ENDPOINT VISIBILITY WITH OSQUERY AND LOKI

GEORGE ADAMS IV & ED WELCH

USE CASE

DETECTING SSH CONNECTIONS AND ALERTING.

This is the end goal we'll build towards as we learn a little more along the way.

FOLLOW ALONG

https://github.com/geowa4/learn-loki

Clone this repo to see slides and run these tools in a Vagrant VM.

OSQUERY

RELEASED BY FACEBOOK IN 2014

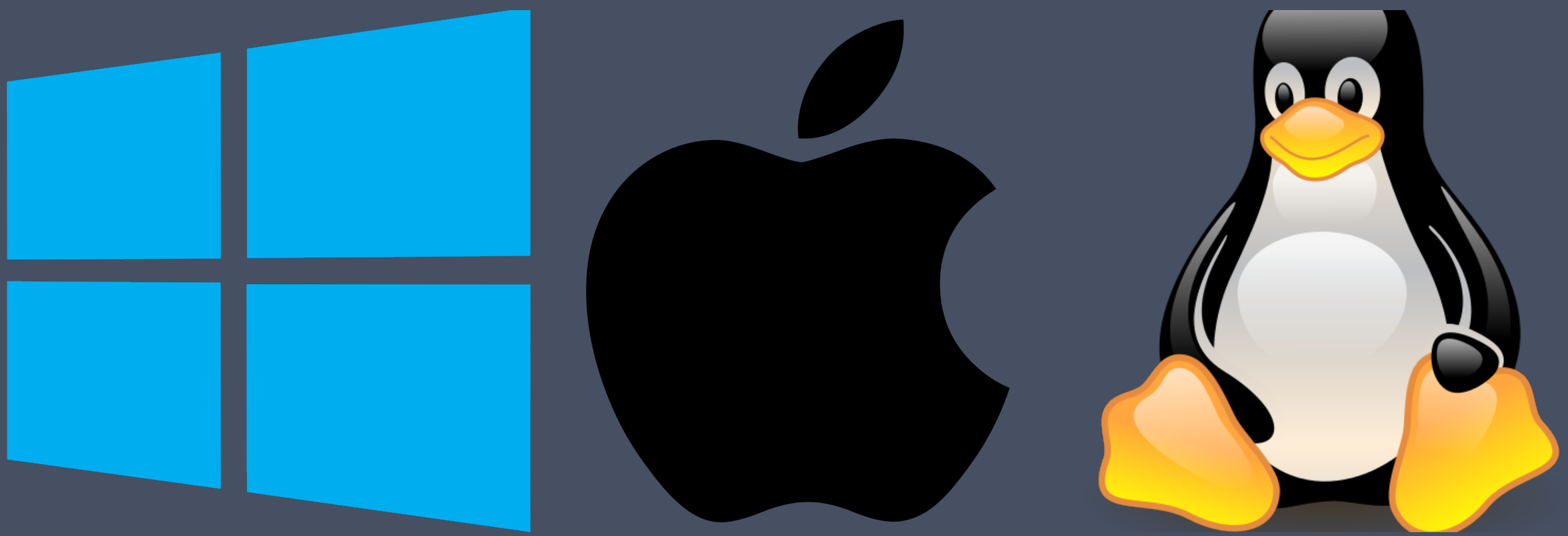
```
commit 73a32b729403b2f5a7c204b0f7cfb86fdfdd0a85
Author: mike@arpaia.co <mike@arpaia.co>
Date:   Wed Jul 30 17:35:19 2014 -0700

    Initial commit
```


It had existed internally for some time, but this was when they dumped everything to GitHub. The next couple commits were fixing repo layout issues from the migration.

OSQUERY

WORKS ON MY MACHINE



One reason I like Osquery is that it runs everywhere.

I can install it on my laptop and my entire fleet.

