

Session 09 report

Authors:

Group K: Caspar Olesen (cole@itu.dk), Dag Bjerre Andersen (daga@itu.dk), Emil Bartholdy (emba@itu.dk), Gustav Kofoed Clausen (gucl@itu.dk), Lasse Raatz (lraa@itu.dk).

Date:

April 15, 2020

Own SLA

The content of our service-level agreement (SLA) for our simulator API consists of the following points:

- **Monthly Uptime Percentage:**

We strive to deliver a monthly 99,99% uptime, meaning that the API is able to serve valid responses to its client in this timeframe.

- **Mean Time To Recover:**

In case of outage, the recovery time is expected to be no more than 12 hours.

- **Average Response Time:**

The expected average response time on API calls on the following API routes:

- GET `/latest` : 100ms
- POST `/msgs/{username}` : 100ms
- GET `/fills/{username}` : 100ms
- POST `/fills/{username}` : 150ms
- GET `/msgs` : 5s
- GET `/msgs/{username}` : 700ms

This is targeted clients located in Europe only, and is measured on a monthly basis.

- **Error Rates:**

The average monthly amount of valid requests from client that results in

response with HTTP status code ≥ 500 is 0,1%.

Artificial Problem Introduced

On April 4, we deployed a new version of our simulator API application, where all requests to the `/msgs` endpoint includes an additional 5000 milliseconds processing time to the original response. This was done to exceed the guaranteed five seconds average response time as specified in our SLA.

As of April 12, we have received no reports from group J, that monitors our system, about the problem introduced.

Monitoring of Group L

On March 26th, we received group L's SLA. It consisted of the following points:

- **Availability of the Service:** 99,9%
- **Response Time:** $\leq 10\text{ ms}$ - that is, a response should maximally be received 10 ms or less after the request was sent.
- **Mean Response Time:** $\leq 5\text{ms}$ - that is, a response should on average be received 5 ms or less after the request was sent.
- **Mean Time to Recover:** $\leq 1\text{ day}$

We set up a number of Datadog probes that send (valid) requests to all of group L's endpoints every five minutes from three locations in Europe: Frankfurt, Ireland, and London. It was immediately obvious that the group wasn't able to live up to their SLA. In fact, their mean response time across all their endpoints was 192 ms - quite far from the 5 ms or less that their SLA promised.

We therefore reached out to the group, informing them of the fact that they could not uphold their SLA, including a link to the Datadog dashboard that we set up. They responded that they were never able to live up to that SLA and sent a new one. The new SLA changed their response time promises to:

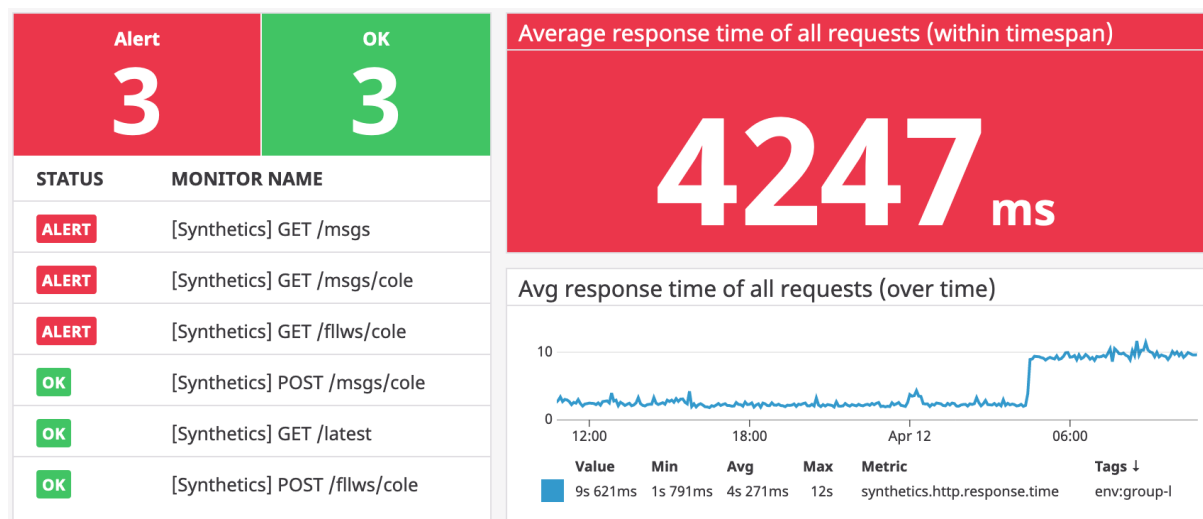
- **Response Time:** $\leq 2000\text{ ms}$
- **Mean Response Time:** $\leq 500\text{ ms}$

