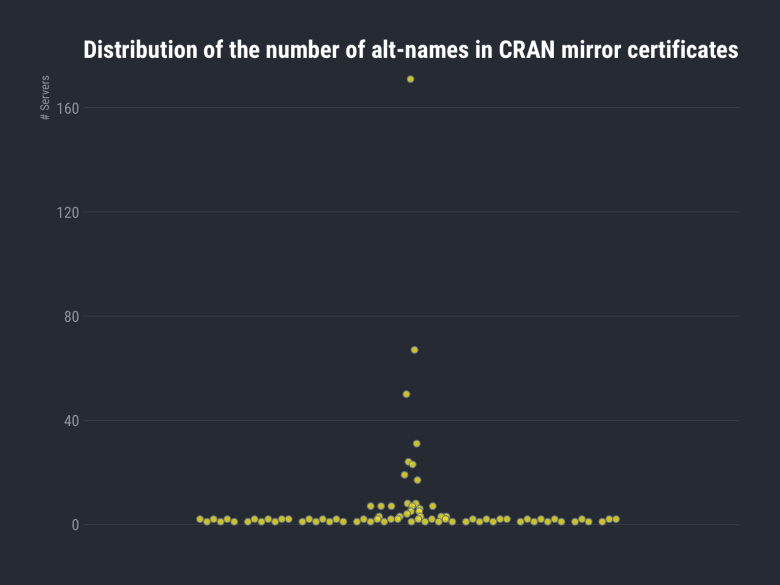
labs(

x = NULL, y = "# Servers",

title = "Distribution of the number of alt-names in CRAN mirror certificates"

) +

theme\_ft\_rc(grid="Y")

[](https://i0.wp.com/rud.is/b/wp-content/uploads/2019/03/alt-names-ct-1.png?ssl=1)

*Most* only front a couple but there are some with a *crazy* amount of domains. We can look at a slice of cran.cnr.berkeley.edu:

filter(certs, host == "cran.cnr.berkeley.edu") %>%

select(names) %>%

head(20)

| **names** |
| --- |
| nature.berkeley.edu |
| ag-labor.cnr.berkeley.edu |
| agro-laboral.cnr.berkeley.edu |
| agroecology.berkeley.edu |
| anthoff.erg.berkeley.edu |
| are-dev.cnr.berkeley.edu |
| are-prod.cnr.berkeley.edu |
| are-qa.cnr.berkeley.edu |
| are.berkeley.edu |
| arebeta.berkeley.edu |
| areweb.berkeley.edu |
| atkins-dev.cnr.berkeley.edu |
| atkins-prod.cnr.berkeley.edu |
| atkins-qa.cnr.berkeley.edu |
| atkins.berkeley.edu |
| bakerlab-dev.cnr.berkeley.edu |
| bakerlab-prod.cnr.berkeley.edu |
| bakerlab-qa.cnr.berkeley.edu |
| bamg.cnr.berkeley.edu |
| beahrselp-dev.cnr.berkeley.edu |

The project repo has some more examples and you can examine as many as you like.

For some CRAN mirrors the certificate is used all over the place at the hosting organization. That alone isn’t bad, but organizations are generally terrible at protecting the secrets associated with certificate generation (just look at how many Google/Apple app store apps are found monthly to be using absconded-with enterprise certs) and since each server with these uber-certs has copies of public & private bits users had better hope that mal-intentioned ne’er-do-wells do not get copies of them (making it easier to impersonate any one of those, especially if an attacker controls DNS).

This Berkeley uber-cert is also kinda cute since it mixes alt-names for dev, prod & qa systems across may different apps/projects (dev systems are notoriously maintained improperly in virtually every organization).

There *are* legitimate reasons and circumstances for wildcard certs and taking advantage of SANs. You can examine what other CRAN mirrors do and judge for yourself which ones are Doing It Kinda OK.