OSQUERY

SQL INTERFACE TO YOUR ENDPOINTS

```
.schema processes

CREATE TABLE processes(
  `pid` BIGINT, `name` TEXT, `path` TEXT, `cmdline` TEXT,
  `disk_bytes_read` BIGINT, `disk_bytes_written` BIGINT,
  ...
  PRIMARY KEY (`pid`)
) WITHOUT ROWID;
```


If we interrogate the schema for the processes table, we see a familiar sight.

OSQUERY

```
select max(disk_bytes_read), pid, name, cmdline, cwd from processes;

+----------------------+------+------+---------+---------------+
| max(disk_bytes_read) | pid  | name | cmdline | cwd           |
+----------------------+------+------+---------+---------------+
| 10465280             | 5478 | bash | -bash   | /vagrant/demo |
+----------------------+------+------+---------+---------------+
```


We even get common SQL functions like max, count, etc.

OSQUERY

```
select p.name, l.port, l.protocol
from processes p inner join listening_ports l on p.pid = l.pid
where p.name = 'VBoxHeadless' and l.port <> 0;

+--------------+------+----------+
| name         | port | protocol |
+--------------+------+----------+
| VBoxHeadless | 3000 | 6        |
| VBoxHeadless | 3100 | 6        |
| VBoxHeadless | 9080 | 6        |
| VBoxHeadless | 9090 | 6        |
| VBoxHeadless | 2222 | 6        |
+--------------+------+----------+
```


Why is a SQL interface nice?
Who remembers how the command to list what processes are listening on which ports?
How about cross-platform?
Not me.
Exploration is easier in SQL than parsing man pages.

OSQUERY

```
select c.name, p.port, p.host_port
from docker_containers c inner join docker_container_ports p
on c.id = p.id;

+--------------------+------+-----------+
| name               | port | host_port |
+--------------------+------+-----------+
| /demo_loki_1       | 80   | 0         |
| /demo_loki_1       | 3100 | 3100      |
| /demo_grafana_1    | 3000 | 3000      |
| /demo_prometheus_1 | 9090 | 9090      |
| /demo_promtail_1   | 9080 | 9080      |
+--------------------+------+-----------+
```


Of course you can also query Docker resouces.

Here we're doing the Docker container version of the process query eariler.

OSQUERY - LAST SCHEMA

```
.schema last

CREATE TABLE last(
   `username` TEXT, `time` INTEGER, `host` TEXT,
   `pid` INTEGER, `tty` TEXT, `type` INTEGER
);
```


Ok, so we can query things.
We have a job to do: detect SSH connections.
Using the `last` table we can see who's been connecting and when.

OSQUERY - LAST QUERY

```
select * from last;
```

```
+----------+------+------+------+------------+-------------------+
| username | tty  | pid  | type | time       | host              |
+----------+------+------+------+------------+-------------------+
| reboot   | ~    | 0    | 2    | 1566866107 | 4.15.0-52-generic |
| runlevel | ~    | 53   | 1    | 1566866118 | 4.15.0-52-generic |
|          | ttyS0| 859  | 5    | 1566866119 |                   |
| LOGIN    | ttyS0| 859  | 6    | 1566866119 |                   |
|          | tty1 | 879  | 5    | 1566866119 |                   |
| LOGIN    | tty1 | 879  | 6    | 1566866119 |                   |
| vagrant  | pts/0| 5396 | 7    | 1566869034 | 10.0.2.2          |
| vagrant  | pts/1| 6465 | 7    | 1566870878 | 10.0.2.2          |
|          | pts/1| 6465 | 8    | 1566870880 |                   |
| vagrant  | pts/1| 6571 | 7    | 1566870886 | 10.0.2.2          |
|          | pts/1| 6571 | 8    | 1566870910 |                   |
+----------+------+------+------+------------+-------------------+
```


What querying `last` looks like.

OSQUERY - PACKS

```
{
  "queries": {
    "last": {
      "query": "select * from last;",
      "interval": "60",
      "platform": "posix",
      "version": "1.4.5",
      "description": "..."
    }
  }
}
```


Cool, but I'm not going to run that query manually on every server.