So, they updated their performance promises from very optimistic to realistic, erring on the pessimistic side. This SLA they were able to uphold for some days, except for the fact that their `/msgs` endpoint responded with a `500` error each time. The average response time did, however, grow steadily. The monitoring was set in place on March 30th, and already on April 1st their average response time exceeded `500 ms` - and it has not dipped below that threshold since. The `/msgs` bug was fixed, but the response time of the endpoint grows in correlation with the amount of messages, and is at the time of writing well above the `2000 ms` threshold.

On April 12th, we noticed a significant increase in the response time for two of their endpoints, namely `GET /msgs/{username}` (`5 s` to `29 s`) and `GET /flws/{username}` (`2 s` to `24 s`). We immediately reached out to the group to let them know that something is up. We have yet to hear any response from the team.

At the time of writing this section, their average response time had grown to `9621 ms`, obviously also spiked by the two slowed endpoints mentioned in the previous paragraph. Before the spike, their average response time was around `2500 ms`.

Screenshot of the Datadog dashboard. Timespan: `The Past Day`



Red means that the response time exceed their SLA threshold: `2000 ms` for each single request and `500 ms` for the average response time.

Penetration Testing Own System

Assets

- **The Production VM Running Services**

If the VM goes down (causing downtime) it will result in following financial losses:

- We pay for a VM that we cannot use.
- If the service becomes unavailable our revenue stream will be affected negatively.

- **User Data**

We collect data about our users. The most sensitive information consists of users' emails and passwords.

- **Various Credentials For Systems**

- SSH keys we use to log into our production VM.
- Logins to external services we use e.g. GitHub, DigitalOcean and Kibana.

- **Logging Data**

- System logs can help us patch security vulnerabilities or optimise uptime and performance.

- **Source code**

- The source code is the cornerstone of the platform. It is our primary intellectual property (IP).

- **Employees**

- Each employee holds valuable information about different subsystems of the application.
- Each employee has a different set of skills for different technologies.

Risks

Risks can have an impact on the service we provide, if it is realised. An impact can be categorised according to the CIA triad: Confidentiality, Integrity and Availability. Confidentiality concerns having data only be available for suitable

actors. Integrity involves maintaining the consistency, accuracy, and trustworthiness of data over its life cycle. Finally, Availability concerns the availability of our services and data.

Each risk below has an impact which has been categorised according to the CIA triad. Furthermore, each risk has been entered in a risk matrix (see next section) with its likelihood of happening and its severity should the risk materialise.

1. Attackers gain SSH access to production VM (a security breach) by getting hold of authorised SSH key or root password

Impact:

- Bringing down production VM (Availability).
- Unauthorised DB access to data leak (Confidentiality) and data manipulation (Integrity).

2. Third-party software (Node packages, etc.) contains malicious code or known security vulnerabilities

Impact:

- Data leaks (Confidentiality).
- Bringing down production VM if unauthorised access is given (Availability).

3. Denial of service attacks

Impact:

- Slowing or bringing down production VM (Availability).

4. SQL Injections

Impact:

- Data leak (Confidentiality).
- Data manipulation (Integrity).

5. Unauthorized access to user accounts

Hacking into user accounts by guessing passwords.

Impact:

- A hacker gains access to user data (Confidentiality), can act as users, or manipulate user data (Integrity).

6. **Malicious source code is pushed to the code repository**

When pushing code to the repository on master branch, code is directly deployed to production VM (if the tests complete). Attackers or internal employees (perhaps unintentionally) are able to push malicious code.

Impact:

- Security vulnerabilities in code (Integrity, Confidentiality)
- Slow down of system performance (Availability)

7. **Services we rely on experience downtime**

Impact:

- DigitalOcean: Bringing down production VM (Availability).
- Sonar Cloud: Not able to run a subset of tests.
- Dockerhub: Not able to deploy new containers to production.
- Github: No repository manager and no access to deployment-pipeline.