**Size (and Algorithm) Matters**

In some crazy twist of pleasant surprises most of the mirrors seem to do OK when it comes to the algorithm and key size used for the certificate(s):

distinct(certs, host, algo, key\_size) %>%

count(algo, key\_size, sort=TRUE)

| **algo** | **key\_size** | **n** |
| --- | --- | --- |
| sha256WithRSAEncryption | 2048 | 59 |
| sha256WithRSAEncryption | 4096 | 13 |
| ecdsa-with-SHA256 | 256 | 2 |
| sha256WithRSAEncryption | 256 | 1 |
| sha256WithRSAEncryption | 384 | 1 |
| sha512WithRSAEncryption | 2048 | 1 |
| sha512WithRSAEncryption | 4096 | 1 |

You can go to the mirror list and hit up [SSL Labs Interactive Server Test](https://www.ssllabs.com/ssltest/) (which has links to many ‘splainers) or use the [ssllabs[](https://git.rud.is/hrbrmstr/ssllabs)](https://git.rud.is/hrbrmstr/ssllabs) R package to get the grade of each site. I dig into the state of config and transport issues below but will suggest that you stick with sites with ecdsa certs or sha256 and higher numbers if you want a general, quick bit of guidance.

**Where Do They Get All These ~~Wonderful~~ Certs?**

Certs come from somewhere. You can self-generate play ones, setup your own internal/legit certificate authority and augment trust chains, or go to a bona-fide certificate authority to get a certificate.

Your browsers and operating systems have a built-in set of certificate authorities they trust and you can use [ssllabs::get\_root\_certs()[](https://git.rud.is/hrbrmstr/ssllabs/src/branch/master/R/root.r)](https://git.rud.is/hrbrmstr/ssllabs/src/branch/master/R/root.r) to see an up-to-date list of ones for Mozilla, Apple, Android, Java & Windows. In the age of Let’s Encrypt, certificates have almost no monetary value and virtually no integrity value so where they come from isn’t *as* important as it used to be, but it’s kinda fun to poke at it anyway:

distinct(certs, host, i\_issuer) %>%

count(i\_issuer, sort = TRUE) %>%

head(28)

| **i\_issuer** | **n** |
| --- | --- |
| CN=DST Root CA X3,O=Digital Signature Trust Co. | 20 |
| CN=COMODO RSA Certification Authority,O=COMODO CA Limited,L=Salford,ST=Greater Manchester,C=GB | 7 |
| CN=DigiCert Assured ID Root CA,OU=www.digicert.com,O=DigiCert Inc,C=US | 7 |
| CN=DigiCert Global Root CA,OU=www.digicert.com,O=DigiCert Inc,C=US | 6 |
| CN=DigiCert High Assurance EV Root CA,OU=www.digicert.com,O=DigiCert Inc,C=US | 6 |
| CN=QuoVadis Root CA 2 G3,O=QuoVadis Limited,C=BM | 5 |
| CN=USERTrust RSA Certification Authority,O=The USERTRUST Network,L=Jersey City,ST=New Jersey,C=US | 5 |
| CN=GlobalSign Root CA,OU=Root CA,O=GlobalSign nv-sa,C=BE | 4 |
| CN=Trusted Root CA SHA256 G2,O=GlobalSign nv-sa,OU=Trusted Root,C=BE | 3 |
| CN=COMODO ECC Certification Authority,O=COMODO CA Limited,L=Salford,ST=Greater Manchester,C=GB | 2 |
| CN=DFN-Verein PCA Global – G01,OU=DFN-PKI,O=DFN-Verein,C=DE | 2 |
| OU=Security Communication RootCA2,O=SECOM Trust Systems CO.\,LTD.,C=JP | 2 |
| CN=AddTrust External CA Root,OU=AddTrust External TTP Network,O=AddTrust AB,C=SE | 1 |
| CN=Amazon Root CA 1,O=Amazon,C=US | 1 |
| CN=Baltimore CyberTrust Root,OU=CyberTrust,O=Baltimore,C=IE | 1 |
| CN=Certum Trusted Network CA,OU=Certum Certification Authority,O=Unizeto Technologies S.A.,C=PL | 1 |
| CN=DFN-Verein Certification Authority 2,OU=DFN-PKI,O=Verein zur Foerderung eines Deutschen Forschungsnetzes e. V.,C=DE | 1 |
| CN=Go Daddy Root Certificate Authority – G2,O=GoDaddy.com\, Inc.,L=Scottsdale,ST=Arizona,C=US | 1 |
| CN=InCommon RSA Server CA,OU=InCommon,O=Internet2,L=Ann Arbor,ST=MI,C=US | 1 |
| CN=QuoVadis Root CA 2,O=QuoVadis Limited,C=BM | 1 |
| CN=QuoVadis Root Certification Authority,OU=Root Certification Authority,O=QuoVadis Limited,C=BM | 1 |

That first one is Let’s Encrypt, which is not unexpected since they’re free and super easy to setup/maintain (especially for phishing campaigns).

