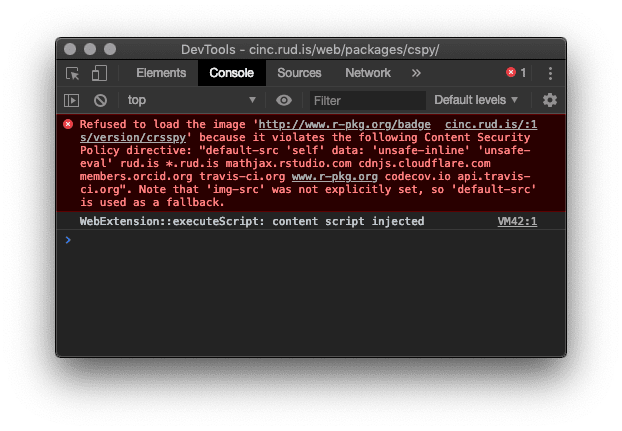
To oversimplify things, the CSP header instructs a browser on what you’ve authorized to be part of your site. You supply directives for different types of content which tell browsers what sites can load content into the current page in an effort to prevent things like [cross-site scripting attacks](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)), [malicious iframes](https://kb.sucuri.net/malware/malicious-iframes), [clickjacking](https://www.owasp.org/index.php/Clickjacking), [cryptojacking](https://www.malwarebytes.com/cryptojacking/), and more.

The web is awash in resources (some aforelinked) on how to configure CSPs and doesn’t need another guide (and this post is about working with CSPs as a data source in R). However, if you aren’t currently using CSPs on any sites you own and want a fairly easy way to develop a policy, grab a copy of the latest Firefox, install Mozilla’s own [CSP Toolkit extension](https://addons.mozilla.org/en-US/firefox/addon/laboratory-by-mozilla/) and just browse your site with that plugin active. It will generate as minimal of a policy as possible as it looks at all the resource requests your site makes. Another method is to use Content-Security-Policy-Report-Only header vs the Content-Security-Policy header since the former one will just report potential resource loading issues instead of blocking the resource loading.

**How About Some R Code Now, Mmmkay?**

Even if you do not have CSPs of your own they’re easy to find since they come along for the ride (when they exist) in HTTP server response headers. To make it easier to see if a site has a CSP and also load a CSP I’ve put together a small package (installation instructions & source locations are on that link) which can do this for us. (If you visit that page you’ll see a failed image load for the CRAN Status Badge. Open up Developer Tools or the javascript console in your browser and you’ll see the CSP rules hard at work:

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

The cspy package is based on Shape Security’s which implements a bona fide parser for CSP rules (as opposed to hacky-string-slicing which we’re also going to do in this post) and has a set of safety checker rules written in R. I violated the *“supporting JARs in one pkg; actual functionality in another pkg”* “rule” since Shape Security only does ~1.5 releases a year and it’s a super small JAR with no dependencies.

Let’s find an R-focused site with a CSP we can work with (assume the tidyverse and stringi are loaded into the R session). We’ll start with a super spiffy R org: RStudio!:

cspy::has\_csp("https://www.rstudio.com/")

## [1] FALSE

Oh, um…well, what about another super spiffy R org: Mango Solutions!:

cspy::has\_csp("http://mango-solutions.com/")

## [1] FALSE

*Heh*. Well…ah&hellip. Hrm. What about Data Camp! They have tons of great R things:

cspy::has\_csp("https://www.datacamp.com/")

## [1] TRUE

W00t!

