| Master in Cybersecurity  Course 2022-2023 |
| --- |
| Thesis App Setup |
| **Thesis** |



***Clem O'Regan, 100480776***

***Luis Cesar Miranda Mogollon, 100489053***

***Working Group: 3***

# Introduction

Our final assignment is based on various applications we have deployed in Google Kubernetes Engine. We have deployed a simple single leg non-secure application using the LAMP architecture in HTTP and also a multi-legged secure application in a distinct namespace where we have a secure architecture in place.

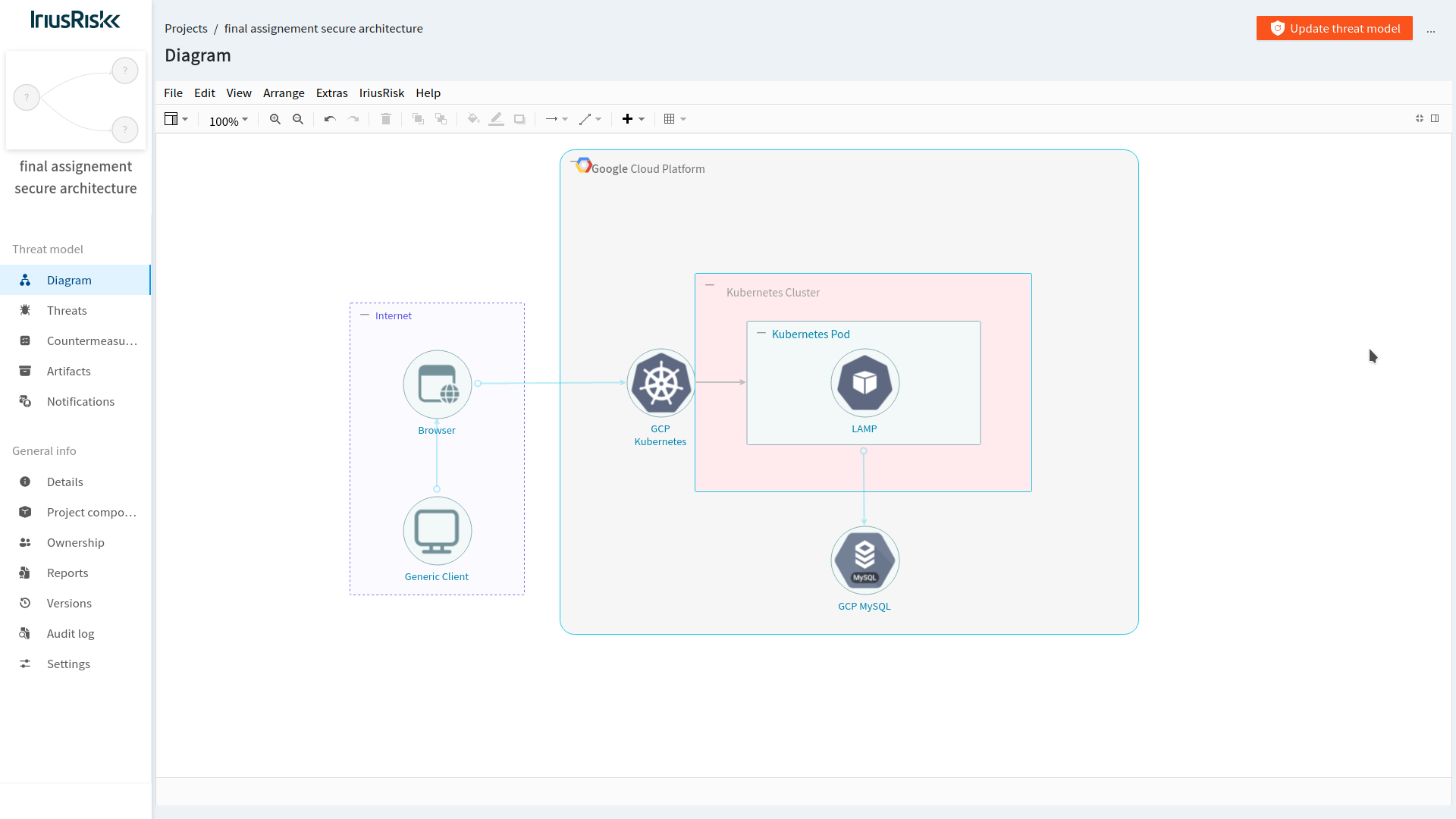
We will detail in this report:

* Technical architecture deployed
* Application detail
* Basic kubernetes configuration
* Secure architecture focusing on:
  + Network setup
  + DoS protection
  + TLS configuration
  + Scalability
  + Availability
* How we can simulate a DoS attack on both architectures and how we can protect against that attack.

Along with this report, we have also zipped up configuration files used to build the architecture and detailed any additional configuration required within the report.

## 

# Basic architecture



LAMP application architecture.

* For the base architecture we have created a LAMP architecture directly in GCP
* We created a dockerfile based on php-fpm. Into this dockerfile we developed a simple PHP app to connect to our MySQL instance, reading some basic data from the DB.
* We considered using persistent volumes (PV & PVCs) in Kubernetes, but as our app is simply reading from the DB we deployed GCP’s DBaaS which is more resilient and more comprehensive backups..
* The Dockerfile for the application can be found in the configuration zip we have included with the assignment submission:./fa\_basicapp/php/docker/Dockerfile

### Basic service architecture

For our basic application we use a Nodeport service which we are communicating with directly from the internet, using the EXTERNAL IP of the node where the pod is deployed.

## 

# Proposed protection measures - Secure Application

## 

Our secure architecture as depicted in the diagram, has the following elements:

* We used google cloud armor for our main application protection. We have set up cloud armor policies to protect against malicious payloads and also most importantly for this assignment to throttle limits against potential DoS attacks
* Application is secured with TLS via a Google managed certificate. In order to put this in place we used an external DNS provider (google domains) and integrated this with Google’s managed certificate through a HTTPS load balancer
* Traefik proxy is a modern reverse proxy and ingress controller that integrates with our infrastructure components and allows us to deploy applications fronted by traefik ingress service.
* The HTTPS load balancer is used to balance requests amongst our traefik instances. Here we are specifically using a HTTPS load balancer, offloading the TLS on the load balancer level. We have load balancer health checks in place, to ensure our traefik instance is up and running and awaiting application deployment events.
* We have setup the external DNS lookup to the IP of the load balancer, creating Google static IPs, to ensure the DNS lookup is stable
* Our application is deployed via helm via ClusterIP service, only being exposed to the outside world via traefik ingress controller, thus providing on centralized point for application access