A “fun” exercise might be to Google/DDG around for historical compromises tied to these CAs (look in the subject ones too if you’re playing with the data at home) and see what, eh, *issues* they’ve had.

You might want to keep more of an eye on this whole “boring” CA bit, too, since some trust stores are [noodling on the idea of trusting surveillance firms](https://www.zdnet.com/article/surveillance-firm-asks-mozilla-to-be-included-in-firefoxs-certificate-whitelist/) and you never know what Microsoft or Google is going to do to placate authoritarian regimes and allow into their trust stores.

At this point in the exercise you’ve got

* how many domains a certificate fronts
* certificate strength
* certificate birthplace

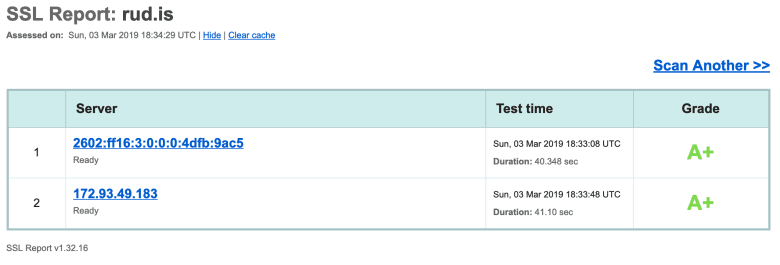
to use when formulating your own decision on what CRAN mirror to use.

But, as noted, certificate breeding is not enough. Let’s dive into the next areas.

**It’s In The Way That You Use It**

You can’t just look at a cert to evaluate site security. Sure, you can spend 4 days and use the aforementioned ssllabs package to get the rating for each cert (well, if they’ve been cached then an API call won’t be an assessment so you can prime the cache with 4 other ppl in one day and then everyone else can use the cached values and not burn the rate limit) or go one-by-one in the SSL Labs test site, but we can also use a tool like [testssl.sh[](https://github.com/drwetter/testssl.sh)](https://github.com/drwetter/testssl.sh) to gather technical data via interactive protocol examination.

I’m being a bit harsh in this post, so fair’s fair and here are [the plaintext results from my own run of testssl.sh for rud.is](https://rud.is/dl/rud.is-testssl.sh.txt) along with [ones from Qualys](https://www.ssllabs.com/ssltest/analyze.html?d=rud.is&latest):

[](https://i2.wp.com/rud.is/b/wp-content/uploads/2019/03/rud.is-ssllabs.png?ssl=1)

As you can see in the detail pages, I am having an issue with the provider of my .is domain (severe limitation on DNS record counts and types) so I fail CAA checks because I literally can’t add an entry for it nor can I use a different nameserver. Feel encouraged to pick nits about that tho as that should provide sufficient impetus to take two weeks of IRL time and some USD to actually get it transferred (yay. international. domain. providers.)

The project repo has all the results from a weekend run on the CRAN mirrors. No special options were chosen for the runs.

list.files(here::here("data/ssl"), "json$", full.names = TRUE) %>%

map\_df(jsonlite::fromJSON) %>%

as\_tibble() -> ssl\_tests

# filter only fields we want to show and get them in order

sev <- c("OK", "LOW", "MEDIUM", "HIGH", "WARN", "CRITICAL")

group\_by(ip) %>%

count(severity) %>%

ungroup() %>%

complete(ip = unique(ip), severity = sev) %>%

mutate(severity = factor(severity, levels = sev)) %>% # order left->right by severity

arrange(ip) %>%

mutate(ip = factor(ip, levels = rev(unique(ip)))) %>% # order alpha by mirror name so it's easier to ref

ggplot(aes(severity, ip, fill=n)) +

geom\_tile(color = "#b2b2b2", size = 0.125) +

scale\_x\_discrete(name = NULL, expand = c(0,0.1), position = "top") +

scale\_y\_discrete(name = NULL, expand = c(0,0)) +

viridis::scale\_fill\_viridis(

name = "# Tests", option = "cividis", na.value = ft\_cols$gray

) +

labs(

title = "CRAN Mirror SSL Test Summary Findings by Severity"