To be fair, the main brand sites for both RStudio and Mango use WordPress and WordPress makes it super hard to write a decent CSP. However, (IMO) that’s kinda no excuse for not having one (since I run WordPress and have one). But, Data Camp does! So, let’s fetch it and take a look:

(dc\_csp <- cspy::fetch\_csp("https://www.datacamp.com/"))

## default-src 'self' \*.datacamp.com \*.datacamp-staging.com

## base-uri 'self'

## connect-src 'self' \*.datacamp.com \*.datacamp-staging.com https://com-datacamp.mini.snplow.net https://www2.profitwell.com https://www.facebook.com https://www.optimalworkshop.com https://in.hotjar.com https://stats.g.doubleclick.net https://\*.google-analytics.com https://frstre.com https://onesignal.com https://\*.algolia.net https://\*.algolianet.com https://wootric-eligibility.herokuapp.com https://d8myem934l1zi.cloudfront.net https://production.wootric.com

## font-src 'self' \*.datacamp.com \*.datacamp-staging.com https://fonts.gstatic.com data:

## form-action 'self' \*.datacamp.com \*.datacamp-staging.com https://\*.paypal.com https://www.facebook.com

## frame-ancestors 'self' \*.datacamp.com \*.datacamp-staging.com

## frame-src 'self' \*.datacamp.com \*.datacamp-staging.com \*.zuora.com https://www.google.com https://vars.hotjar.com https://beacon.tapfiliate.com https://b.frstre.com https://onesignal.com https://tpc.googlesyndication.com https://\*.facebook.com https://\*.youtube.com

## img-src 'self' \*.datacamp.com \*.datacamp-staging.com https://s3.amazonaws.com https://checkout.paypal.com https://cdnjs.cloudflare.com https://www.gstatic.com https://maps.googleapis.com https://maps.gstatic.com https://www.facebook.com https://q.quora.com https://bat.bing.com https://\*.ads.linkedin.com https://www.google.com https://\*.g.doubleclick.net https://\*.google-analytics.com https://www.googletagmanager.com blob data:

## script-src 'self' 'unsafe-inline' 'unsafe-eval' \*.datacamp.com \*.datacamp-staging.com https://\*.googletagmanager.com https://\*.google-analytics.com https://browser.sentry-cdn.com https://js.braintreegateway.com https://\*.zuora.com https://\*.paypal.com https://js-agent.newrelic.com https://bam.nr-data.net https://maps.googleapis.com https://assets.calendly.com http://com-datacamp.mini.snplow.net https://cdn.jsdelivr.net https://\*.algolia.net https://www.gstatic.com https://www.google.com https://cdnjs.cloudflare.com https://static.tapfiliate.com https://cdn.onesignal.com https://onesignal.com https://static.hotjar.com https://script.hotjar.com https://\*.linkedin.com https://sjs.bizographics.com/insight.min.js https://bat.bing.com/bat.js https://a.quora.com/qevents.js https://connect.facebook.net https://www.googleadservices.com https://www.googletagmanager.com https://\*.googlesyndication.com https://dna8twue3dlxq.cloudfront.net/js/profitwell.js https://disutgh7q0ncc.cloudfront.net/beacon.js data:

## style-src 'self' 'unsafe-inline' \*.datacamp.com \*.datacamp-staging.com https://cdnjs.cloudflare.com https://fonts.googleapis.com https://assets.calendly.com

## worker-src 'self'

## report-uri https://infra-reporter.datacamp.com/csp-report

Wow. Literally: *wow*. We’ll let their recent breach and lack of 2FA for logins slide for a bit b/c this is a robust CSP so it’s obvious they do care about you (regardless of whatever CSP helper tools are out there these CSPs are kind of a pain to setup so the fact that this one is so thorough is in indicator they “get it”).

R folks are likely looking at that untidy mess and cringing, though. We’ll tidy it right up, soon, but let’s make sure it’s a valid CSP:

glimpse(validate\_csp(dc\_csp))

## Observations: 1

## Variables: 8

## $ message "A draft of the next version of CSP deprecates report-uri in favour …

## $ type "INFO"

## $ start\_line 1

## $ start\_column 2680

## $ start\_offset 2679

## $ end\_line 1

## $ end\_column 2690

## $ end\_offset 2689

The validator only noted that a soon-to-be deprecated directive is being used (but that’s just informational and perfectly fine until CSP v3 is out of Draft and browsers support it) and the validation output has positional values for where the violations occur (which shows the benefits of using a true parser vs string splitter).