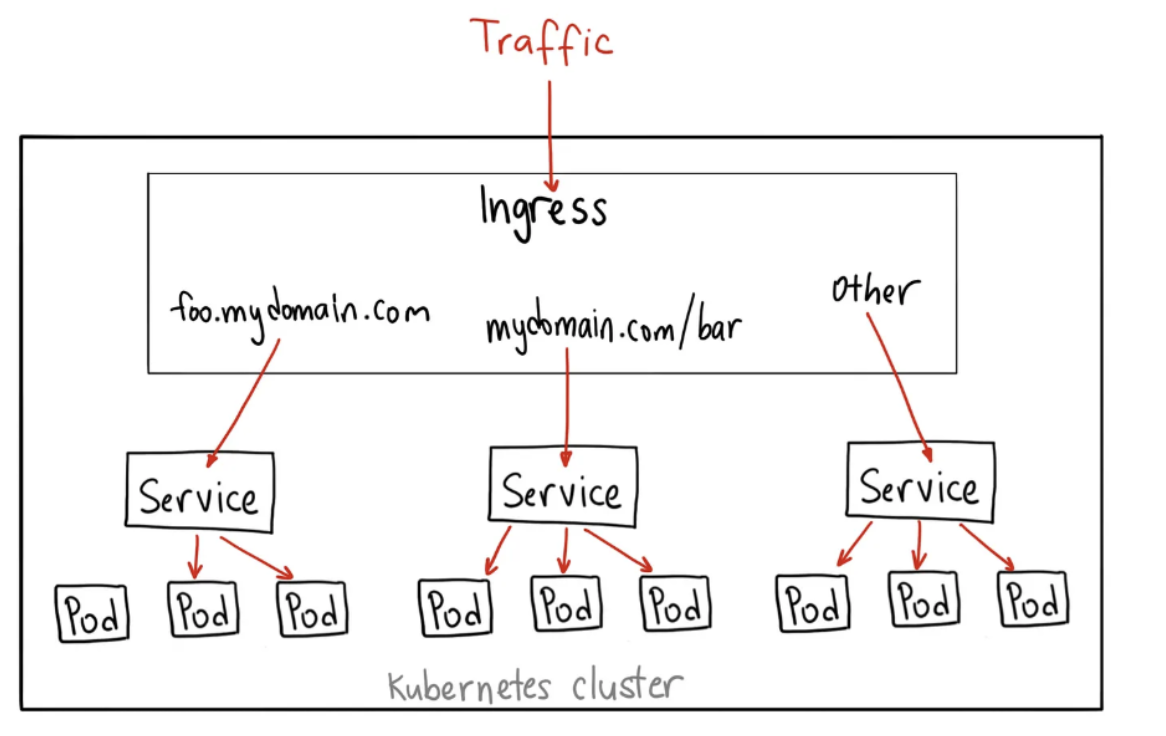
Along with the above network and service architecture we are also using the following secure architecture principles:

* Deployment of application via helm charts
* Availability for failover and distribution of load:
  + Pod autoscaling
  + Pod replicas
  + DB replicas
* Data protection
  + Ensuring all connections are via HTTPS
  + Whitelisting DB connections so that no one (including internal applications and services) can communicate with the DB unless specifically setup to do so
* Container security:
  + RBAC on application deployed with our helm chart
  + Service account for secure application deployed via helm chart
* Application security
  + Kubernetes secrets for sensitive credentials (DB access)
  + DB whitelisting

## Detailed setup for secure application

### Deploy traefik to cluster

We are using traefik as our main ingress controller into the cluster for our secure application, as per the following depiction:



We deploy traefik adding the traefik chart to our local helm repo:

| helm repo add traefik https://helm.traefik.io/traefik  helm repo update |
| --- |

We then add a values.yaml file to customize the deployed traefik for our cluster

| ingressClass:  enabled: true  isDefaultClass: false  additionalArguments:  - "--providers.kubernetesingress.ingressclass=traefik"  - "--ping"  providers:  kubernetesCRD:  enabled: true  ports:  web:  port: 8000  expose: true  exposedPort: 80  protocol: TCP  nodePort: 32080  websecure:  port: 8443  expose: true  exposedPort: 443  protocol: TCP  nodePort: 32443  service:  enabled: true  type: NodePort |
| --- |

Note: we are deploying both the websecure and web ports for traefik, but later we disable the web access, and continue with only websecure. Also traefik is exposed here as NodePort which will communicate only with the load balancer, which in turn is looked up by the external DNS. All other services will be deployed to the cluster as ClusterIP with traefik annotations.

We then deploy traefik via:

| kubectl create ns tlsoff  helm upgrade --install traefik traefik/traefik --values helm/traefik/values.yaml -n tlsoff |
| --- |

Our next step was to obtain a static global IP address. That IP address was used to expose our stack to the internet through Google Load Balancer. The following command can be executed to get the IP address:

| gcloud compute addresses create tlsoff-ip --global |
| --- |

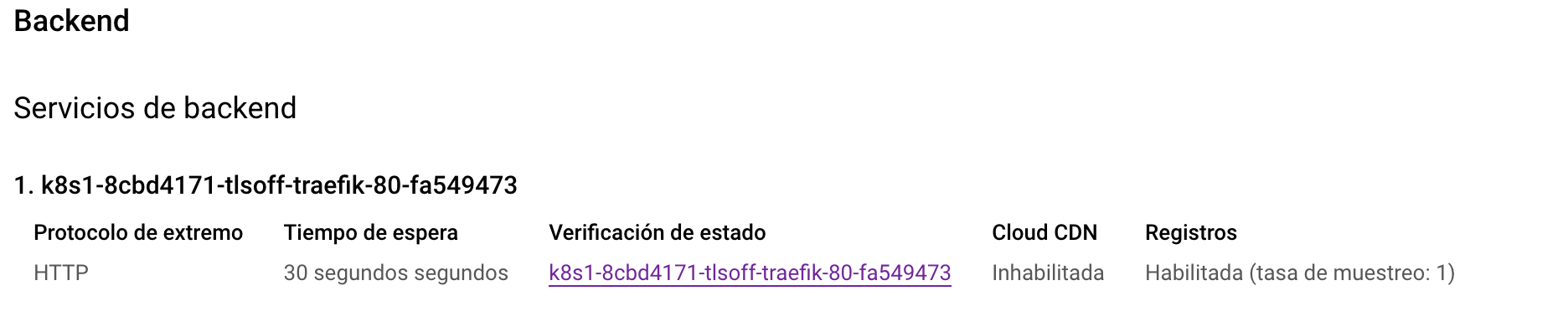
Now that we have the IP address, we can move forward and create a ingress for traefik

| apiVersion: networking.k8s.io/v1  kind: Ingress  metadata:  name: lb-ingress  namespace: lampsecure  annotations:  kubernetes.io/ingress.global-static-ip-name: "tlsoff-ip"  spec:  defaultBackend:  service:  name: traefik  port:  number: 80 |
| --- |

That Ingress resource created a defaultBackend and redirects everything to Traefik Proxy running on the specified port. As part of this apply to the cluster, the static IP is synched with a HTTP(S) load balancer automatically created by GCP. When the load balancer is up and running we create the health-check to the traefik application to ensure the load balancer is functioning and stable.