With Osquery, you configure "packs" to do that for you.

Here we see we're selecting all the columns from `last` every 60 seconds and only on POSIX machines.

OSQUERY - DECORATORS

```
{
  "decorators": {
    "load": [
      "SELECT uuid AS host_uuid FROM system_info;",
      "SELECT user AS username FROM logged_in_users ORDER BY time DESC LIMIT 1;"
    ]
  }
}
```


Ok, I know someone got in, but where was it?

That's where decorators come in. The results from these queries are appended to every query result. If you're on AWS, you can add a query for your EC2's tags.

OSQUERY - RESULTS

/var/log/osquery/osqueryd.results.log


```
{
  "name": "pack_incident-response_last",
  "hostIdentifier": "ubuntu-bionic",
  "calendarTime": "Tue Aug 27 01:55:13 2019 UTC",
  "decorations": {
    "host_uuid": "2401CCE9-23EA-4D4D-8C84-D5C8437EBE15",
    "username": "vagrant"
  },
  "columns": {
    "host": "10.0.2.2",
    "pid": "6465",
    "time": "1566870878",
    "tty": "pts/1",
    "type": "7",
    "username": "vagrant"
  },
  "action": "added"
}
```


This is a sample result showing us logging in as `vagrant`.

The result and others from queries defined in our packs can be found in that log file.

One key to remember about the results log is that if the query returns no new data, nothing is written to the log.

Only changes are appended.

OSQUERY - RECAP

IT'S BEEN AROUND A WHILE

CROSS-PLATFORM

IT'S JUST SQL

SCHEDULE QUERIES WITH "PACKS"

Next, we'll see how to aggregate those intrusions so we can alert on them.

LOKI

PROMETHEUS-INSPIRED LOGGING FOR CLOUD NATIVES.

MADE BY GRAFANA

To know what this means, let's take a quick detour to Prometheus.

PROMETHEUS

METRICS COLLECTION VIA "PULL"

PROMETHEUS - PUSH

TODO: diagram push vs pull

PROMETHEUS - PULL

TODO: diagram push vs pull

PROMETHEUS - TIME-SERIES

TIME-SERIES DATA STORE

QUERYABLE VIA PROMQL

PROMETHEUS - SCRAPING

```
scrape_configs:
  - job_name: "promtail"
    static_configs:
      - targets:
          - promtail:9080
```


Here we're showing a simple config that tells Prometheus to scrape the metrics from a host "promtail" on port 9080.
Prometheus will issue a GET and store the parsed metrics from the response.

PROMETHEUS - DATA STRUCTURE

METRICS HAVE LABELS IN ADDITION TO VALUES.

```
rss_enjoyment{track="tech", talk="osquery_loki"} 11
```


What is Prometheus scraping and how do we query it?
This is what Prometheus scrapes.

The "rss_enjoyment" metric with labels "track" and "talk" is all the way up to 11.
Querying is done using the same syntax, but feel free to leave off bits like the specific talk if you just want to chart the "tech" track.

LOKI - DATA STRUCTURE

LOG ENTRIES HAVE LABELS. TOO.

```
{track="tech", talk="osquery_loki"} "best talk ever"
{track="tech", talk="osquery_loki"} "i want to know more"
{track="tech", talk="osquery_loki"} "i hear that one guy runs rocdev"
```


Loki uses the same structure as Prometheus in that each log entry uses labels.
And the label queries you can do (LogQL) look just like label queries in PromQL.

LOKI - LABELS

THE MATCHING LABELS ALLOW US TO SWITCH BACK AND FORTH FREELY.

This is a huge benefit that reduces the volume you have to learn and streamlines investigations.