The custom print method for these csp objects puts each directive on a separate line and has left the resource values intact. As noted, it’s ugly and untidy so let’s take care of that:

(csp\_df <- as.data.frame(dc\_csp))

## # A tibble: 111 x 3

## directive value origin

##

## 1 default-src 'self' https://www.datacamp.com/

## 2 default-src \*.datacamp.com https://www.datacamp.com/

## 3 default-src \*.datacamp-staging.com https://www.datacamp.com/

## 4 base-uri 'self' https://www.datacamp.com/

## 5 connect-src 'self' https://www.datacamp.com/

## 6 connect-src \*.datacamp.com https://www.datacamp.com/

## 7 connect-src \*.datacamp-staging.com https://www.datacamp.com/

## 8 connect-src https://com-datacamp.mini.snplow.net https://www.datacamp.com/

## 9 connect-src https://www2.profitwell.com https://www.datacamp.com/

## 10 connect-src https://www.facebook.com https://www.datacamp.com/

## # … with 101 more rows

Now we have one row for each directive/value pair. The origin (where the policy came from) also comes along for the ride in the event you want to bind a bunch of these together.

Let’s see how many third-party sites they had to catch and add to this policy:

filter(csp\_df, stri\_detect\_fixed(value, ".")) %>%

pull(value) %>%

stri\_replace\_all\_fixed("\*.", "") %>%

urltools::url\_parse() %>%

distinct(domain) %>%

filter(!stri\_detect\_regex(domain, "(datacamp\\.com|datacamp-staging\\.com)$")) %>%

arrange(domain) %>%

pull(domain)

## [1] "a.quora.com" "ads.linkedin.com" "algolia.net"

## [4] "algolianet.com" "assets.calendly.com" "b.frstre.com"

## [7] "bam.nr-data.net" "bat.bing.com" "beacon.tapfiliate.com"

## [10] "browser.sentry-cdn.com" "cdn.jsdelivr.net" "cdn.onesignal.com"

## [13] "cdnjs.cloudflare.com" "checkout.paypal.com" "com-datacamp.mini.snplow.net"

## [16] "connect.facebook.net" "d8myem934l1zi.cloudfront.net" "disutgh7q0ncc.cloudfront.net"

## [19] "dna8twue3dlxq.cloudfront.net" "facebook.com" "fonts.googleapis.com"

## [22] "fonts.gstatic.com" "frstre.com" "g.doubleclick.net"

## [25] "google-analytics.com" "googlesyndication.com" "googletagmanager.com"

## [28] "in.hotjar.com" "js-agent.newrelic.com" "js.braintreegateway.com"

## [31] "linkedin.com" "maps.googleapis.com" "maps.gstatic.com"

## [34] "onesignal.com" "paypal.com" "production.wootric.com"

## [37] "q.quora.com" "s3.amazonaws.com" "script.hotjar.com"

## [40] "sjs.bizographics.com" "static.hotjar.com" "static.tapfiliate.com"

## [43] "stats.g.doubleclick.net" "tpc.googlesyndication.com" "vars.hotjar.com"

## [46] "wootric-eligibility.herokuapp.com" "www.facebook.com" "www.google.com"

## [49] "www.googleadservices.com" "www.googletagmanager.com" "www.gstatic.com"

## [52] "www.optimalworkshop.com" "www2.profitwell.com" "youtube.com"

## [55] "zuora.com"

That list is one big reason these CSPs can be gnarly to configure and also why some argue there’s potentially little benefit in them. Why little benefit? Let’s see what sites they’ve allowed browsers to load and execute javascript content from:

filter(csp\_df, directive == "script-src") %>%

select(value) %>%

filter(!stri\_detect\_regex(value, "(datacamp\\.com$|datacamp-staging\\.com$|'|data:)")) %>%

pull(value)