We then update the healthcheck to point to the /healthcheck path:

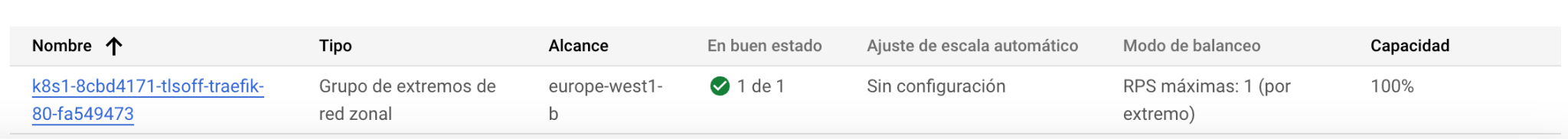
This we find by accessing the LB created above via GCP and checking the backend configuration:



We then update the healthcheck path itself to that of traefik:

| gcloud compute health-checks update http \  --request-path "/healthcheck" k8s1-8cbd4171-tlsoff-traefik-80-fa549473 |
| --- |

After refreshing the LB config, the health check will be enabled correctly:



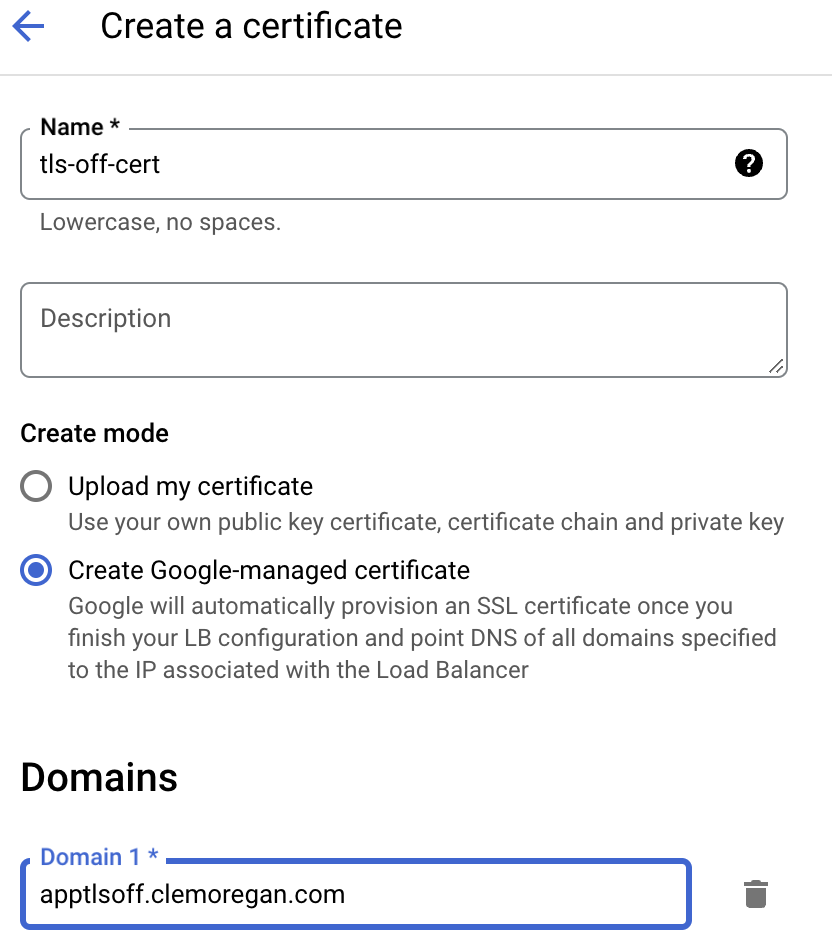
Double check the load balancer is healthy by checking annotations:

| kubectl get ing lb-ingress -n tlsoff -o json |jq '.metadata.annotations' |
| --- |

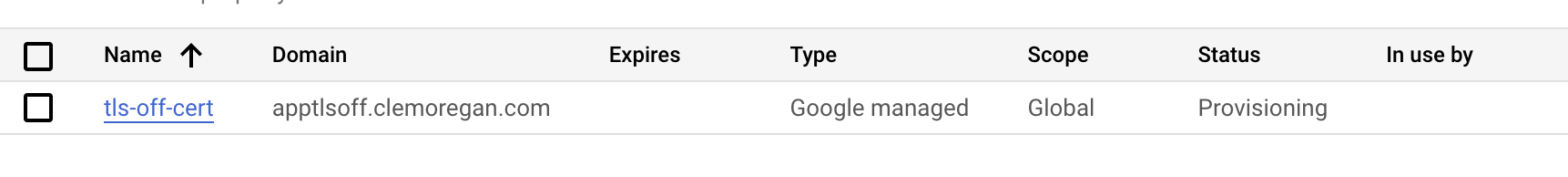
## Create the SSL certificate, domain & HTTPS LB config

Creating the SSL certificate - reference the following documentation: <https://cloud.google.com/load-balancing/docs/ssl-certificates/google-managed-certs#update-dns>

Go to GCP console and Security certificates - classic certificate and “Create SSL certificate”. Note: Can also do this via Terraform.



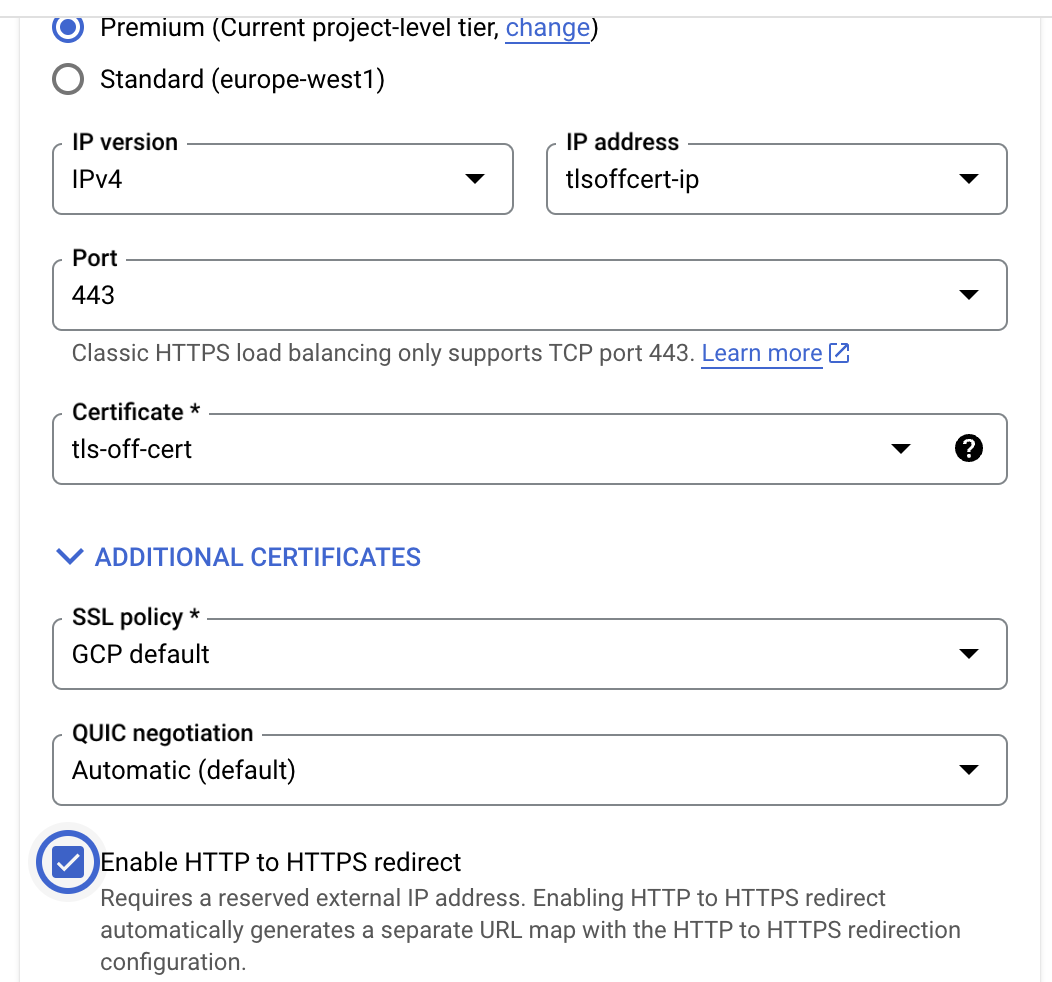
This will set the certificate to status provisioning..