) +

theme\_ft\_rc(grid="") +

theme(axis.text.y = element\_text(size = 8, family = "mono")) -> gg

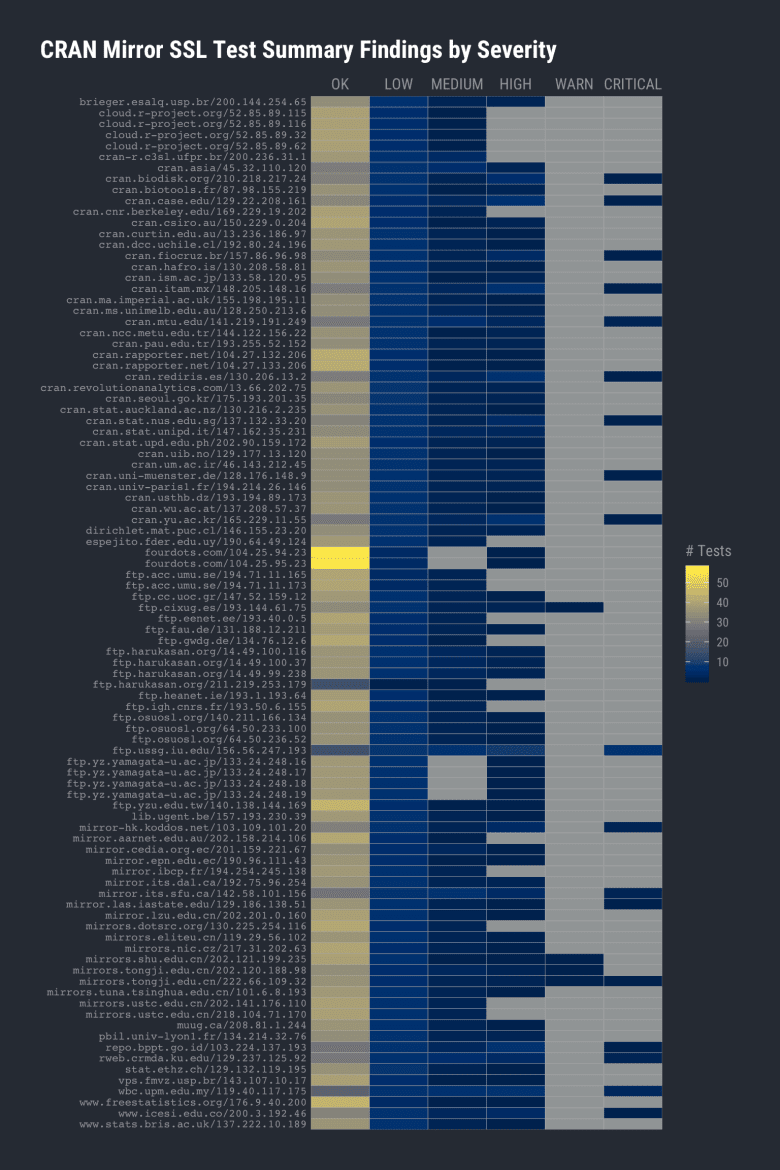
# We're going to move the title vs have too wide of a plot

gb <- ggplot2::ggplotGrob(gg)

gb$layout$l[gb$layout$name %in% "title"] <- 2

grid::grid.newpage()

grid::grid.draw(gb)

[](https://i1.wp.com/rud.is/b/wp-content/uploads/2019/03/testssl-1.png?ssl=1)

Thankfully *most* SSL checks come back OK. Unfortunately, many do not:

filter(ssl\_tests,severity == "HIGH") %>%

count(id, sort = TRUE)

| **id** | **n** |
| --- | --- |
| BREACH | 42 |
| cipherlist\_3DES\_IDEA | 37 |
| cipher\_order | 34 |
| RC4 | 16 |
| cipher\_negotiated | 10 |
| LOGJAM-common\_primes | 9 |
| POODLE\_SSL | 6 |
| SSLv3 | 6 |
| cert\_expiration\_status | 1 |
| cert\_notAfter | 1 |
| fallback\_SCSV | 1 |
| LOGJAM | 1 |
| secure\_client\_renego | 1 |

filter(ssl\_tests,severity == "CRITICAL") %>%

count(id, sort = TRUE)

| **id** | **n** |
| --- | --- |
| cipherlist\_LOW | 16 |
| TLS1\_1 | 5 |
| CCS | 2 |
| cert\_chain\_of\_trust | 1 |
| cipherlist\_aNULL | 1 |
| cipherlist\_EXPORT | 1 |
| DROWN | 1 |
| FREAK | 1 |
| ROBOT | 1 |
| SSLv2 | 1 |

