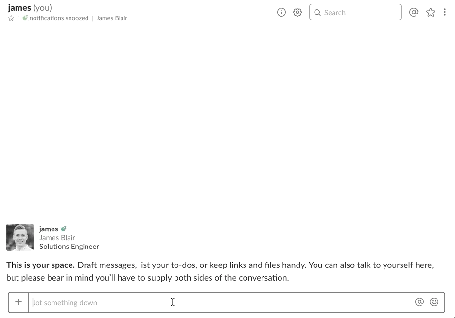
In this entry, we will secure the API created ,so that it only responds to authenticated requests, and deploy it using [RStudio Connect](https://www.rstudio.com/products/connect/).



As a reminder, this API is built on top of simulated customer call data. The slash command we create will allow users to view a customer status report within Slack. This status report contains customer name, total calls, date of birth, and a plot of call history for the past 20 weeks.

**Setup**

Successfully following this example assumes you have created a [Slack](https://slack.com/) account and you have [followed the instructions for creating an app](https://api.slack.com/slack-apps). The Plumber API as it currently exists.

This API can be run through the UI as previously described, or by running plumber::plumb("plumber.R")$run(port = 5762) from the directory containing the API defined in plumber.R. As it stands now, this API could be deployed and used by Slack. However, it’s important to remember that we have no control over the request that Slack makes to the API. Because of this, we can’t rely on RStudio Connect’s [built-in API authentication mechanism](http://docs.rstudio.com/connect/admin/content-management.html#api-keys) to secure the API because there is no way to submit a key with the request. Our options are either to expose the API with no security, meaning anyone can access the endpoints we’ve defined, or to find some other mechanism for securing the API so that it only responds to authorized requests.

**API Security Patterns**

The [plumber documentation](https://www.rplumber.io/docs/security.html) provides a good introduction to API security for the R user:

The majority of R programmers have not been trained to give much attention to the security of the code that they write. This is for good reason since running R code on your own machine with no external input gives little opportunity for attackers to leverage your R code to do anything malicious. However, as soon as you expose an API on a network, your concerns and thought process must adapt accordingly.

API security can be challenging to address. As it stands today, it is the developer’s responsibility to provide proper security on API endpoints, though in the future, there may be additional security features added to plumber or available via other R packages.

As mentioned in the [plumber documentation](https://www.rplumber.io/docs/security.html), there are a number of things to consider when designing API security. For example, if the API is deployed on an internal network, securing the API may not be as important as it would be if the API was publicly exposed on the internet. When an API needs to be secured, there are several potential attack vectors that need to be handled. In this specific example, we are exposing a public endpoint that provides access to sensitive customer data. If we are unable to authenticate incoming requests, then we risk exposing sensitive data. To prevent this data from falling into the wrong hands, we will focus on verifying incoming requests so that the API only responds to requests made from Slack.

There are several different methods for authenticating requests made to API endpoints. One common method is the use of API keys, which are cryptographically secure values sent with the request to verify the identity of the client. However, in this case, we have no control over the request Slack sends, so we cannot include such a key in the request. Thankfully, Slack has provided an alternative authentication method using [signed secrets](https://api.slack.com/docs/verifying-requests-from-slack). Full details can be read in the Slack documentation, but in essence, each Slack application is assigned a unique secret value that, when used in connection with other request details, can be used to verify that an incoming request is indeed coming from Slack and not an unknown third party.

**Securing the API**

In order to secure our API so that only requests from Slack are honored, we first need to obtain the signing secret for our application. This value can be found in the Basic Information section of the Slack application settings. Now, it is important to remember that this is called a signing secret for a reason: it should not be shared with anyone. To avoid exposing this secret, we can save it as an environment variable. We add this in a current R session by using Sys.setenv(SLACK\_SIGNING\_SECRET = ), or we can add it to our [.Renviron](https://csgillespie.github.io/efficientR/set-up.html#renviron) file so that it is set for every R session. Once this is done, we can access this value in R using Sys.getenv("SLACK\_SIGNING\_SECRET"). Now we are ready to create a function to verify if incoming requests are from Slack.