Then update the load balancer with the certificate (as per link reference above).

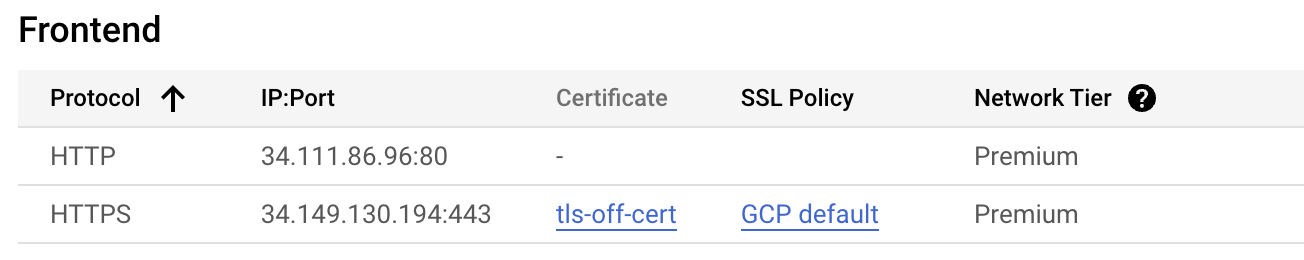
For TLS OFF need a premium network tier & need to relate the frontend configuration with the static IP (can then enable TLS offloading checkbox). Note as the load balancer already has used the static IP we created we can’t switch over to this IP without creating a conflict with HTTP and HTTPS front end configs, so we create a new static IP that we will use in the HTTPS FE configuration in the LB.

| gcloud compute addresses create tlsoffcert-ip --global |
| --- |



Save and update LB config.

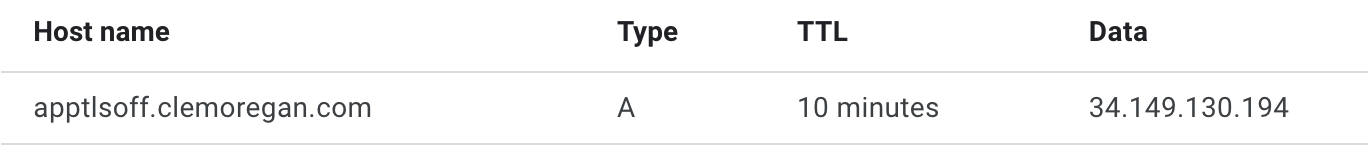
This creates both a HTTP and HTTPS endpoint for us on the LB config.



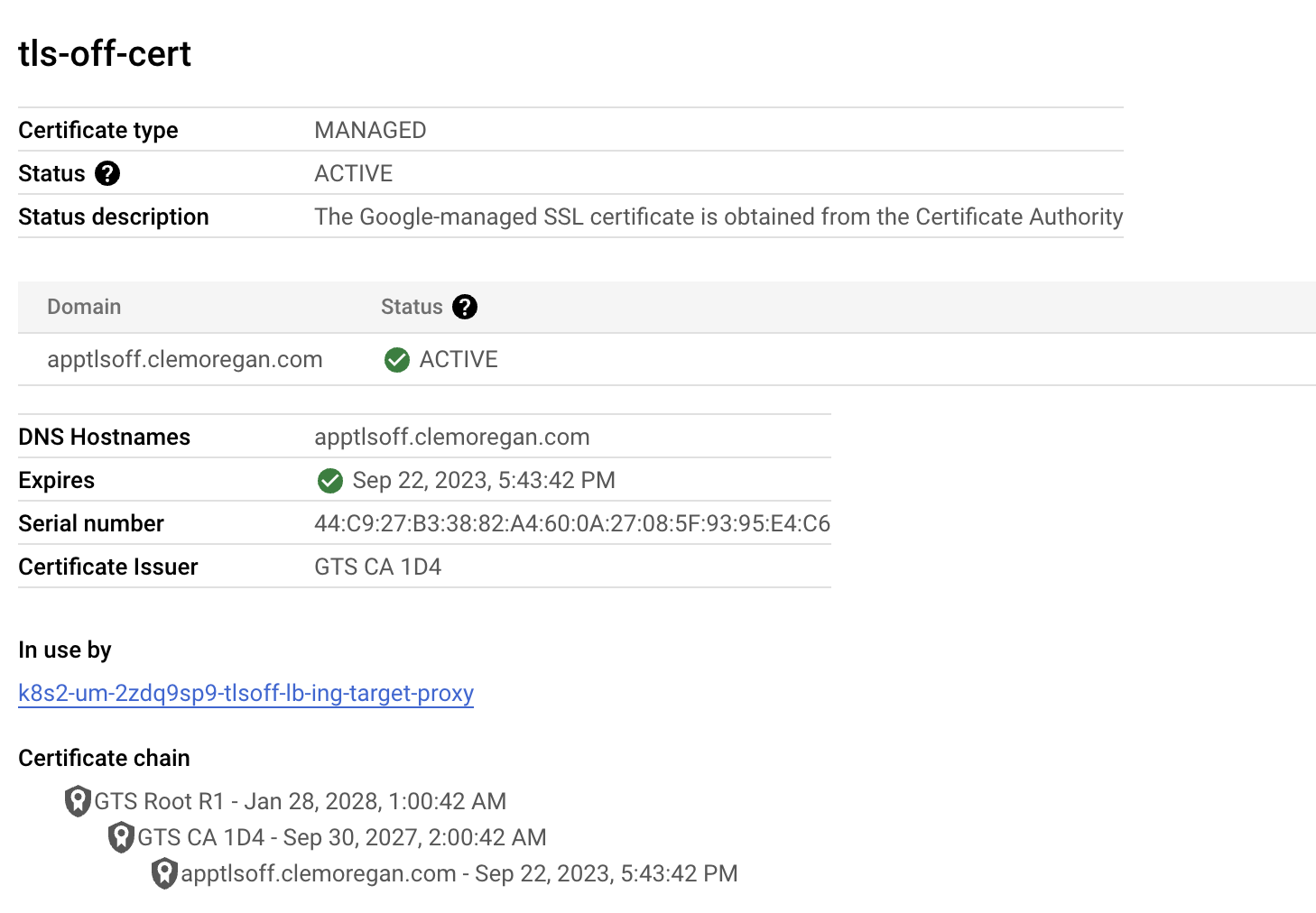
\* We can delete the HTTP one later for security, but no problem technically with it. The backend configuration so still be health as this hasn’t changed.