Some CRAN mirror site admins aren’t keeping up with secure SSL configurations. If you’re not familiar with some of the acronyms here are a few (fairly layman-friendly) links:

* [BREACH](https://en.wikipedia.org/wiki/BREACH)
* [DROWN](https://en.wikipedia.org/wiki/DROWN_attack)
* [FREAK](https://en.wikipedia.org/wiki/FREAK)
* [LOGJAM](https://en.wikipedia.org/wiki/Logjam_(computer_security))– [POODLE](https://en.wikipedia.org/wiki/POODLE)
* [ROBOT](https://robotattack.org/)

You’d be hard-pressed to have me say that the presence of these is the end of the world (I mean, you’re trusting random servers to provide packages for you which may run in secure enclaves on production code, so how important can this really be?) but I also wouldn’t attach the word “secure” to any CRAN mirror with HIGH or CRITICAL SSL configuration weaknesses.

**Getting Ahead[er] Of Myself**

We did the httr::HEAD() request primarily to capture HTTP headers. And, we *definitely* got some!

map\_df(mir\_dat, ~{

if (length(.x$head$headers) == 0) return(NULL)

host <- .x$host

flatten\_df(.x$head$headers) %>%

gather(name, value) %>%

mutate(host = host)

}) -> hdrs

count(hdrs, name, sort=TRUE) %>%

head(nrow(.))

| **name** | **n** |
| --- | --- |
| content-type | 79 |
| date | 79 |
| server | 79 |
| last-modified | 72 |
| content-length | 67 |
| accept-ranges | 65 |
| etag | 65 |
| content-encoding | 38 |
| connection | 28 |
| vary | 28 |
| strict-transport-security | 13 |
| x-frame-options | 8 |
| x-content-type-options | 7 |
| cache-control | 4 |
| expires | 3 |
| x-xss-protection | 3 |
| cf-ray | 2 |
| expect-ct | 2 |
| set-cookie | 2 |
| via | 2 |
| ms-author-via | 1 |
| pragma | 1 |
| referrer-policy | 1 |
| upgrade | 1 |
| x-amz-cf-id | 1 |
| x-cache | 1 |
| x-permitted-cross-domain | 1 |
| x-powered-by | 1 |
| x-robots-tag | 1 |
| x-tuna-mirror-id | 1 |
| x-ua-compatible | 1 |

There are a handful of [“security” headers](https://securityheaders.com/) that kinda matter so we’ll see how many “secure” CRAN mirrors use “security” headers:

c(

"content-security-policy", "x-frame-options", "x-xss-protection",

"x-content-type-options", "strict-transport-security", "referrer-policy"

) -> secure\_headers

count(hdrs, name, sort=TRUE) %>%

filter(name %in% secure\_headers)

| **name** | **n** |
| --- | --- |
| strict-transport-security | 13 |
| x-frame-options | 8 |
| x-content-type-options | 7 |
| x-xss-protection | 3 |
| referrer-policy | 1 |

I’m honestly shocked any were in use but only a handful or two are using even one “security” header. cran.csiro.au uses all five of the above so good on ya Commonwealth Scientific and Industrial Research Organisation!

I keep putting the word “security” in quotes as R does nothing with these headers when you do an install.packages(). As a whole they’re important but mostly when it comes to your safety when browsing those CRAN mirrors.

I would have liked to have seen at *least* one with some Content-Security-Policy header, but a girl can at least dream.

**Version Aversion**

There’s another HTTP response header we can look at, the Server one which is generally there to help attackers figure out whether they should target you further for HTTP server and application attacks. No, I mean it! Back in the day when geeks rules the internets — and it wasn’t just a platform for cat pictures and pwnd IP cameras — things like the Server header were cool because it might help us create server-specific interactions and build cool stuff. Yes, modern day REST APIs are likely better in the long run but the naiveté of the silver age of the internet was definitely something special (and also led to the chaos we have now). But, I digress.