LOKI - QUERY

LOGQL

```
$ logcli query --tail '{name="pack_incident-response_last"}'
2019-08-29T03:01:37Z
{filename="/var/log/osquery/osqueryd.results.log", job="osquery_results", name="pack_incident-response_last"}
{
  "name":"pack_incident-response_last",
  "hostIdentifier":"ubuntu-bionic",
  "calendarTime":"Thu Aug 29 03:01:37 2019 UTC",
  "unixTime":1567047697,
  "epoch":0,
  "counter":115,
  "decorations":{"host_uuid":"661449FD-E11A-462B-9EA9-63A3EE8F9BDC","username":"vagrant"},
  "columns":{"host":"10.0.2.2","pid":"7404","time":"1567047680","tty":"pts/1","type":"7","username":"vagrant"},
  "action":"added"
}
```


Let's take a look at a LogQL query and try to find our sample result from earlier.
This commonality in query structure is nice, but there's one more link with Prometheus and Loki: metrics extraction.

LOKI - COLLECTION

PROMTAIL

We've seen how Osquery generates logs.

We've seen how to read logs.

Promtail is how those logs get from your endpoint to Loki.

PROMTAIL

FORWARDS LOGS
AND
EXTRACTS METRICS

Promtail does something more than just forwarding logs from A to B.

It reads them, parses them, and makes metrics available.

PROMTAIL - SCRAPING

```
clients:
  - url: http://loki:3100/api/prom/push
scrape_configs:
  - job_name: osquery
    static_configs:
      - targets:
        - localhost
    labels:
      job: osquery_results
      __path__: /var/log/osquery/osqueryd.results.log
```


But let's start with how it knows how to get Osquery results to Loki.

Basically, we make a scrape config that looks just like what we had for Prometheus.

PROMTAIL - RESULT REMINDER


```
{
  "name": "pack_incident-response_last",
  "hostIdentifier": "ubuntu-bionic",
  "calendarTime": "Thu Aug 29 03:01:37 2019 UTC",
  "unixTime": 1567047697,
  "epoch": 0,
  "counter": 115,
  "decorations": {
    "host_uuid": "661449FD-E11A-462B-9EA9-63A3EE8F9BDC",
    "username": "vagrant"
  },
  "columns": {
    "host": "10.0.2.2",
    "username": "vagrant",
    "type": "7",
    "time": "1567047680",
    "tty": "pts/1",
    "pid": "7404"
  },
  "action": "added"
}
```


Let's just remind ourselves what the Osquery result looked like before we start processing it with Promtail.

PROMTAIL - PIPELINES

```
pipeline_stages:
  - json:
      expressions:
        timestamp: unixTime
        name: name
        username: columns.username
  - timestamp:
      source: timestamp
      format: Unix
  - labels:
      name: name
```


Once we read the log line, we want to pull some things out of it.
First, it's JSON; always use structured logs since it makes automation so much easier and readability doesn't really suffer.
Then we want to convert the `unixTime` field to `timestamp` and tell Promtail that is Unix format so Loki shows the right time for the event.
Finally, we take the `name` of the pack query and add it to our labels for searching.

PROMTAIL - METRICS

```
pipeline_stages:
  - ...
  - metrics:
      last_logins:
        type: Counter
        description: count last logins
        source: name
        config:
          value: pack_incident-response_last
          action: inc
```


And this gets us our metric counting how many times `last` finds a new event.

PROMTAIL - PROMETHEUS

```
scrape_configs:
  - job_name: "promtail"
    static_configs:
      - targets:
          - promtail:9080
```


Now this is where our Prometheus config from earlier comes in.
We can use Prometheus to scrap Promtail to record that counter.

WHERE WE ARE NOW

OSQUERY PRODUCING RESULTS

PROMTAIL FORWARDING TO LOKI

QUERY AND TAIL LOGS IN LOKI

PROMTAIL EXTRACTING METRICS

PROMETHEUS SCRAPING PROMTAIL

We've covered a lot of ground so let's quickly recap what we've accomplished so far.