At this point, your Google-managed certificate status might still be PROVISIONING. Google Cloud is working with the Certificate Authority to issue the certificate

Update google domain records to point to HTTPS LB IP.



After 10 minutes or so on refreshing the certificate, the certificate should now become active:



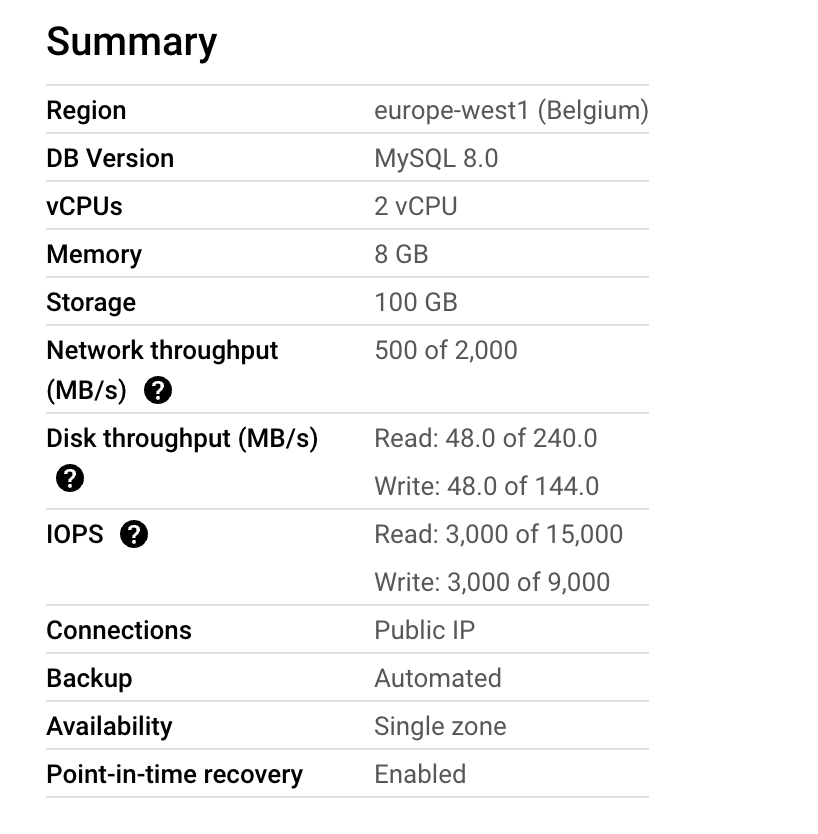
We can then check the healthcheck & dashboard ingress we had deployed previously:

* https://traefik.clemoregan.com/healthcheck/
* https://traefik.clemoregan.com/dashboard/

### DB setup

Deploy a managed instance DB - MySQL. Again take platform provided service, so doesn’t impact cluster performance. Choosing same region as cluster for performance optimized. Following is a dev instance.

Credentials: tfm/tfm2023



In DB connections - add all access to DB (from K8s nodes & from public network).

Create a DB & table:

| CREATE TABLE subjects (  SubjectID int,  Title varchar(255)  ); |
| --- |

Insert data into table:

| INSERT INTO subjects (SubjectID, Title) VALUES (1, 'Secure Architecture');  INSERT INTO subjects (SubjectID, Title) VALUES (2, 'Cyber Attack');  INSERT INTO subjects (SubjectID, Title) VALUES (3, 'Cyber Defence');  INSERT INTO subjects (SubjectID, Title) VALUES (4, 'Data Protection'); |
| --- |

## Deploying the secure application

We then deploy the secure application via helm.

We have included all files used in a separate zip, but for illustration purposes will include the values.yaml file here. The full helm chart can be found in the assignment zip under ./fa\_config/fa\_secureapp/helm/php/

| replicaCount: 3  image:  repository: gcr.io/tfm-jun18/subjects-off  #pullPolicy: Always  serviceAccount:  create: true  annotations: {}  name: "phpsec"  podAnnotations: {}  podSecurityContext: {}  #fsGroup: 1000  securityContext: {}  # capabilities:  # drop:  # - ALL  # readOnlyRootFilesystem: true  #runAsNonRoot: true  #runAsUser: 1000  service:  type: ClusterIP  port: 80  ingress:  enabled: true  annotations:  kubernetes.io/ingress.class: traefik  #traefik.ingress.kubernetes.io/router.tls: "true"  traefik.ingress.kubernetes.io/router.entrypoints: web  hosts:  - host: tlsoff.clemoregan.com  paths:  - path: /  pathType: Prefix  resources: {}  limits:  cpu: 100m  memory: 128Mi  requests:  cpu: 100m  memory: 128Mi  autoscaling:  enabled: true  minReplicas: 2  maxReplicas: 10  targetCPUUtilizationPercentage: 80  # targetMemoryUtilizationPercentage: 80  memorylimit: 80Mi  nodeSelector: {}  tolerations: []  affinity: {}  # Following section will be filled by secret during template creation  db:  dbpass: |
| --- |

Important points to point out from secure application deployment yaml:

* All kubernetes components for the application (deployment, service, RBAC, secrets etc) are deployed via the helm command
* Application is based on a similar type application as with the basic architecture
* Helm command to launch the application is:

| helm install subjects helm/subjects -n tlsoff --set dbpass=<dbpass> |
| --- |

* Secrets for the application are currently passed in via deployment variables, so no sensitive information is stored in the application (php application has been modified to reflect this)
* We create a 3 pod replica count by default for failover and availability, and also have configured a horizontal pod autoscaler with the deployment using autoscaling/v2.
* Application is exposed as ClusterIP service and necessary traefik annotations in place, to expose the application to traefik ingress controller.
* Resource limits have been set up to ensure the pods do not consume more resources if exposed to an attack.

## 

# TLS Full Setup

First try the managed cert setup:

1. Follow: <https://cloud.google.com/kubernetes-engine/docs/how-to/managed-certs#console>
   1. gcloud compute addresses create tftst --project=tls-terraform --global (did this via the GCP console)
   2. kubectl apply -f managed-cert.yaml (DNS entries to cert will be stored here)
   3. kubectl apply -f mc-service.yaml
   4. kubectl apply -f managed-cert-ingress.yaml
2. Check status: “kubectl get ingress” - If ADDRESS field is not filled, give it time.. May have to wait a bit for ingress to register (check events to make sure all ok)
3. Then create the DNS domains specified in cert above in google domain, and put in IP of create load balancer!
4. Have to then wait again (up to 60m), in meantime check:
   1. “watch kubectl describe managedcertificate managed-cert”
   2. Also watch events
   3. After that time cert should go to ACTIVE
5. You can then verify cert is secure by going to <https://domain/> in browser

Deleting (if cluster created by TF):

* kubectl delete -f managed-cert.yaml
* gcloud compute addresses delete tftst --project=tls-terraform –global
* terraform destroy
* After delete certificates (certificate manager)
* Delete LB that was created as well!