In theory, no HTTP server in it’s rightly configured digital mind would tell you what it’s running down to the version level, but most do. (Again, feel free to pick nits that I let the world know I run nginx…or *do* I). Assuming the CRAN mirrors haven’t been configured to deceive attackers and report what folks told them to report we can survey what they run behind the browser window:

filter(hdrs, name == "server") %>%

separate(

value, c("kind", "version"), sep="/", fill="right", extra="merge"

) -> svr

count(svr, kind, sort=TRUE)

| **kind** | **n** |
| --- | --- |
| Apache | 57 |
| nginx | 15 |
| cloudflare | 2 |
| CSIRO | 1 |
| Hiawatha v10.8.4 | 1 |
| High Performance 8bit Web Server | 1 |
| none | 1 |
| openresty | 1 |

I really hope Cloudflare is donating bandwidth vs charging these mirror sites. They’ve likely benefitted greatly from the diverse FOSS projects many of these sites serve. (I hadn’t said anything bad about Cloudflare yet so I had to get one in before the end).

Lots run Apache (makes sense since CRAN-proper does too, not that I can validate that from home since I’m IP blocked…*bitter much, hrbrmstr*?) Many run nginx. CSIRO likely names their server that on purpose and hasn’t actually written their own web server. Hiawatha is, indeed, a valid web server. While there are also “high performance 8bit web servers” out there I’m willing to bet that’s a joke header value along with “none”. Finally, “openresty” is also a valid web server (it’s nginx++).

We’ll pick on Apache and nginx and see how current patch levels are. Not all return a version number but a good chunk do:

apache\_httpd\_version\_history() %>%

arrange(rls\_date) %>%

mutate(

vers = factor(as.character(vers), levels = as.character(vers))

) -> apa\_all

filter(svr, kind == "Apache") %>%

filter(!is.na(version)) %>%

mutate(version = stri\_replace\_all\_regex(version, " .\*$", "")) %>%

count(version) %>%

separate(version, c("maj", "min", "pat"), sep="\\.", convert = TRUE, fill = "right") %>%

mutate(pat = ifelse(is.na(pat), 1, pat)) %>%

mutate(v = sprintf("%s.%s.%s", maj, min, pat)) %>%

mutate(v = factor(v, levels = apa\_all$vers)) %>%

arrange(v) -> apa\_vers

filter(apa\_all, vers %in% apa\_vers$v) %>%

arrange(rls\_date) %>%

group\_by(rls\_year) %>%

slice(1) %>%

ungroup() %>%

arrange(rls\_date) -> apa\_yrs

ggplot() +

geom\_blank(

data = apa\_vers, aes(v, n)

) +

geom\_segment(

data = apa\_yrs, aes(vers, 0, xend=vers, yend=Inf),

linetype = "dotted", size = 0.25, color = "white"

) +

geom\_segment(

data = apa\_vers, aes(v, n, xend=v, yend=0),

color = ft\_cols$gray, size = 8

) +

geom\_label(

data = apa\_yrs, aes(vers, Inf, label = rls\_year),

family = font\_rc, color = "white", fill = "#262a31", size = 4,

vjust = 1, hjust = 0, nudge\_x = 0.01, label.size = 0

) +

scale\_y\_comma(limits = c(0, 15)) +

labs(

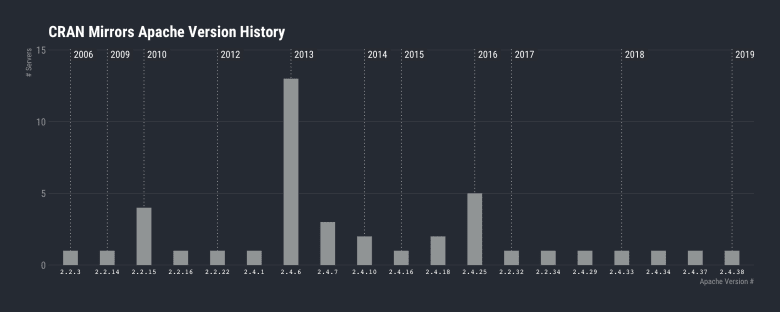
x = "Apache Version #", y = "# Servers",

title = "CRAN Mirrors Apache Version History"

) +

theme\_ft\_rc(grid="Y") +

theme(axis.text.x = element\_text(family = "mono", size = 8, color = "white"))

[](https://i2.wp.com/rud.is/b/wp-content/uploads/2019/03/apache-history-1.png?ssl=1)

O\_O

I’ll let you decide if a six-year-old version of Apache indicates how well a mirror site is run or not. Sure, mitigations could be in place but I see no statement of efficacy on any site so we’ll go with #lazyadmin.

