Prometheus troubleshooting: Where is my newly configured data?

Intro

A few days Terry (@tspotts) pinged me about a case where his recently configured **Prometheus targets doesn't work**. This guide aims to be a **supporting document** on how I **investigated** and resolved this case. Even though here I describe a specific investigation, the **method** and the **tools** can be used to **investigate** other **Prometheus** related problems as well.

Symptom

The configured targets reside in **prometheus-vitess** in the **syseng-gke-prod** cluster, under the **vitess namespace**.

First, he pointed me to a **block** of configuration , which **works**:

```
    {
```

```
        name: '[redacted]vitess-dev',
        team: 'dbre',
        zones: ['us-central1-a', 'us-central1-c', 'us-central1-b', 'us-central1-f'],
        hosts: [''],
        suffix: 'legacy',
        gceLabels: scrapeConfig.gceInstanceChefRoles +
                   [
                     {
                       source_labels: ['chef_role'],
                       regex: 'GCP_devdbpaymentsshard',
                       action: 'drop',
                     },
                     {
                       source_labels: ['chef_role'],
                       regex: 'Vitess_VtGate',
                       action: 'keep',
                     },
                     {
                       target_label: 'dev',
                     },
                   ],
```

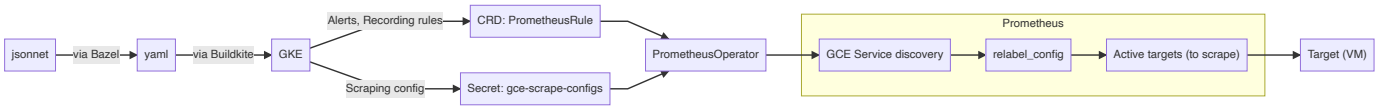
and then Terry showed me the new block that should **work**, but it **doesn't**:

```
{
        name: '[redacted]vitess-dev',
        team: 'dbre',
        zones: ['us-central1-a', 'us-central1-c', 'us-central1-b', 'us-central1-f'],
        hosts: [''],
        suffix: 'legacy-paymentsshard',
        gceLabels: scrapeConfig.gceInstanceChefRoles +
                   [
                     {
                       source_labels: ['chef_role'],
                       regex: 'Vitess_VtGate',
                       action: 'drop',
                     },
                     {
                       source_labels: ['chef_role'],
                       target_label: 'scrape_config',
                       regex: 'GCP_devdbpaymentsshard',
                       action: 'keep',
                     },
                     {
                       target_label: 'dev',
                     },
                   ],
```

Investigation

Architecture

First, let me present the process of how we configure Prometheus, starting from the jsonnet code till it scrapes the target.



At **each** of these **stages** there can be a **potential problem** why the config doesn't get rendered, or it doesn't get rendered correctly. In my **investigation**, since I knew nothing about this prometheus instance, I went from **beginning** till the **end**, wing to wing review.

Bazel

I'm not going to go into jsonnet details beause that'd be a document of its own, so just **jumping** one step **ahead**, but the important part is that we **write** the configuration code in **jsonnet** which is **generating** a **json/yaml** output. We can start **verifying** the output, and **if** we see our **code** generated there, we can say we're (most likely) **happy**.

To start my investigation, I was looking for a **unique string** in the malfunctioning code block, so I can **search** for that in each of these **stages**. First I considered `Vitess_VtGate` but that wasn't unique so I **picked** another one, the **suffix parameter** `legacy-paymentsshard`. The jsonnet code suppose to generate this block of code for each defined exporters:

```
      exporters(zone):: [
        scrapeConfig.nodeExporter(self, zone, { name: 'percona-exporter', port: 9104
}),
        scrapeConfig.nodeExporter(self, zone, { name: 'node-exporter', port: 9100 }),
        scrapeConfig.nodeExporter(self, zone, { name: 'process-exporter', port: 9256
}),
        scrapeConfig.blackboxExporter(self, zone, { name: 'ssh', module: 'ssh_banner',
port: 22 }),
      ],
```

First, I **checked** the available **targets** for Bazel:

```
aspitzer@aspitzer prometheus-vitess % bazelisk query ...
Starting local Bazel server and connecting to it...
//clusters/syseng-gke-prod/prometheus-vitess:grafana-access-token
...
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess-json
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess-validate
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess.apply
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess.create
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess.delete
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess.diff
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess.replace
//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess.resolve
...
```

There are many **targets** in the list we can't render, but the ones **with .apply**, we surely **can**, that'll be a k8s macro in the BUILD.bazel file:

```
k8s_syseng_prod(
    name = "prometheus-vitess",
    template = ":prometheus-vitess-json",
)
```

I just want to **see** the **unique string** I picked **shows up** in the config, that's my **first step**. That means the code block **rendered**, not sure if it's 100% correct but that's **good enough** for now.

We can **render** the **json** two ways, either with the base target name `//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess` , in which case **Bazel** will send the **rendered** output to **standard output** (I usually prefer this, I can pipe it into a less) or we can use the one with `-json` suffix, `//clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess-json` which will generate the json output in a file.