## Integrating with traefik

* Ensure create static IP and create cert 1a & traefikcert.yaml created (will be created in traefik namespace)
* Install traefik via helm (using the values file above first?) into traefik namesapce
* Apply traefik-ingress.yaml
  + (on port 443?? but kubectl ingress on port 80?)
  + So certificate not auto recognized in LB
  + Change LB config - either via console or some config file way
* Apply traefik-ingressroute.yaml
* May need to update LB background as well - not healthy! But can do that after I think!

# 

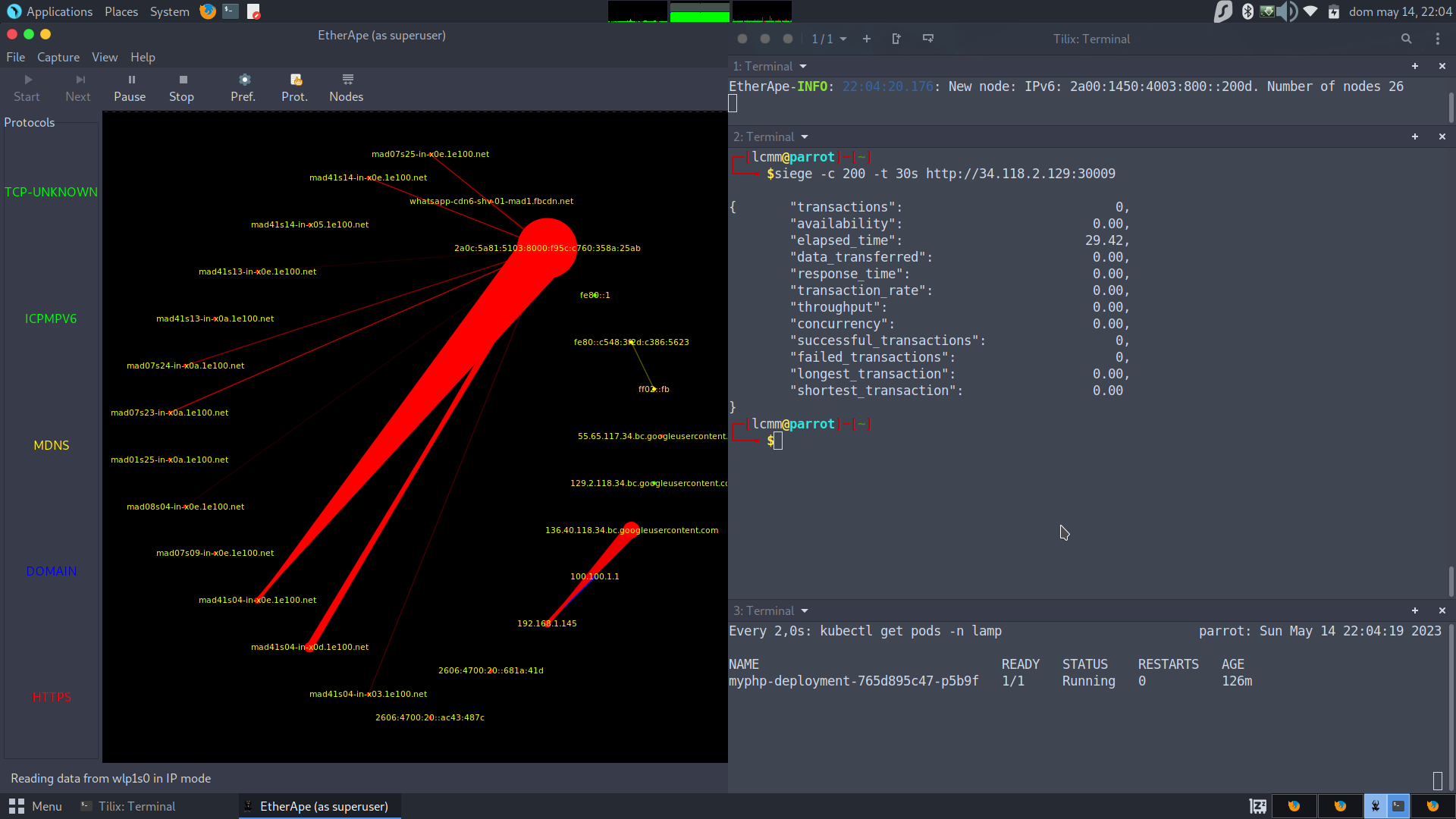