But, it’s gotta be better with nginx, right? It’s all cool & modern!

nginx\_version\_history() %>%

arrange(rls\_date) %>%

mutate(

vers = factor(as.character(vers), levels = as.character(vers))

) -> ngx\_all

filter(svr, kind == "nginx") %>%

filter(!is.na(version)) %>%

mutate(version = stri\_replace\_all\_regex(version, " .\*$", "")) %>%

count(version) %>%

separate(version, c("maj", "min", "pat"), sep="\\.", convert = TRUE, fill = "right") %>%

mutate(v = sprintf("%s.%s.%s", maj, min, pat)) %>%

mutate(v = factor(v, levels = ngx\_all$vers)) %>%

arrange(v) -> ngx\_vers

filter(ngx\_all, vers %in% ngx\_vers$v) %>%

arrange(rls\_date) %>%

group\_by(rls\_year) %>%

slice(1) %>%

ungroup() %>%

arrange(rls\_date) -> ngx\_yrs

ggplot() +

geom\_blank(

data = ngx\_vers, aes(v, n)

) +

geom\_segment(

data = ngx\_yrs, aes(vers, 0, xend=vers, yend=Inf),

linetype = "dotted", size = 0.25, color = "white"

) +

geom\_segment(

data = ngx\_vers, aes(v, n, xend=v, yend=0),

color = ft\_cols$gray, size = 8

) +

geom\_label(

data = ngx\_yrs, aes(vers, Inf, label = rls\_year),

family = font\_rc, color = "white", fill = "#262a31", size = 4,

vjust = 1, hjust = 0, nudge\_x = 0.01, label.size = 0

) +

scale\_y\_comma(limits = c(0, 15)) +

labs(

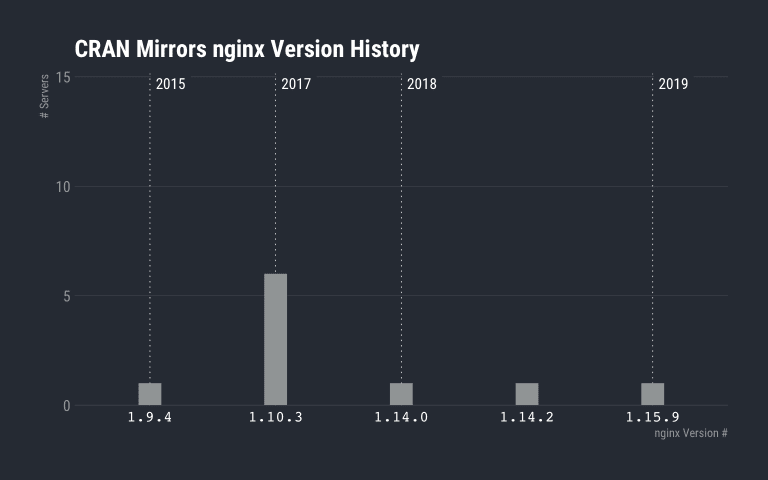
x = "nginx Version #", y = "# Servers",

title = "CRAN Mirrors nginx Version History"

) +

theme\_ft\_rc(grid="Y") +

theme(axis.text.x = element\_text(family = "mono", color = "white"))

[](https://i1.wp.com/rud.is/b/wp-content/uploads/2019/03/nginx-history-1.png?ssl=1)



I will at close out this penultimate section with a “thank you!” to the admins at Georg-August-Universität Göttingen and Yamagata University for keeping up with web server patches.

**You Made It This Far**

If I had known you’d read to the nigh bitter end I would have made cookies. You’ll have to just accept the ones the blog gives your browser (those ones taste taste pretty bland tho).

The last lightweight element we’ll look at is “what else do these ‘secure’ CRAN mirrors run”?