## [1] "https://\*.googletagmanager.com" "https://\*.google-analytics.com"

## [3] "https://browser.sentry-cdn.com" "https://js.braintreegateway.com"

## [5] "https://\*.zuora.com" "https://\*.paypal.com"

## [7] "https://js-agent.newrelic.com" "https://bam.nr-data.net"

## [9] "https://maps.googleapis.com" "https://assets.calendly.com"

## [11] "http://com-datacamp.mini.snplow.net" "https://cdn.jsdelivr.net"

## [13] "https://\*.algolia.net" "https://www.gstatic.com"

## [15] "https://www.google.com" "https://cdnjs.cloudflare.com"

## [17] "https://static.tapfiliate.com" "https://cdn.onesignal.com"

## [19] "https://onesignal.com" "https://static.hotjar.com"

## [21] "https://script.hotjar.com" "https://\*.linkedin.com"

## [23] "https://sjs.bizographics.com/insight.min.js" "https://bat.bing.com/bat.js"

## [25] "https://a.quora.com/qevents.js" "https://connect.facebook.net"

## [27] "https://www.googleadservices.com" "https://www.googletagmanager.com"

## [29] "https://\*.googlesyndication.com" "https://dna8twue3dlxq.cloudfront.net/js/profitwell.js"

## [31] "https://disutgh7q0ncc.cloudfront.net/beacon.js"

Some of those are specified all the way down to the .js file but others work across entire CDN networks. Granted, there’s nothing super bad like Amazon’s S3 apex domains (wildcarded or not) but there’s alot of trust going on in that list. The good thing about CSPs is that if Data Camp learns about an issue with any of those sites it can quickly zap them from the header since they’ve already done the hard enumeration work. Plus, if one of those sites is compromised and loads a resource that tries to execute code from a domain not in that list, a policy error will be generated and Data Camp’s excellent SOC (they responded super well to the breach) can take active measures to inform their downstream provider that there’s something wrong.

We can also do some safety checks:

check\_script\_unsafe\_inline(csp\_df)

## Category: script-unsafe-inline

## Severity: HIGH

## Message: 'unsafe-inline' allows the execution of unsafe in-page scripts and event handlers.

##

## directive value origin

## 67 script-src 'unsafe-inline' https://www.datacamp.com/

check\_script\_unsafe\_eval(csp\_df)

## Category: script-unsafe-eval

## Severity: HIGH

## Message: 'unsafe-eval' allows the execution of code injected into DOM APIs such as eval().

##

## directive value origin

## 68 script-src 'unsafe-eval' https://www.datacamp.com/

check\_plain\_url\_schemes(csp\_df)

## Category: unsafe-execution

## Severity: HIGH

## Message: URI(s) found that allow the exeution of unsafe scripts.

##

## directive value origin

## 102 script-src data: https://www.datacamp.com/

check\_wildcards(csp\_df)

check\_missing\_directives(csp\_df)

## Category: missing-directive

## Severity: HIGH

## Message: Missing object-src allows the injection of plugins which can execute JavaScript. Can you set it to 'none'?

##

## directive value

## 1 object-src

check\_ip\_source(csp\_df)

check\_deprecated(csp\_df)

## Category: deprecated-directive

## Severity: INFO

## Message: Found deprecated directive(s). See https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy for more information.

##

## directive value

## 111 report-uri https://infra-reporter.datacamp.com/csp-report

## origin

## 111 https://www.datacamp.com/

check\_nonce\_length(csp\_df)

check\_src\_http(csp\_df)

## Category: http-src

## Severity: MEDIUM

## Message: Use HTTPS vs HTTP.

##

## directive value

## 81 script-src http://com-datacamp.mini.snplow.net

## origin

## 81 https://www.datacamp.com/

Along with pretty printing they return data frames of potentially problematic rules.

For some pages, Data Camp does what you likely do when you knit an R markdown document: save it as a self-contained page. Because of that it is absolutely necessary to use 'unsafe-inline', 'unsafe-eval', and data: directive values for a few directives. It just can’t be helped. We already knew about the report-uri deprecation. While object-src isn’t in the CSP-proper, there is a default-src and that will take effect. But, the checker knows it’s not set to 'none' which means plugins can still load from Data Camp’s own domains (which may be perfectly fine depending on the type of embeds they use).