Slack provides the following three-step process for verifying requests:

* Your app receives a request from Slack
* Your app computes a signature based on the request
* You make sure the computed signature matches the signature on the request

In order to verify all incoming requests, we can define an additional filter for our API that follows the above recipe.

#\* Verify incoming requests

#\* @filter verify

function(req, res) {

# Forward requests coming to swagger endpoints

if (grepl("swagger", tolower(req$PATH\_INFO))) return(forward())

# Check for X\_SLACK\_REQUEST\_TIMESTAMP header

if (is.null(req$HTTP\_X\_SLACK\_REQUEST\_TIMESTAMP)) {

res$status <- 401

}

# Build base string

base\_string <- paste(

"v0",

req$HTTP\_X\_SLACK\_REQUEST\_TIMESTAMP,

req$postBody,

sep = ":"

)

# Slack Signing secret is available as environment variable

# SLACK\_SIGNING\_SECRET

computed\_request\_signature <- paste0(

"v0=",

openssl::sha256(base\_string, Sys.getenv("SLACK\_SIGNING\_SECRET"))

)

# If the computed request signature doesn't match the signature provided in the

# request, set status of response to 401

if (!identical(req$HTTP\_X\_SLACK\_SIGNATURE, computed\_request\_signature)) {

res$status <- 401

} else {

res$status <- 200

}

if (res$status == 401) {

list(

text = "Error: Invalid request"

)

} else {

forward()

}

}

There are a lot of moving pieces to this filter, but essentially we are following the process outlined by Slack for verifying requests. We also allow Swagger endpoints to be served without verification so that the Swagger UI can still be generated for our API.

Once this filter is in place, all incoming requests will be verified. However, this will create issues with our /plot/history/ endpoint since it is called using a standard GET request without any Slack authentication. To ensure that this endpoint is able to be utilized as we want, we’ll make some small updates to the endpoint and add #\* @preempt verify to the plumber comments before the function. This prevents the verify filter from applying to this endpoint.

Now, this prevents the Slack authentication process from applying to our plot endpoint. However, this endpoint, if left unsecured, provides unfiltered access to sensitive customer data. We need an effective way to secure this endpoint so that it only responds to requests generated from Slack.

Since the only thing we control in the request to this endpoint is the URL, we can update our endpoint so that an encrypted parameter is passed as part of the URL. This parameter is a combination of the current datetime and the customer ID that is then encrypted using our Slack signing secret. We can use the encrypt\_string() function from the safer package to securely encrypt this string. The following example illustrates this process.

Library(safer)

current\_time <- Sys.time()

customer\_id <- 89

parameter\_string <- paste(current\_time, customer\_id, sep = ";")

safer::encrypt\_string(parameter\_string, Sys.getenv("SLACK\_SIGNING\_SECRET"))

## [1] "m7NfMZfpY1n5EuivjuiFQsyKopT68HiX+NIgk5S+VBlDHrVqzRM="

Once we have created this encrypted value, we pass it to the URL of our plot endpoint. Then, within the plot endpoint, we decrypt the string, extract the customer ID, and check to see if the current time is within five seconds of the time encoded in the string. If more than five seconds have passed, we consider the request to be unauthorized. To help with this process, we define two helper functions:

encrypt\_string <- function(string) {

urltools::url\_encode(safer::encrypt\_string(paste(Sys.time(), string, sep = ";"),

key = Sys.getenv("SLACK\_SIGNING\_SECRET")))

}

plot\_auth <- function(endpoint, time\_limit = 5) {

# Save current time to compare against endpoint time value

current\_time <- Sys.time()

# Try to decrypt endpoint and extract user id

tryCatch({

# Decrypt endpoint using SLACK\_SIGNING\_SECRET

decrypted\_endpoint <- safer::decrypt\_string(endpoint,

key = Sys.getenv("SLACK\_SIGNING\_SECRET"))