To do this, we’ll turn to [Rapid7 OpenData](https://opendata.rapid7.com/) and look at what else is running on the IP addresses used by these CRAN mirrors. We already know some certs are promiscuous, so what about the servers themselves?

cran\_mirror\_other\_things <- readRDS(here::here("data/cran-mirror-other-things.rds"))

# "top" 20

distinct(cran\_mirror\_other\_things, ip, port) %>%

count(ip, sort = TRUE) %>%

head(20)

| **ip** | **n** |
| --- | --- |
| 104.25.94.23 | 8 |
| 143.107.10.17 | 7 |
| 104.27.133.206 | 5 |
| 137.208.57.37 | 5 |
| 192.75.96.254 | 5 |
| 208.81.1.244 | 5 |
| 119.40.117.175 | 4 |
| 130.225.254.116 | 4 |
| 133.24.248.17 | 4 |
| 14.49.99.238 | 4 |
| 148.205.148.16 | 4 |
| 190.64.49.124 | 4 |
| 194.214.26.146 | 4 |
| 200.236.31.1 | 4 |
| 201.159.221.67 | 4 |
| 202.90.159.172 | 4 |
| 217.31.202.63 | 4 |
| 222.66.109.32 | 4 |
| 45.63.11.93 | 4 |
| 62.44.96.11 | 4 |

Four isn’t bad since we kinda expect at least 80, 443 and 21 (FTP) to be running. We’ll take those away and look at the distribution:

distinct(cran\_mirror\_other\_things, ip, port) %>%

filter(!(port %in% c(21, 80, 443))) %>%

count(ip) %>%

count(n) %>%

mutate(n = factor(n)) %>%

ggplot() +

geom\_segment(

aes(n, nn, xend = n, yend = 0), size = 10, color = ft\_cols$gray

) +

scale\_y\_comma() +

labs(

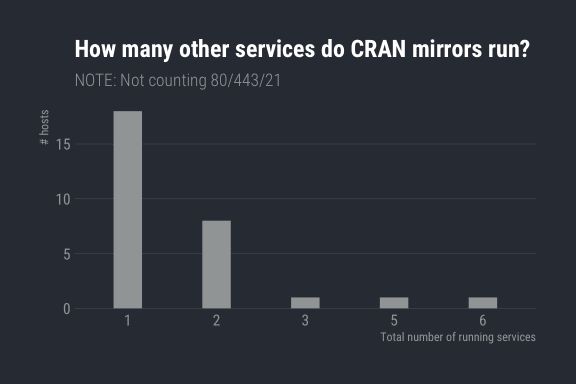
x = "Total number of running services", y = "# hosts",

title = "How many other services do CRAN mirrors run?",

subtitle = "NOTE: Not counting 80/443/21"

) +

theme\_ft\_rc(grid="Y")

[](https://i0.wp.com/rud.is/b/wp-content/uploads/2019/03/other-stuff-1.png?ssl=1)

So, what are these other ports?

distinct(cran\_mirror\_other\_things, ip, port) %>%

count(port, sort=TRUE)

| **port** | **n** |
| --- | --- |
| 80 | 75 |
| 443 | 75 |
| 21 | 29 |
| 22 | 18 |
| 8080 | 6 |
| 25 | 5 |
| 53 | 2 |
| 2082 | 2 |
| 2086 | 2 |
| 8000 | 2 |
| 8008 | 2 |
| 8443 | 2 |
| 111 | 1 |
| 465 | 1 |
| 587 | 1 |
| 993 | 1 |
| 995 | 1 |
| 2083 | 1 |
| 2087 | 1 |

22 is SSH, 53 is DNS, 8000/8008/8080/8553 are web high ports usually associated with admin or API endpoints and generally a bad sign when exposed externally (especially on a “secure” mirror server). 25/465/587/993/995 all deal with mail sending and reading (not exactly a great service to have on a “secure” mirror server). I didn’t poke too hard but 208[2367] tend to be cPanel admin ports and those being internet-accessible is also not great.

Port 111 is sunrpc and is a *really bad thing* to expose to the internet or to [run at all](http://lmgtfy.com/?q=port+111+sunrpc+exploit). But, the server is a “secure” CRAN mirror, so perhaps everything is fine.

**FIN**

While I hope this posts informs, I’ve worked in cybersecurity for ages and — as a result — don’t really expect anything to change. Tomorrow, I’ll still be blocked from the main CRAN & r-project.org site despite having better “security” than the vast majority of these “secure” CRAN mirrors (and was following the rules). Also CRAN mirror settings tend to be fairly invisible since most modern R users use the RStudio default (which is really not a bad choice from any “security” analysis angle), choose the first item in the mirror-chooser (Russian roulette!), or live with the setting in the site-wide Rprofile anyway (org-wide risk acceptance/”blame the admin”).