The one finding I’ll have to side with the checker on is the one at the end (the output of check\_src\_http()). They seem to allow one of the many analytics providers for the site use http:// vs https://. If they really do load javascript content from that site via plain http:// then it would be possible for an attacker to hijack the content and insert malicious javascript. Yet another good reason to use DNS-based uBlock Origin to prevent any trackers or analytics scripts from loading at all.

One further thing to note is that Data Camp uses Amazon AWS for their site hosting and you *kinda* have to jump through hoops to get this header added there so they have seriously done a decent job thinking about your safety.

**At-Scale Wrangling**

The cspy package is fine for casual/light inspection but it’s uses a Java shim which means marshaling of data between R and the JVM and that’s at-scale. Depending on what you’re trying to do, some basic string wrangling may suffice.

At $DAYJOB we perform many [HTTP scans](https://opendata.rapid7.com/sonar.https/). One scan we have internally (I don’t think it’s part of Open Data but I’ll check Monday) is a regular grab of the Alexa Top 1m named hosts. In the most recent scan we found ~65K sites with a CSP header. You can grab a CSV containing the virtual host name and the found CSP policy string file top1m-csp.csv.gz attached with this repository.

Once you download that we can just use string ops to get to a tidy data frame:

read\_csv("top1m-csp.csv.gz", col\_types = "cc") %>% # use the location where you stored it

mutate(csp = stri\_trans\_tolower(csp)) -> top1m\_csp

stri\_split\_regex(top1m\_csp$csp, ";[[:space:]]\*", simplify = FALSE) %>% # split directives for each policy

map2\_df(top1m\_csp$vhost, ~{ # we want to keep the origin

stri\_match\_first\_regex(.x, "([[:alpha:]\\-]+)[[:space:]]+(.\*)$|([[:alpha:]\\-]+)") %>% # get the directive

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

mutate(origin = sprintf("https://%s/", .y)) %>%

mutate(raw = .x) # save the raw CSP

}) %>%

mutate(V2 = ifelse(is.na(V2), V4, V2)) %>% # figure out directive

mutate(V3 = ifelse(is.na(V3), "", V3)) %>% # and value (keeping in mind no-value directives)

select(directive = V2, value = V3, origin, raw) %>%

as\_tibble() %>%

separate\_rows(value, sep="[[:space:]]+") -> top1m\_csp # tidy it up with each row being directive|value|origin like the above

That took just about 2 minutes for me and that’s two minutes too long to spread around to intrepid readers who may be trying this at home so an RDS of the result.

So, only ~65K out of the top 1m sites have a CSP header (~6.5%) but that doesn’t mean they’re *good*. Let’s take a look at the invalid directives they’ve used:

top1m\_csp %>%

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

count(directive, sort = TRUE) %>%

mutate(is\_valid = directive %in% valid\_csp\_directives) %>%

filter(!is\_valid) %>%

select(-is\_valid) %>%

print(n=65)