So, I **ran**:

```
aspitzer@aspitzer prometheus-vitess % bazelisk run //clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess | less
```

and I was **searching** for my unique string, `legacy-paymentsshard`. I **found** many PrometheusRule CRDs, but **nothing** that relates to **scraping**, for example:

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    prometheus: vitess
    role: vitess-rules
```

```
  name: prometheus-rules-scrape-legacy-paymentsshard-[redacted]vitess-dev
  namespace: vitess
spec:
  groups:
  - name: Scrape Alerts - legacy-paymentsshard-[redacted]vitess-dev
    rules:
    - alert: TextfileScrapingFailing
      annotations:
...
```

I was happy to see my unique string as part of PrometheusRules, but that **wasn't enough** as I was investigating a **scraping** issue. The PrometheusRule can describe **alerts** and **recording rules**, but not scraping rules, so I had to **look further**. At the end of the **json output**, I found a very long **secret** manifest as well:

```
kind: Secret
metadata:
  name: gce-scrape-configs
  namespace: vitess
```

This looks **promising**, so I had to **decode** the data part to see if my **unique string** shows up there. So, I ran

```
$ bazelisk run //clusters/syseng-gke-prod/prometheus-vitess:prometheus-vitess | grep
gce-scrape-configs.yaml: | sed 's/^.*: //g' | base64 -d | jq
```

This **decoded** my secret data, and piped it into a **jq**. Now, I **saw** these:

```
...
  "job_name": "ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a",
    "metrics_path": "/probe",
    "params": {
      "module": [
        "ssh_banner"
      ]
    },
    "relabel_configs": [
      {
        "replacement": "${1}:22",
        "source_labels": [
          "__meta_gce_private_ip"
        ],
        "target_label": "__param_target"
      },
      {
        "replacement": "blackbox-exporter.exporters.svc:9115",
        "target_label": "__address__"
```

```
      },
      {
        "source_labels": [
          "__meta_gce_instance_name"
...
```

OK, I can **confirm** that the **scraping code** we defined in the jsonnet **shows up** in the scraping configuration for Prometheus.

Prometheus Operator

Next, I have to **confirm** that the config we **generate** in the secret called `gce-scrape-configs` is **picked** up by the **Prometheus Operator**. The way **prometheus operator** works is that it **defines** certain **CRDs** (Custom Resource Definitions, objects in Kubernetes that are not part of the core Kubernetes, they're 3rd party extensions) that we can **apply** into the cluster, and the **Operator** is continously listening on the API endpoint **looking** for these **CRDs**. If a **CRD** is applied that **belongs** to the **Prometheus Operator**, it'll **do** whatever that CRD is meant to do. One of the reasons historically Operators were written to support apps in Kubernetes that were not written natively for Kubernetes, for example they expect a config file (which are almost all apps before Kubernetes), while everything in Kubernetes is configured via objects, manifests (that are stored in etcd backend in the control plane at Google). So Operators became this bridge where we can create Kubernetes manifests which then are picked up by the Operator and translate it to the app's config format whatever it requires. Another reason was to simplify app management and upgrades in Kubernetes.

The most **fundamental** Prometheus Operator CRD is the `prometheus` CRD itself, which **deploys** a **Prometheus pod** into the namespace. Another important one is what we just mentioned, the **PrometheusRule**, which is a small piece of configuration for the Operator. The Operator picks up all the PrometheusRules and melts them into a single config file for the AlertManager.

So, we'll check the `prometheus` CRD whether it's configured to use the `gce-scrape-configs` secret, as a scraping configuration:

```
aspitzer@aspitzer prometheus-vitess % kubectl get prometheus -n vitess vitess -o yaml |
grep -A3 -B3 gce-scrape-configs
metadata:
  annotations:
...
spec:
  additionalScrapeConfigs:
    key: gce-scrape-configs.yaml
    name: gce-scrape-configs
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
...
```

Yes, the **secret** we found, which **includes** our **config** is **configured** via the spec.additionalScrapeConfigs **setting**!

So far, it looks like everything is fine. Let's **continue**.

Prometheus

To move on to **investigating Prometheus** itself, I needed a higher layer/Prometheus layer unique string to look for, so I picked a specific **job** to **investigate**. So far, I was looking for `legacy-paymentsshard`, but since that **pattern** will show up on **more jobs** (if you remeber we generate that block of code for all the exporters defined, and in that block we have at least 5 exporters defined) in Prometheus **I'll look** for `ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a` as I saw in the scrape config.

To confirm everything we've seen so far, I wanted to see the **configuration** in **Prometheus** itself. To do that, I had to find the **Prometheus' URL**, so I went to the vitess namespace and queried all the ingresses:

```
aspitzer@aspitzer prometheus-vitess % kubectl get ingress -n vitess
NAME                    CLASS    HOSTS
     ADDRESS        PORTS     AGE
prometheus-ingress      <none>   prometheus.vitess.[redacted]syseng-gke-prod.[redacted]com
      10.253.8.22   80, 443   2y14d
vitess-pod-0-ingress    <none>   prometheus-pod-0.vitess.[redacted]syseng-gke-
prod.[redacted]com   10.253.8.22   80, 443   2y14d
vitess-pod-1-ingress    <none>   prometheus-pod-1.vitess.[redacted]syseng-gke-
prod.[redacted]com   10.253.8.22   80, 443   2y14d
```

So, the **address** is `https://prometheus.vitess.[redacted]syseng-gke-prod.[redacted]com`.

I went to Status -> Config in Prometheus, where I **searched** for the **job** under investigation `ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a`, it was there:

```
...
- job_name: ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a
  honor_timestamps: true
  params:
    module:
    - ssh_banner
  scrape_interval: 15s
  scrape_timeout: 10s
  metrics_path: /probe
  scheme: http
  follow_redirects: true
  enable_http2: true
```

```
  relabel_configs:
  - source_labels: [__meta_gce_private_ip]
    separator: ;
    regex: (.*)
    target_label: __param_target
    replacement: ${1}:22
...
```

Great. Let's **review** the **main sections** of this **configuration**:

```
- job_name: ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a
  honor_timestamps: true
  params:
    module:
    - ssh_banner
  scrape_interval: 15s
  scrape_timeout: 10s
  metrics_path: /probe
  scheme: http
  follow_redirects: true
  enable_http2: true
  relabel_configs:
  - source_labels: [__meta_gce_private_ip]
    separator: ;
    regex: (.*)
    target_label: __param_target
    replacement: ${1}:22
    action: replace
  - separator: ;
    regex: (.*)
    target_label: __address__
    replacement: blackbox-exporter.exporters.svc:9115
    action: replace
  ...
  - separator: ;
    regex: (.*)
    target_label: dev
    replacement: $1
    action: replace
  gce_sd_configs:
  - project: [redacted]vitess-dev
    zone: us-central1-a
    filter: name:*
    refresh_interval: 1m
    port: 80
    tag_separator: ','
```

The **first** part contains the **basics**, scrape interval, scheme, job name. The **second** part is **relabel_configs**, and the **third** part is the **service discovery** in GCE (gce_sd_configs).

Prometheus will

- first try to **find** the **targets** via **service discovery**, then
- it'll **apply** the **relabel_configs** section, and then
- it'll **scrape** the targets

The `source_labels` in relabel_configs is **referring** to are populated by the **service discovery** module, and it contains various metadata about the target (IP, network details, zone, etc.). It's important to note that there are multiple relablings in Prometheus, this relabel_configs at the scraping section specifically happens right after service discovery and before target scraping.

Relabelings

What's the **purpose** of relabeling? There can be many, usually **filtering** out (drop) **targets**, **metrics** we don't need. In this very specific job, we use it for **something more**.

Exporters

Prometheus' **scraping** is very **simple**. Every Nth second (scrape_interval) it **reaches out** to the **target's IP** and **port** via **HTTP** and expects a newline separated (human readable) **text** with the **metrics** and their **value**, hasmarked lines are **help** and also defines the metrics' **type** (gauge, counter, etc).

For **example**:

```
...
probe_duration_seconds 0.019504261
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful to detect if
the IP address changes.
# TYPE probe_ip_addr_hash gauge
...
```

If we want to **scrape** with Prometheus a **non-standard** Prometheus endpoint, we'll have to **use** an **exporter** to "translate" between **Prometheus** and the **endpoint** we want to monitor. All **exporters** follow this **logic**:

- they're **open** for an incoming **Prometheus HTTP request**
- they **monitor** the **target** the appropriate way
- **return** a Prometheus **compatible result**

There are hundreds of already written exporters and the **difference** between them is the **middle** part, **how** they **monitor** their **target**, that's always target specific, this way **Prometheus** doesn't have to **know** (or care) about it. One notable official exporter is the **Blackbox exporter**, which allows us to **monitor** (scrape) **TCP endpoints**, like in this case, an **SSH endpoint**.

We keep **our exporters** in the `exporters` **namespace** in **syseng-gke-prod**, athough some exporters must run at/near the target itself (for example, node exporter runs on each node, exporting OS metrics).

Blackbox exporter

Here we **use** a **blackbox exporte**r, which is a simple **proxy** that anyone can call via **HTTP**, via the **query string** we tell the blackbox exporter what the TCP endpoint we want to monitor, the blackbox exporter **checks** it for us, and **returns** the outcome in a **Prometheus compatible** format. We can easily **test** this via a **curl** (I ran this curl from a test pod, inside GKE, but you can test it also from outside, from your laptop):

```
root@testpod-sendai:/# curl "http://blackbox-exporter.exporters.svc:9115/probe?
module=ssh_banner&target=10.248.7.138%3A22"
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in
seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 7.6e-06
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.019504261
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful to detect if
the IP address changes.
# TYPE probe_ip_addr_hash gauge
probe_ip_addr_hash 2.809066758e+09
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

As you can see in my example the **query parameters** contain:

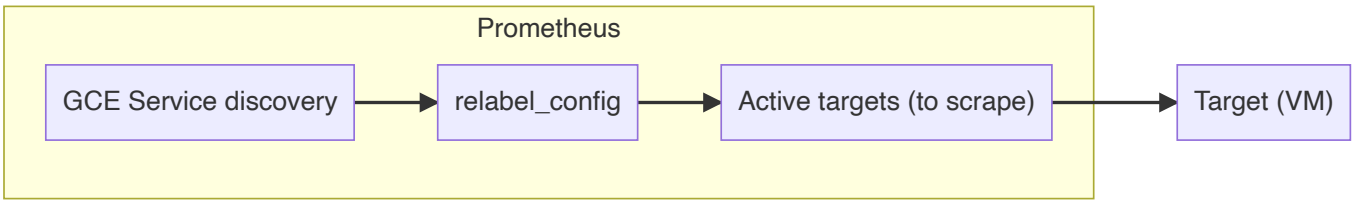
- module=ssh_banner
  - indicating that we expect a **valid SSH banner** string, you can see this parameter also in the target job configuration)
- target=10.248.7.138%3A22
  - tells the exporter which **port + IP** to **connect** to. It's urlencoded, where `%3A` stands for `:` so this is actually `10.248.7.138:22`

and the **result** of the connect attempt is

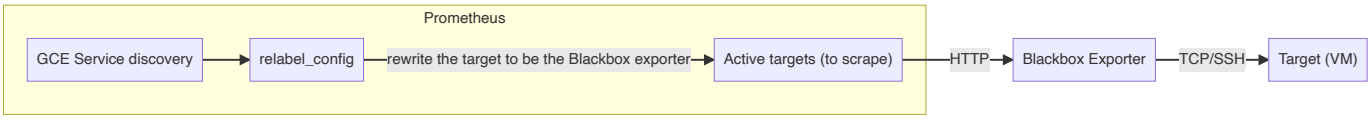
```
probe_success 1
```

meaning the target **responded** with a **valid SSH banner**.