# Split endpoint on ;

endpoint\_split <- unlist(strsplit(decrypted\_endpoint, split = ";"))

# Convert time

endpoint\_time <- as.POSIXct(endpoint\_split[1])

# Calculate time difference

time\_diff <- difftime(current\_time, endpoint\_time, units = "secs")

# If more than 5 seconds have passed since the request was generated, then

# error

if (time\_diff > time\_limit) {

"Unauthorized"

} else {

endpoint\_split[2]

}

},

error = function(e) "Unauthorized"

)

}

Once these helper functions are in place, we can update our /plot/history endpoint as follows:

#\* Plot customer weekly calls

#\* @png

#\* @param cust\_secret encrypted value calculated in /status endpoint

#\* @response 400 No customer with the given ID was found.

#\* @preempt verify

#\* @get /plot/history

function(res, cust\_secret) {

# Authenticate that request came from /status

cust\_id <- plot\_auth(cust\_secret)

# Return unauthorized error if cust\_id is "Unauthorized"

if (cust\_id == "Unauthorized") {

res$status <- 401

stop("Unauthorized request")

} else if (!cust\_id %in% sim\_data$id) {

res$status <- 400

stop("Customer id" , cust\_id, " not found.")

}

# Filter data to customer id provided

plot\_data <- dplyr::filter(sim\_data, id == cust\_id)

# Customer name (id)

customer\_name <- paste0(unique(plot\_data$name), " (", unique(plot\_data$id), ")")

# Create plot

history\_plot <- plot\_data %>%

ggplot(aes(x = time, y = calls, col = calls)) +

ggalt::geom\_lollipop(show.legend = FALSE) +

theme\_light() +

labs(

title = paste("Weekly calls for", customer\_name),

x = "Week",

y = "Calls"

)

# print() is necessary to render plot properly

print(history\_plot)

}

Now we need to make one small change to our /status endpoint so that we properly build the appropriate URL for our image. We construct the list response to the /status endpoint as follows, where image\_url has been updated.

attachments = list(

list(

color = customer\_status,

title = paste0("Status update for ", customer\_name, " (", customer\_id, ")"),

fallback = paste0("Status update for ", customer\_name, " (", customer\_id, ")"),

# History plot

image\_url = paste0(base\_url,

"/plot/history?cust\_secret=",

encrypt\_string(customer\_id)),

# Fields provide a way of communicating semi-tabular data in Slack

fields = list(

list(

title = "Total Calls",

value = sum(customer\_data$calls),

short = TRUE

),

list(

title = "DoB",

value = unique(customer\_data$dob),

short = TRUE

)

)

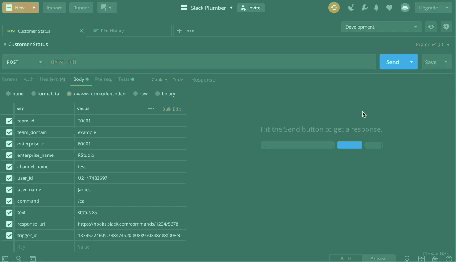
)

)

Just like that, we have a secure API!

**All Together Now**

Now, given the authorization pieces we have implemented, it is a bit more difficult to test our API since our endpoints will only respond to authorized requests.



It appears that everything is working as expected! Our endpoints fail when the authorization criteria are not met, and otherwise they succeed. Notice that the plot endpoint works when initially called, but when a subsequent call is made it fails since more than five seconds have passed since the /status endpoint was invoked.

Now, the final step in this process is publishing this API so that Slack can properly interact with it.

**Conclusion**

This brings us to the conclusion of this series. We’ve discovered the power of plumber in exposing R to downstream consumers via RESTful API endpoints. We built a Slack app powered entirely by R and plumber, and now we have secured the underlying API so that it only responds to authorized requests. As we have seen, plumber provides a powerful and flexible framework for exposing R functions as APIs. These APIs can be safely secured so that only authorized requests are permitted.