WHAT'S LEFT

CHARTING & ALERTING

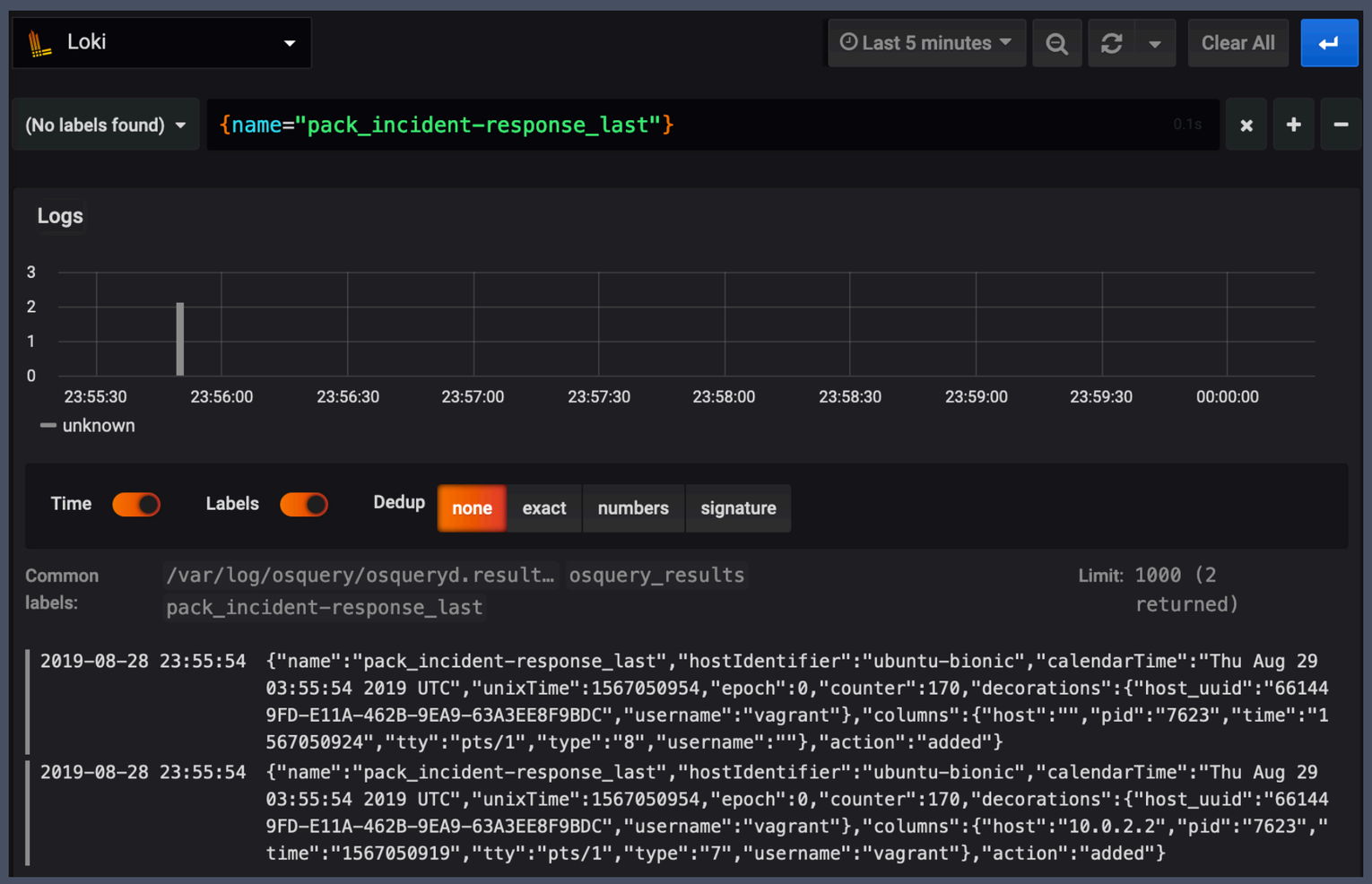
GRAFANA

SUPPORTS BOTH PROMTHEUS AND LOKI AS DATA SOURCES

Charting and alerting are things Grafana does well.