So, if you remember the **standard scraping** process flowchart before:



Will **become** this:



Back to our Prometheus investigation

In the job we picked for investigation `ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a` we want to **test** whether a **hosts' SSH** port is **open** (and valid) or not. So we **don't scrape** the endpoint **directly** (it wouldn't make sense, Prometheus sends an HTTP request and expects a Prometheus format text, an SSH daemon won't give it to us), but we're **using relabeling** to **rewrite**

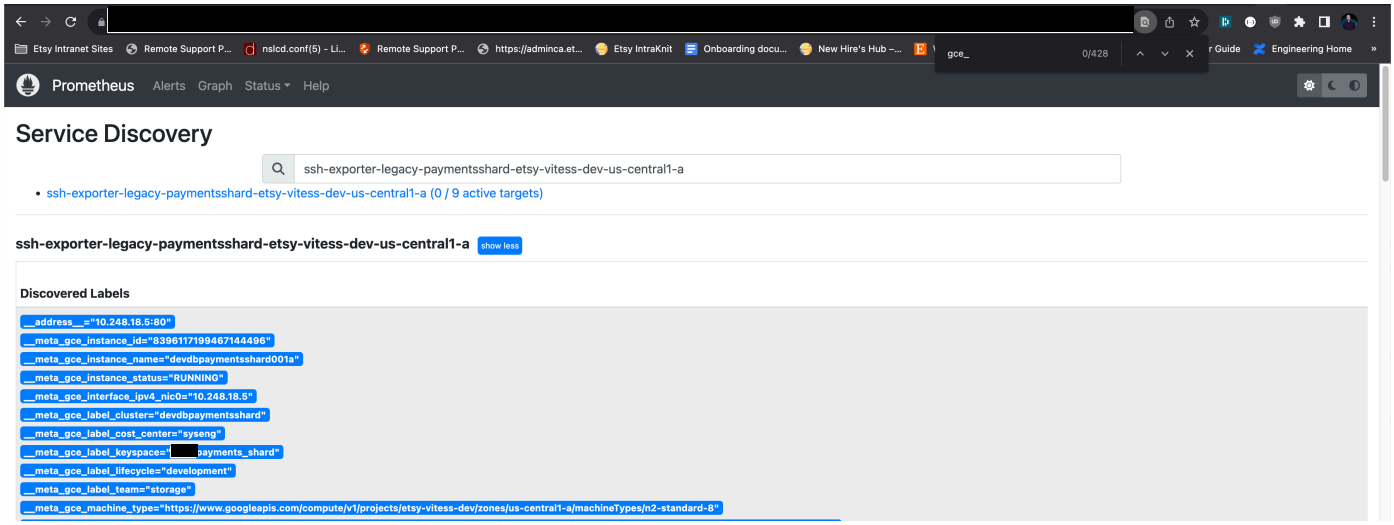
- **what** prometheus should **scrape**

    - the **blackbox exporter**
- and **what query parameters** it should use
    - the **original IP + port** discovered by the GCE service discovery module

Back to our investigation, we **see** the job **configuration** in **Prometheus**, which is **great**. Next is, **service discovery**.

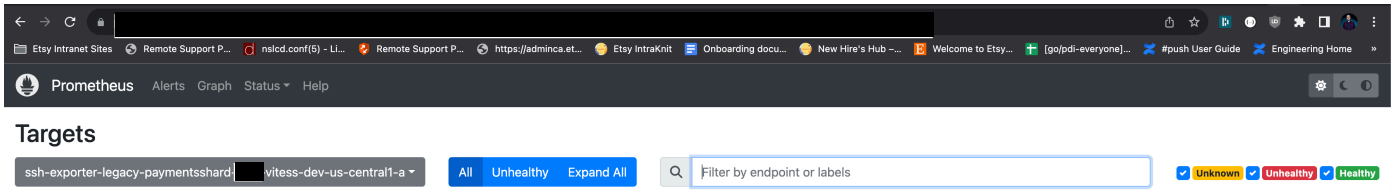
Service Discovery

We can use Status -> Service Discovery at Prometheus to see **what** services were **discovered**. When you go to Status -> Service discovery, you'll see an **input field** to seach for the job we're **investigating**. This is what I got:



As you can see, the **service discovery knows** about our job (we confirmed it's in the config, so that's not surprising), but it has **0/9 active targets**. **What** this means? **Active targets** are **scraped**, so this means that the **service discovery** part **worked**, it found 9 VMs, but it **doesn't scrape** any of them. **Why**?

We can **confirm** this under **Status -> Targets**, where we'll see **no targets** listed:



Correct, it's **empty**.

So, we see that our **process** looks **correct** until the **service discovery** part, as we were able to discover **9 VMs**, but it **fails** to **scrape** them for some reason.

Also here we can see the IP of the first **discovered** VM: **10.248.18.5**

At this point we're pretty **suspicious** of the process **between service discovery** and **scraping**. So far we **verified everything before** service discovery, and after scraping we only have the **blackbox exporter** and the **endpoint** itself.