# Stress Testing

### Apache bench

ab -n 1000 -c 5 <https://whoami.clemoregan.com/>

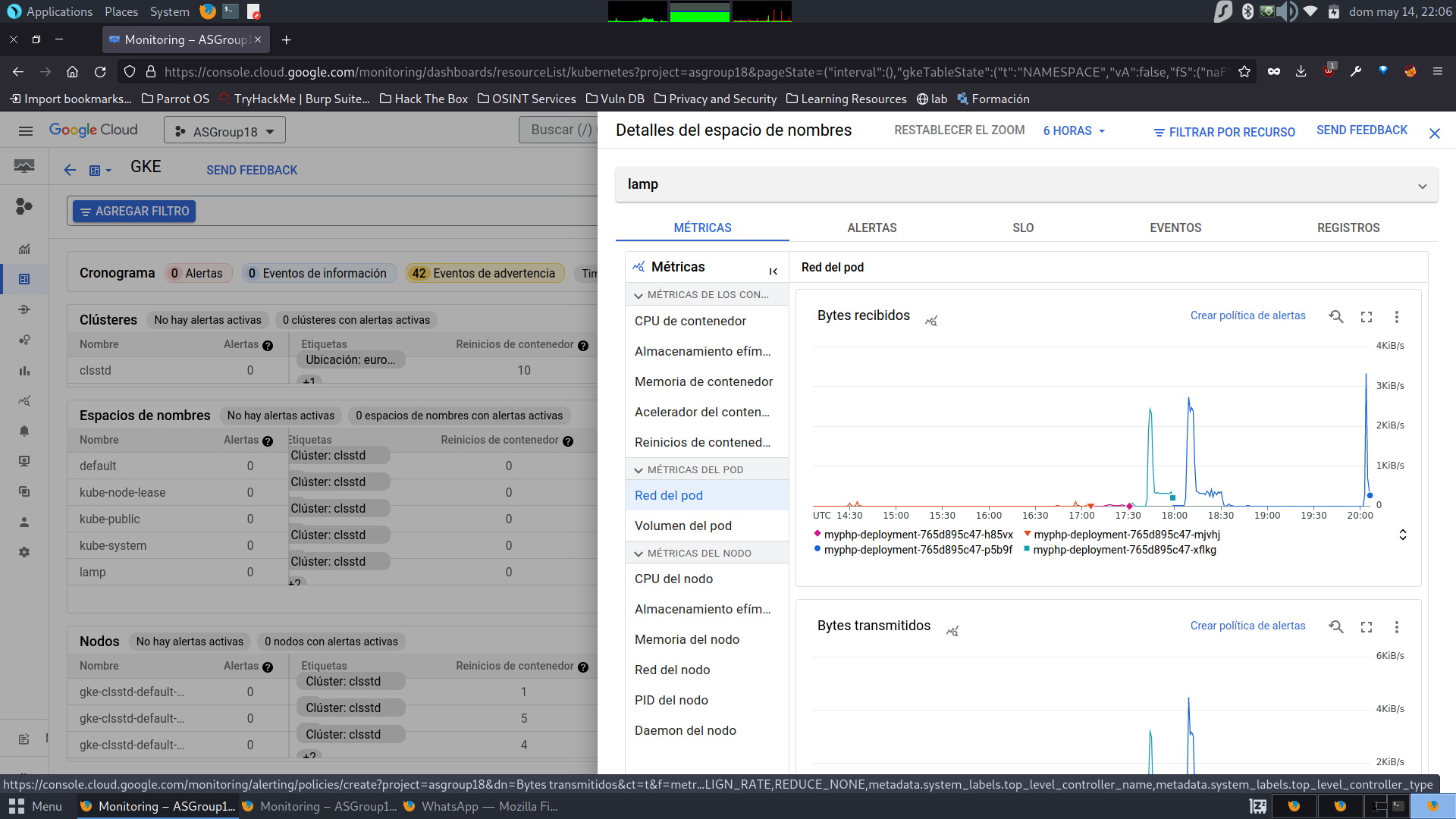
### Siege

For stress testing our system we used Siege, an open source tool that allows multi-threaded http load testing. It allows one to hit a web server with a configurable amount of concurrent simulated users allowing us to measure the duress of the architecture.

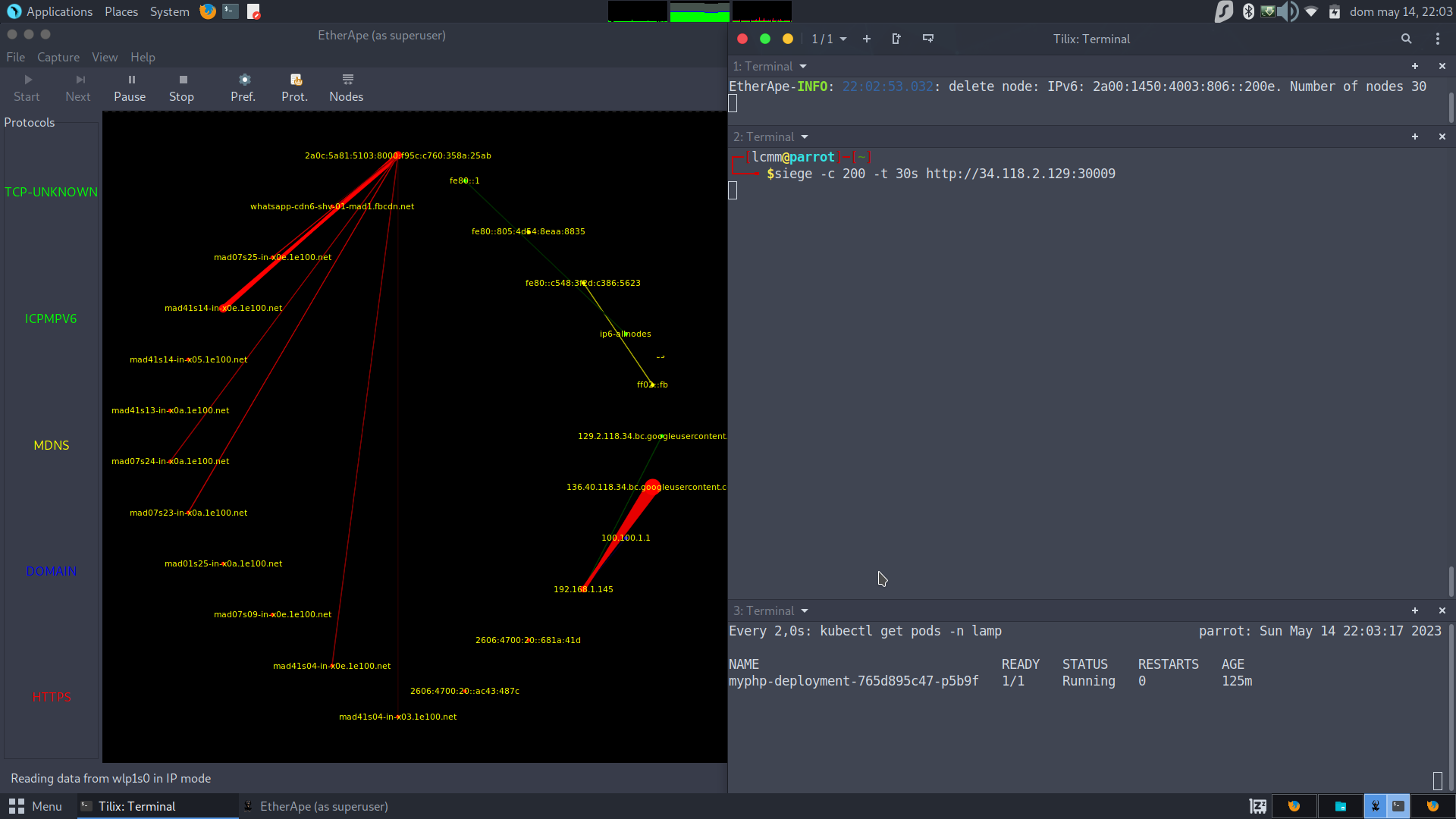


Siege shines against other tools like Slowrys, GoldenEye and even services like Loader.io due to how simple it is to run tests without complex configurations on the server or the attacking machine.