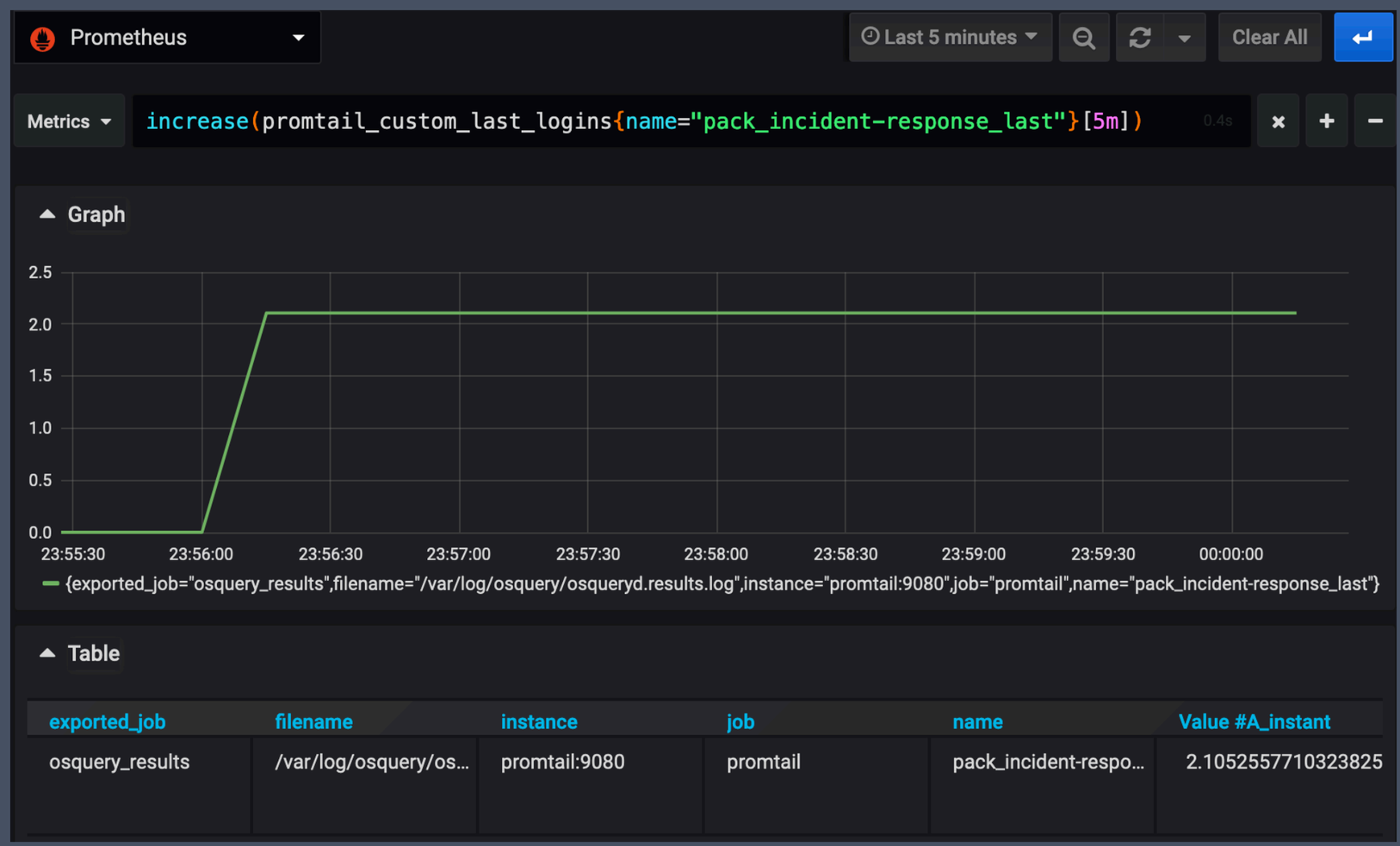
If you're a Prometheus shop, you might have Alertmanager running, and that will work, too.