Testing the Blackbox exporter

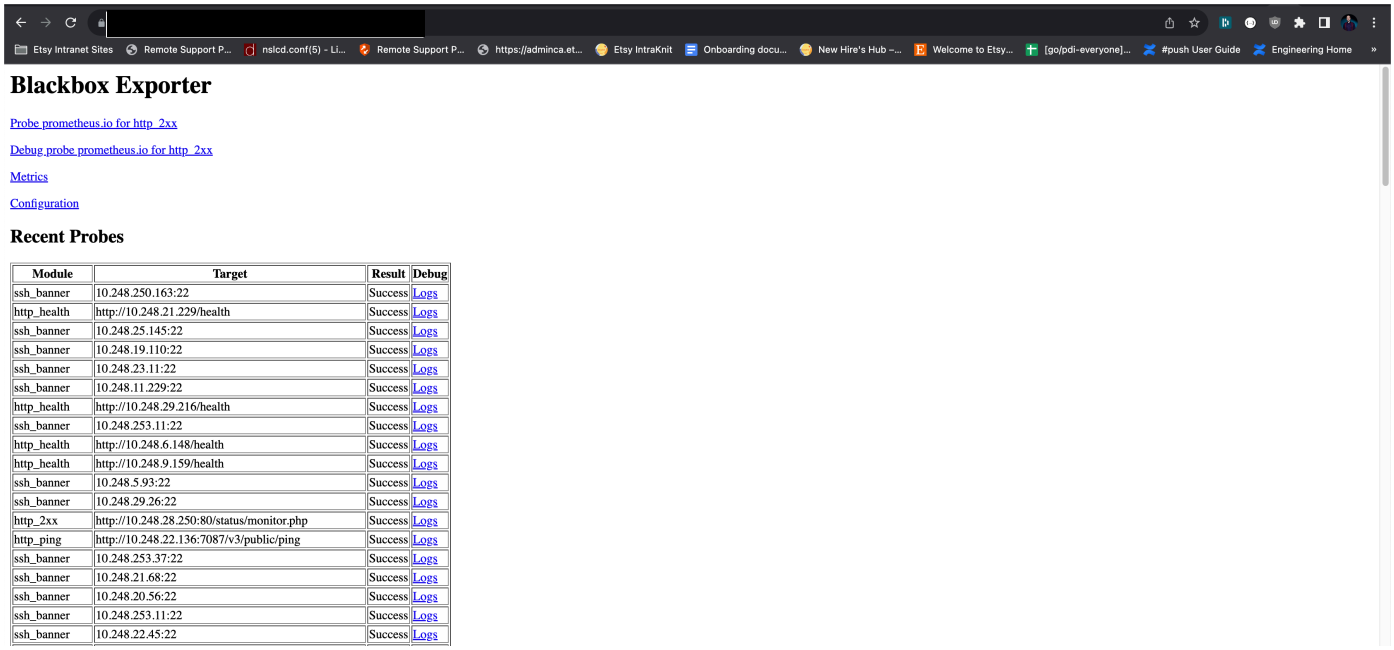
There are **two ways** to text the **Blackbox exporter**, with **Curl** and via the **Admin UI**. The Admin UI is just a basic web interface **showing** all the previous requests and their result. Let's **find** the HTTP address of the blackbox exporter. This was the label rewrite rule, related to the blackbox exporter:

```
  - separator: ;
    regex: (.*)
    target_label: __address__
    replacement: blackbox-exporter.exporters.svc:9115
    action: replace
```

`blackbox-exporter.exporters.svc` this tells us that the **service** is `blackbox-exporter` in the `exporters` **namespace**, so let's look for an ingress there:

```
aspitzer@aspitzer fix_prometheus % kubectl get ingress -n exporters
NAME                          CLASS    HOSTS
        ADDRESS         PORTS     AGE
blackbox-exporter-ingress     <none>   blackbox-exporter.c.[redacted]syseng-gke-
prod.[redacted]com    10.253.8.135   80, 443   3y223d
```

We **found** the admin UI for the **Blackbox exporter**, it's `blackbox-exporter.c.[redacted]syseng-gke-prod.[redacted]com`. Let's **see** what's there:



Blackbox Exporter

Probe prometheus.io for http_2xx

Debug probe prometheus.io for http_2xx

Metrics

Configuration

Recent Probes

Module	Target	Result	Debug
ssh_banner	10.248.250.163:22	Success	Logs
http_health	http://10.248.21.229/health	Success	Logs
ssh_banner	10.248.25.145:22	Success	Logs
ssh_banner	10.248.19.110:22	Success	Logs
ssh_banner	10.248.23.11:22	Success	Logs
ssh_banner	10.248.11.229:22	Success	Logs
http_health	http://10.248.29.216/health	Success	Logs
ssh_banner	10.248.253.11:22	Success	Logs
http_health	http://10.248.6.148/health	Success	Logs
http_health	http://10.248.9.159/health	Success	Logs
ssh_banner	10.248.5.93:22	Success	Logs
ssh_banner	10.248.29.26:22	Success	Logs
http_2xx	http://10.248.28.250:80/status/monitor.php	Success	Logs
http_ping	http://10.248.22.136:7087/v3/public/ping	Success	Logs
ssh_banner	10.248.253.37:22	Success	Logs
ssh_banner	10.248.21.68:22	Success	Logs
ssh_banner	10.248.20.56:22	Success	Logs
ssh_banner	10.248.253.11:22	Success	Logs
ssh_banner	10.248.22.45:22	Success	Logs

If I search for our IP we discovered **10.248.18.5** it's **not there**.