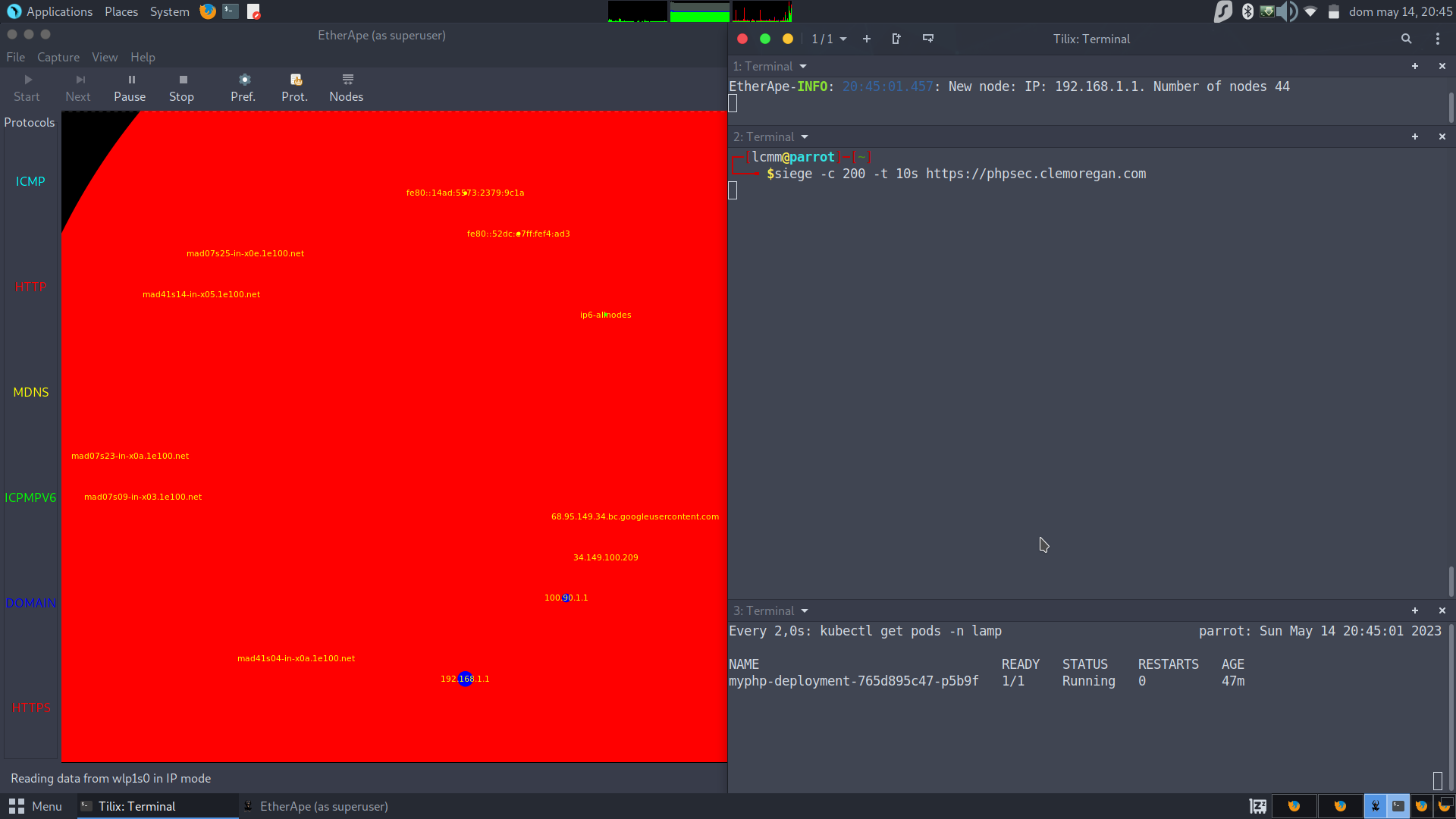
We also used the Monitoring Tools from Google Cloud we can see the spikes representing the unnatural amount of requests received by the server.



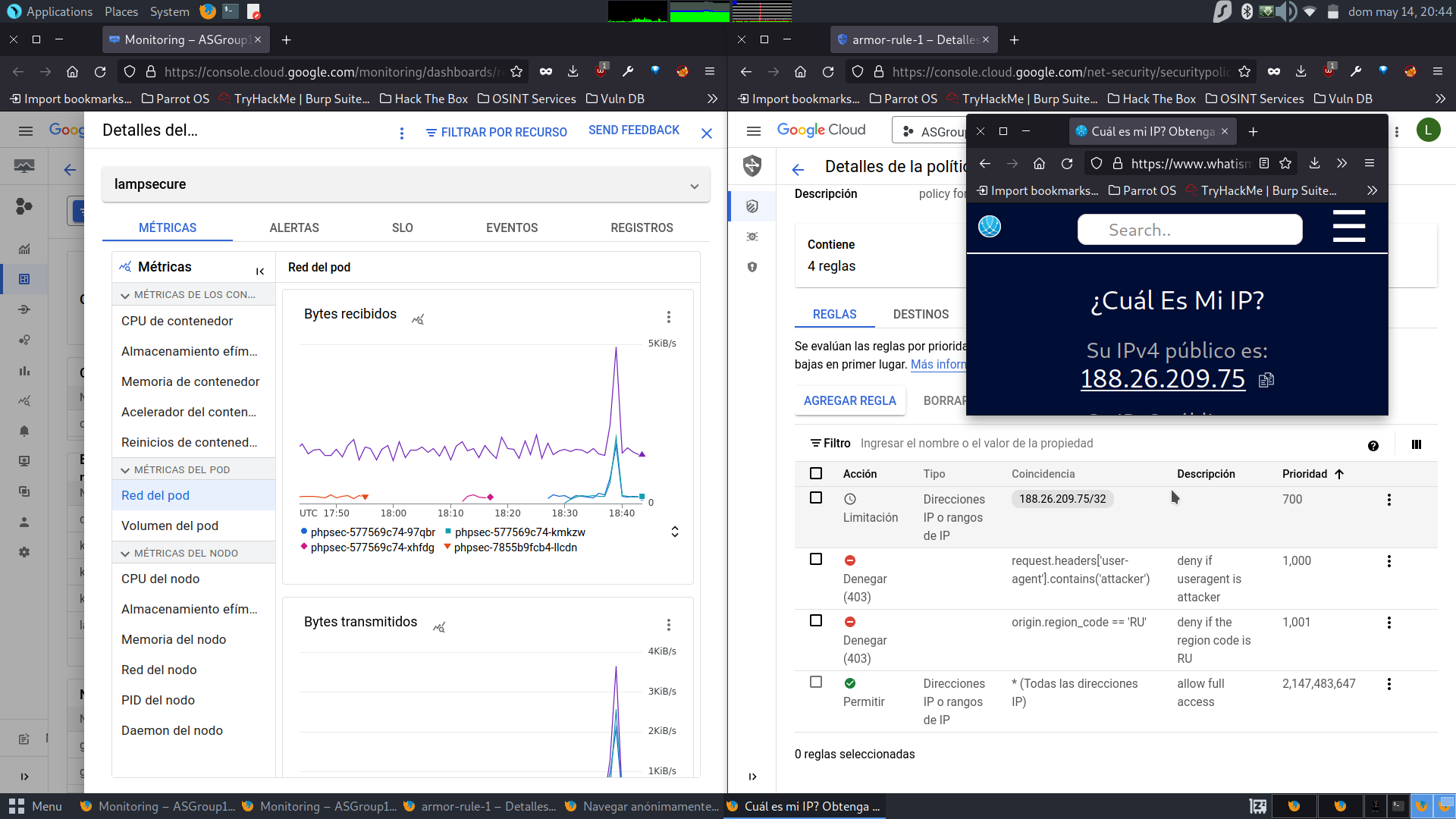
First we attacked the basic architecture to have a benchmark on how well our secure application works. We used Siege to generate an abnormal amount of traffic successfully bringing down the service that was running without protection. We attacked with 200 concurrent simultaneous users in a 1 minute timeframe.



After that we did the same attack on our secure application that is running with countermeasures.



The second attack didn’t perform as well as the first instance. In this case the attacker IP address was picked by a Cloud Armor rule and was blocked for enough time to survive the overflow and continuing to provide the service to other clients.



## 

## 

With this we concluded that we successfully improved our first architecture and were able to protect it against layer 7 DoS attacks.

## Monitoring

For monitoring start off by enabling managed service for prometheus for GCP.

Although we could deploy within the cluster prometheus and grafana, using the managed service could be more reliable as it does not interfere with cluster performance.

Clicking on cluster anf viewing options we can see it is enabled for the cluster.