GRAFANA - LOKI



Explore Loki in Grafana to find our event.

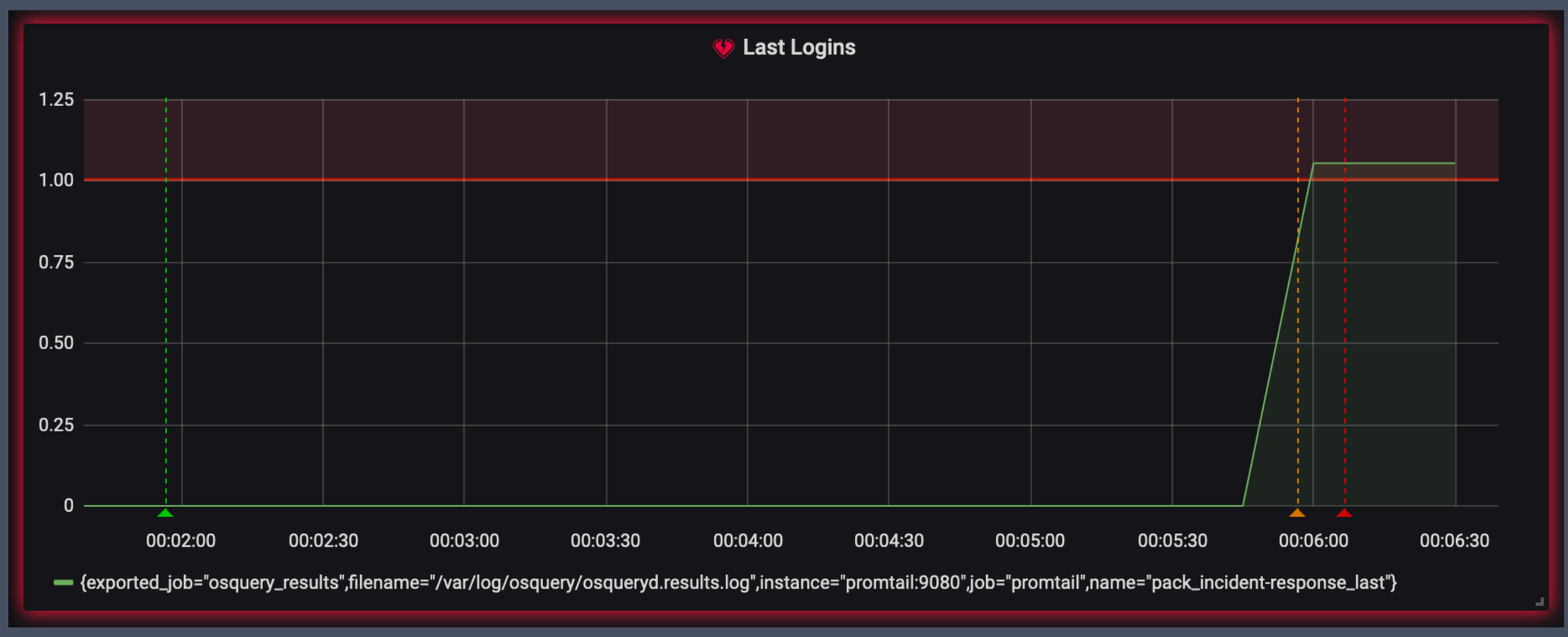
GRAFANA - PROMETHEUS



Use the same labels to search Prometheus.

GRAFANA - ALERTING

```
increase(promtail_custom_last_logins{name="pack_incident-response_last"}[5m])
```



Make a simple chart on a dashboard and alert if the number of `last` logins increases.

PUTTING IT ALL TOGETHER

TODO: diagram all the components