## # A tibble: 65 x 2

## directive n

##

## 1 referrer 69

## 2 reflected-xss 58

## 3 t 50

## 4 self 34

## 5 tscript-src 22

## 6 tframe-src 17

## 7 timg-src 16

## 8 content-security-policy 14

## 9 default 10

## 10 policy-definition 9

## 11 allow 8

## 12 defalt-src 8

## 13 tstyle-src 8

## 14 https 7

## 15 nosniff 7

## 16 null 7

## 17 tfont-src 6

## 18 unsafe-inline 6

## 19 http 5

## 20 allow-same-origin 4

## 21 content 4

## 22 defaukt-src 4

## 23 default-scr 4

## 24 defaut-src 4

## 25 data 3

## 26 frame-ancestor 3

## 27 policy 3

## 28 self-ancestors 3

## 29 defualt-src 2

## 30 jivosite 2

## 31 mg-src 2

## 32 none 2

## 33 object 2

## 34 options 2

## 35 policy-uri 2

## 36 referrer-policy 2

## 37 tconnect-src 2

## 38 tframe-ancestors 2

## 39 xhr-src 2

## 40 - 1

## 41 -report-only 1

## 42 ahliunited 1

## 43 anyimage 1

## 44 blok-all-mixed-content 1

## 45 content-security-policy-report-only 1

## 46 default-sec 1

## 47 disown-opener 1

## 48 dstyle-src 1

## 49 frame-ancenstors 1

## 50 frame-ancestorss 1

## 51 frame-ancestros 1

## 52 mode 1

## 53 navigate-to 1

## 54 origin 1

## 55 plugin-type 1

## 56 same-origin 1

## 57 strict-dynamic 1

## 58 tform-action 1

## 59 tni 1

## 60 upgrade-insecure-request 1

## 61 v 1

## 62 value 1

## 63 worker-sc 1

## 64 x-frame-options 1

## 65 yandex 1

Spelling errors are more common than one might have hoped (another reason CSPs are hard since humans tend to have to make them).

We can also see the directives that are most used:

top1m\_csp %>%

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

count(directive, sort = TRUE) %>%

mutate(is\_valid = directive %in% valid\_csp\_directives) %>%

filter(is\_valid) %>%

select(-is\_valid) %>%

print(n=25)

## # A tibble: 25 x 2

## directive n

##

## 1 script-src 57636

## 2 frame-ancestors 47685

## 3 upgrade-insecure-requests 47515

## 4 block-all-mixed-content 38038

## 5 report-uri 37483

## 6 default-src 36369

## 7 img-src 35093

## 8 style-src 23005

## 9 connect-src 22415

## 10 frame-src 15827

## 11 font-src 12878

## 12 media-src 6718

## 13 child-src 5043

## 14 object-src 4235

## 15 worker-src 3076

## 16 form-action 1522

## 17 base-uri 861

## 18 manifest-src 374

## 19 sandbox 79

## 20 script-src-elem 54

## 21 plugin-types 47

## 22 report-to 32

## 23 prefetch-src 22

## 24 require-sri-for 14

## 25 style-src-elem 6

Curious readers should load up urltools and see what domains those script-src directives are allowing. Remember, every value entry that points to an third-party site means it’s OK for that site to mess with your browser or spy on you.

And, take a look at where folks are doing their CSP error reporting:

filter(top1m\_csp, directive %in% c("report-uri", "report-to")) %>%

mutate(report\_host = case\_when(

!grepl("^http[s]\*://", value) ~ origin,

TRUE ~ value

)) %>%

select(report\_host) %>%

mutate(report\_host = sub('"$', "", report\_host)) %>%

pull(report\_host) %>%

urltools::url\_parse() %>%

as\_tibble() %>%

count(domain, sort=TRUE) %>%

print(n=20)

## # A tibble: 18,565 x 2

## domain n

##

## 1 csp.medium.com 113

## 2 sentry.io 43

## 3 de.forumhome.com 40

## 4 report-uri.highwebmedia.com 33

## 5 csp.yandex.net 23

## 6 99designs.report-uri.io 20

## 7 endpoint1.collection.us2.sumologic.com 20

## 8 capture.condenastdigital.com 17

## 9 secure.booked.net 14

## 10 bimbobakeriesusa.com 10

## 11 oroweat.com 10

## 12 ponyfoo.com 8

## 13 monzo.report-uri.com 6

## 14 19jrymqg65.execute-api.us-east-1.amazonaws.com 5

## 15 app.getsentry.com 5

## 16 csp-report.airfrance.fr 5

## 17 csp.nic.cz 5

## 18 myklad.org 5

## 19 reports-api.sqreen.io 5

## 20 tracker.report-uri.com 5

## # … with 1.854e+04 more rows

The list (18K!) is far more diverse than I had anticipated.

You can do tons more (especially since the wrangling has been done for you :-).

**FIN**

A future post will show how to process CSP error reports in R and there will likely be more security/safety-themed posts as well.