Now, let's **check** this same **IP** with **curl**, via the **blackbox exporter** (as you can see I updated the IP in the query parameter with the IP Prometheus discovered), to **see** if the **service** is actually **reachable**:

```
aspitzer@aspitzer prometheus-vitess % curl "http://blackbox-exporter.c.[redacted]syseng-gke-prod.[redacted]com/probe?module=ssh_banner&target=10.248.18.5%3A22"
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 1.2524e-05
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.008075408
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful to detect if the IP address changes.
# TYPE probe_ip_addr_hash gauge
probe_ip_addr_hash 2.25358095e+09
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

As you can see, it **works**, probe_succcess is 1, the exporter was **able** to **reach** 10.248.18.5:22 and **received** a **valid SSH banner**. With this we **confirmed** that all the **components** are **fine**, we have to focus our **attention** to the process **between** the **service discovery** and **scraping**.

Relabeling

The **only part left** for investigation is the **relabeling** part, as that's the **only part** we have **between** the **service discovery** and the **scraping** (active targets).

Probably the **relabaling** part is the **most cryptic** in this process flow, as there is **no easy way to test it**. We **manipulate** the labels in a way to alter the **behaviour** of Prometheus, but unless we make a syntax error, it won't let us know that something is wrong, it just won't work.

Let's see our **relabeling** configuration, we're going back to **Status -> Configuration**, and **search** for our job `ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a`, let's **review** them **one by one**. The source labels are all coming from the **service discovery** module, you can find them all here. All the **relabeling options** and their explanations are further described in the official documentation.

```
relabel_configs:
- source_labels: [__meta_gce_private_ip]
  separator: ;
  regex: (.*)
  target_label: __param_target
  replacement: ${1}:22
  action: replace
```

We're **taking** the **original label** `__meta_gce_private_ip` (which is the private IP of the VM, received via the service discovery module) and we **rewrite** it to IP:22 (we'll use this to send to the blackbox exporter to reach out to), **store** the result in `__param_target` which wil be **passed** to the **Blackbox exporter** as a **query string**, explained here

```
- separator: ;
  regex: (.*)
  target_label: __address__
  replacement: blackbox-exporter.exporters.svc:9115
  action: replace
```

We're **overwriting** the `__address__` label with the **blackbox exporter's address** which is used by the next stage, the scraping mechanism **what to scrape**. Sort of a hack, but it works.

```
- source_labels: [__meta_gce_instance_name]
  separator: ;
  regex: (.*)
  target_label: instance
  replacement: $1
  action: replace
```

Creating a **new label** called `instance` using the source label `__meta_gce_instance_name`, we **copy it as-is**. We do these copying because we don't need all the source labels, so we copy the ones we need into regular label names (without `__`)

```
- source_labels: [__meta_gce_metadata_instance_template]
  separator: ;
  regex: .*/(.*)
  target_label: instance_template
  replacement: $1
  action: replace
```

Creating a **new label** called `instance_template` using the source label `__meta_gce_metadata_instance_template`, we **copy it as-is**

```
- source_labels: [__meta_gce_project]
  separator: ;
  regex: (.*)
  target_label: project
  replacement: $1
  action: replace
```

Create a **new label** `project` using the source label `__meta_gce_project`

```
- source_labels: [__meta_gce_zone]
  separator: ;
  regex: .*/(.*)
  target_label: zone
  replacement: $1
  action: replace
```

Create a **new label** `zone` using the source label `__meta_gce_zone`

```
- source_labels: [__meta_gce_zone]
  separator: ;
  regex: .*/([0-9a-zA-Z-]*)-[a-z]
  target_label: region
  replacement: $1
  action: replace
```

Create a **new label** `region` using the source label `__meta_gce_zone` and applying the regexp capture filter `.*/([0-9a-zA-Z-]*)-[a-z]`

```
- source_labels: [__meta_gce_metadata_chef_run_list]
  separator: ;
  regex: role\[([0-9a-zA-Z-_]*)\]
  target_label: chef_role
  replacement: $1
  action: replace
```

New label, `chef_role` using `__meta_gce_metadata_chef_run_list`

```
- source_labels: [chef_role]
  separator: ;
  regex: Vitess_VtGate
  replacement: $1
  action: drop
```

If the `chef_role` label (we just created before) **contains** `Vitess_VtGate`, **drop** (**exclude**) the target from **scraping**

```
- source_labels: [chef_role]
  separator: ;
  regex: GCP_devdbpaymentsshard
  target_label: scrape_config
  replacement: $1
  action: keep
```

If the `chef_role` label **matches** `GCP_devdbpaymentsshard`, **keep** (**include**) the **target** for scraping. Now this rule is a bit interesting, as normally we use target_label and replacement when the action is `replace`, but here it's `keep`, so it's either going to ignore the target_label here or will make a copy of the chef_role (I'd say the first, but not sure)

```
  - separator: ;
    regex: (.*)
    target_label: dev
    replacement: $1
    action: replace
```

This is also a bit confusing as it's a replacement rule but missing the `source_labels` field. I'd say this is also ignored.

Correct, I copied this config directly from Prometheus' config, and in the config file we only have this:

```
  - target_label: dev
```

The rest are filled with the defaults, but I'd say this is **ignored**.

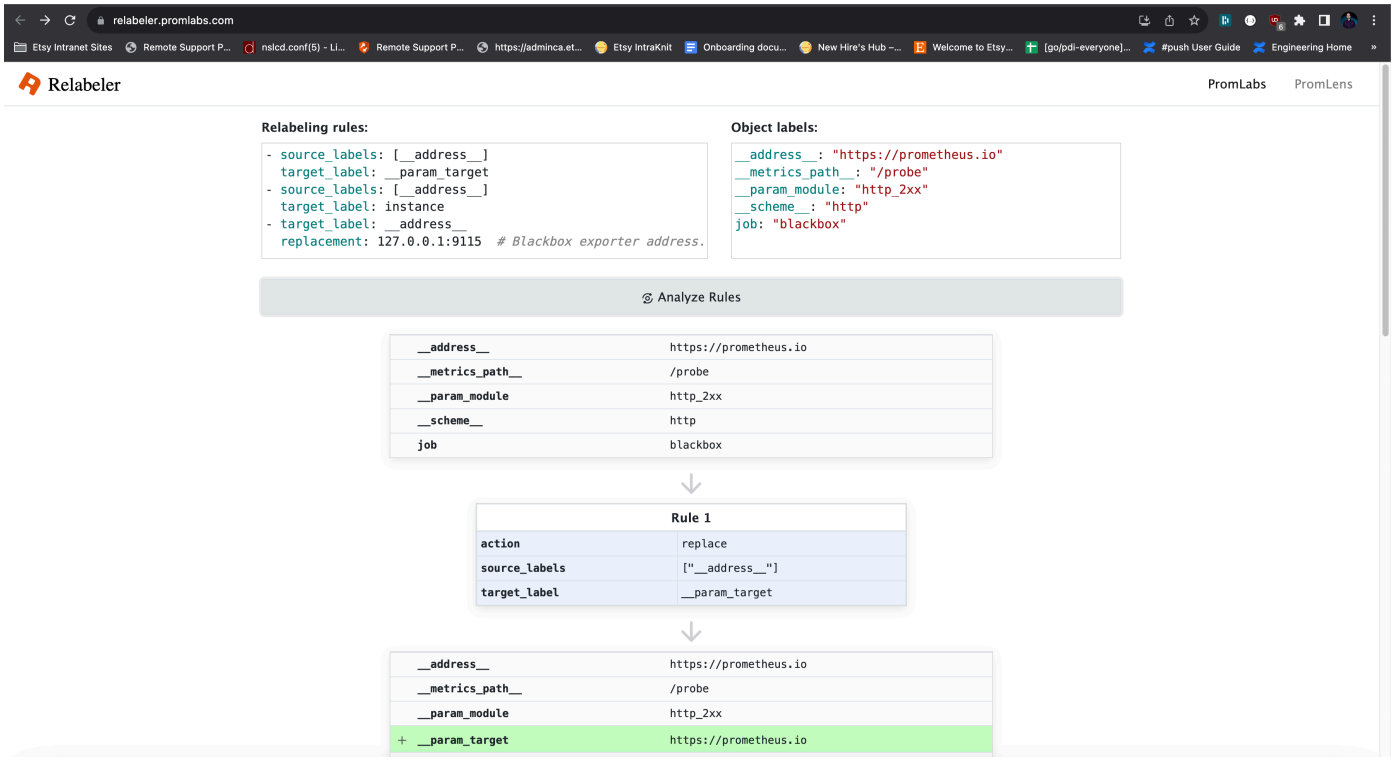
Test relabeling

We have all the **relabeling configurations**, **how** to **test** it? **One way** is to **commit** our change, and **see** if starts to scrape or not, but that's extremely **slow** and **cumbersome**, we'll have to find a shorter iteration loop for troubleshooting.

There are **two** more **effective** ways to **troubleshoot** the **relabelings**.

Relabeler

Relabeler is a website by the co-founder of Prometheus, and it's a pretty **awesome** tool to explain relabeling. We just have to add the **relabeling rules** and the **object** labels, and it shows step by step what's **relabeling** is **doing**:



Since it's an **external website** I **don't use** or **recommend** to use it with our labels, but to **experiment/test** individual rules with no real values is extremely **useful**.

Promtool

A bit more ugly but even **more usefuly** tool is **promtool**, what we'll use here to **review** our **relabeling rules**. We can do that from our **laptop**, we'll need to do the followings:

- **install prometheus** which includes promtool
  - `brew install prometheus`
- we need to **fetch** the live **config file** from our prometheus we want to troubleshoot
  - `kubectl cp -n vitess prometheus-vitess-0:/etc/prometheus/config_out/prometheus.env.yaml prometheus.env.yaml`
  - in this case, we copy the config file from the pod itself to our laptop
- we need to **authenticate** ourselves for 3rd party apps at **GCP**, for the service discovery to work
  - `gcloud auth application-default login`
- we'll need a **job** name to debug
  - we'll use the job we picked previously, `ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a`

Promtool can do a few **useful** things (validate config, validate exporter format, etc.) we'll use it to **verify** the **service discovery** layer, which also **includes** showing us the **label rewrites**. It's not as detailed as Relabel, it can't show us all the steps one by one, but it can **show** us the **original labels** and the **outcome**, that's good enough most of the time. Let's do it.

The **syntax** is:

```
promtool check service-discovery <config file> <job>
```

Let's do it:

```
aspitzer@aspitzer fix_prometheus % promtool check service-discovery prometheus.env.yaml
ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a
[
  {
    "discoveredLabels": {
      "__address__": "10.248.18.5:80",
      "__meta_gce_instance_id": "8396117199467144496",
      "__meta_gce_instance_name": "devdbpaymentsshard001a",
      "__meta_gce_instance_status": "RUNNING",
      "__meta_gce_interface_ipv4_nic0": "10.248.18.5",
      "__meta_gce_label_cluster": "devdbpaymentsshard",
      "__meta_gce_label_cost_center": "syseng",
      "__meta_gce_label_keyspace": "[redacted]payments_shard",
      "__meta_gce_label_lifecycle": "development",
      "__meta_gce_label_team": "storage",
      "__meta_gce_machine_type": "https://www.googleapis.com/compute/v1/projects/[redacted]
[redacted]dev/zones/us-central1-a/machineTypes/n2-standard-8",
      "__meta_gce_metadata_chef_attrs_json": "
{\"chef_environment\":\"production\",\"vitess_config\":{\"vttablet_config\":
{\"init_shard\":\"-80\",\"keyspace\":\"[redacted]payments_shard\"}}}",
      "__meta_gce_metadata_chef_run_list": "recipe[vitess::shard],
role[GCP_devdbpaymentsshard]",
      "__meta_gce_metadata_recreate_chef_client": "yes",
      ...
      "__meta_gce_network": "https://www.googleapis.com/compute/v1/projects/[redacted]
[redacted]prod/global/networks/[redacted]
      "__meta_gce_private_ip": "10.248.18.5",
      "__meta_gce_project": "[redacted]vitess-dev",
      "__meta_gce_subnetwork": "https://www.googleapis.com/compute/v1/projects/[redacted]
[redacted]prod/regions/us-central1/subnetworks/primary",
      "__meta_gce_tags": ",dev-ansible-client,dev-db-server,orchestrator-client,vtctld-
client,vtgate-server,vttablet-server,vttopo-client,",
      "__meta_gce_zone": "https://www.googleapis.com/compute/v1/projects/[redacted]vitess-
dev/zones/us-central1-a",
      "__metrics_path__": "/probe",
      "__param_module": "ssh_banner",
      "__scheme__": "http",
      "__scrape_interval__": "15s",
      "__scrape_timeout__": "10s",
```

```
      "job": "ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a"
    },
    "labels": {}
  },
...
```

Each object in this entry is a **discovered target**, and there are two **attributes** we'll look for: `discoveredLabels` are the labels **before** the rewrite, and `labels` are the labels **after** the label rewrite process, **that's** what the **scraping** works **with**.

The problem

Here we already **found** a **problem**, previously we wrote that the **scraper** is using the label `__address__` to figure out **what** to scrape, and our `labels` is **empty**! This means the **scraper** can't even **figure out** what to scrape, that's why it **never even went** to the **Blackbox exporter**.

Let's **confirm** that **all targets** are **excluded**:

```
aspitzer@aspitzer fix_prometheus % promtool check service-discovery prometheus.env.yaml
ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a | jq ".[] | .labels"
{}
{}
{}
{}
{}
{}
{}
{}
{}
```

All **empty**! And they are **exactly 9 entries**, just as we saw before when we saw **0/9** are **active targets**.

Ok, so we now we **confirmed** that the **problem** is with the **relabeling section.**

Since we already went through the relabeling rules, there is one that **looks** a potential problem:

```
  - source_labels: [chef_role]
    separator: ;
    regex: GCP_devdbpaymentsshard
    target_label: scrape_config
    replacement: $1
    action: keep
```

This says Prometheus to keep (include) this target for scraping, let's check the label `chef_role`, what we just **created** a few steps before:

```
  - source_labels: [__meta_gce_metadata_chef_run_list]
    separator: ;
    regex: role\[([0-9a-zA-Z-_]*)\]
    target_label: chef_role
    replacement: $1
    action: replace
```

OK, so **chef_role** is a regex rewrite, based on `__meta_gce_metadata_chef_run_list` label. Let's **review** that label's **value** (it's among the `discoveredLabels`):

```
      "__meta_gce_metadata_chef_run_list": "recipe[vitess::shard],
role[GCP_devdbpaymentsshard]",
```

Ha! I'm sure you all can **see** what the **problem** is here. There are usually two type of basic regex functions, one is **search** and the other is **match**. Search works in substrings, match doesn't. Match will only match if the expression matches the whole value/line (exact match), in this case `recipe[vitess::shard], role[GCP_devdbpaymentsshard]`, which `role\[([0-9a-zA-Z-_]*)\]` doesn't, it already fails at the beginning because the value starts with `recipe`.

Solution

Let's **correct** the **regex** and see if it **fixes** our **problem**, all I do is add `.*` to the beginning and end, so the **regex works** also as a **substring**:

```
aspitzer@aspitzer fix_prometheus % diff prometheus.env.yaml prometheus.env.yaml_ORIG
1843c1843
<   - regex: .*role\[([0-9a-zA-Z-_]*)\].*
---
>   - regex: role\[([0-9a-zA-Z-_]*)\]
```

Let's re-run the **promtool** command with the updated config to see if **labels** show up this time:

```
```

```
aspitzer@aspitzer fix_prometheus % promtool check service-discovery prometheus.env.yaml
ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a | jq ".[] | .labels"

{
  "__address__": "blackbox-exporter.exporters.svc:9115",
  "__metrics_path__": "/probe",
  "__param_module": "ssh_banner",
  "__param_target": "10.248.18.5:22",
  "__scheme__": "http",
  "__scrape_interval__": "15s",
  "__scrape_timeout__": "10s",
  "chef_role": "GCP_devdbpaymentsshard",
  "instance": "devdbpaymentsshard001a",
  "job": "ssh-exporter-legacy-paymentsshard-[redacted]vitess-dev-us-central1-a",
  "project": "[redacted]vitess-dev",
  "region": "us-central1",
  "zone": "us-central1-a"
}
{}
{}
{}
{}
{}
{}
{}
{}
```

Yes! It **worked**. This **change** will start **scraping** this target, as the **labels** are looking correct, now the **scraper knows** where to **connect**!

Conclusion

The reason why the **Prometheus** block **didn't work** because the **label rewrite** rule which creates the **chef_label** label wasn't **matching** fully the **value** on these **VMs**, so the chef_role label became an empty string. In other words, the **labeling rule** fails to **determine** the **chef_role** correctly in some occasions, like in this example.