8. **Services we rely on experience security breaches**

Impact:

- Unauthorised access to services and production VM (Confidentiality, Integrity, Availability)

9. **Man-in-the-middle attacks due to non-encrypted traffic**

Currently, our service does not use HTTPS to encrypt traffic with sensitive information, like user passwords, between client and server.

Impact:

- Data leaks (Confidentiality).

10. **Use of default credentials for tools and simulator application**

Default credentials used for services like Kibana, Grafana. Furthermore, credentials to access simulator is hard-coded in source code.

Impact:

- Data leaks (Confidentiality).

11. **Source code leak**

Impact:

- Competitors can easily copy or mimic our services.
- Malicious actors can find vulnerabilities in our system for various attacks.





Risk Matrix

Table 1 contains each risk identified by its number above. They have been categorised according to likelihood of happening and severity should the risk materialise.

Scale definition

- Catastrophic: We can't recover
- Critical: Will harm business and image
- Marginal: We will survive
- Negligible: Little to no impact

Table 1

<u>Aa</u> Likelihood / Severity	 Negligible	 Marginal	 Critical	 Catastrophic
<u>Certain</u>		11		
<u>Likely</u>		10		
<u>Possible</u>		9	2	
<u>Unlikely</u>		5		6
<u>Rare</u>		3	7, 4	8, 1

Risk Mitigations

We have implemented some risk mitigation strategies in order to accommodate for the risks identified above.

- OWASP ZAP: Tool to find vulnerabilities.
- Dependabot: Find known security vulnerabilities in NPM dependencies.

- Code Review: Enables colleagues to inspect code for security vulnerabilities.
- Testing: Failed tests block the ability to merge/push to master.
- SonarCloud: Finds simple bugs and security issues in source code.
- Monitoring and logging: We are logging all HTTP requests to our simulator API including errors of bad requests and server error to detect irregular activity or sign of attacks.
- Snapshooter.io: Regularly backup of data in production database.

Risk Acceptance

- We cannot cannot change default credentials for Grafana and simulator API due to requirements from the course.
- No DRP (disaster recovery plan) established.

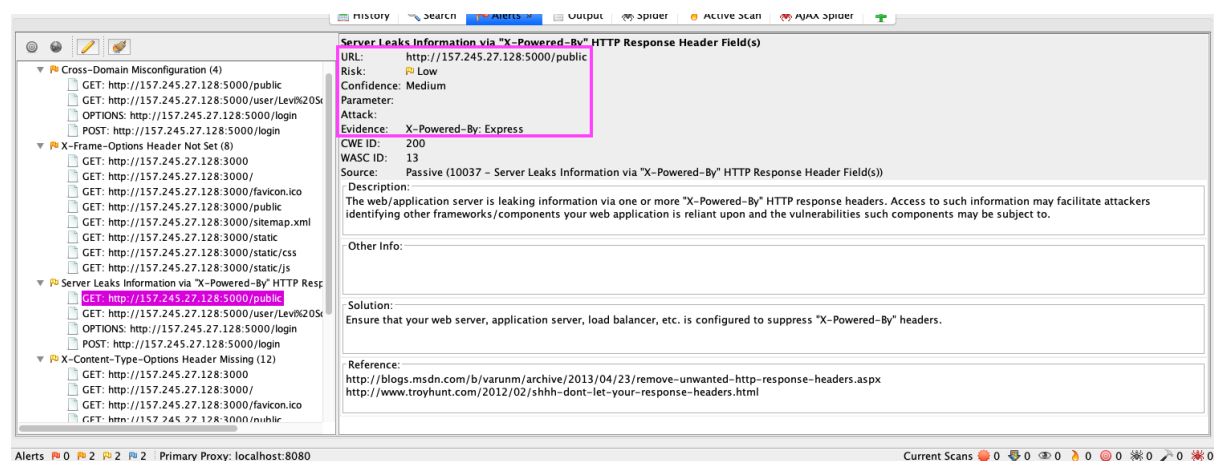
Results From Penetration Testing Own System

Use Of OWASP ZAP Tool

OWASP ZAP is an open-source web application security scanner. It is very simple to use. Just copy-paste the URL and run the crawler. The program lists all the issues/alerts it found. All alerts are marked with a risk-level and a small description.

Finding A Vulnerability

The result of running the ZAP tool gave us the following report.



From the above we see that that it reports a vulnerability: Server Leaks Information via `X-Powered-By` HTTP Response Header Field(s).

Fixing The Vulnerability

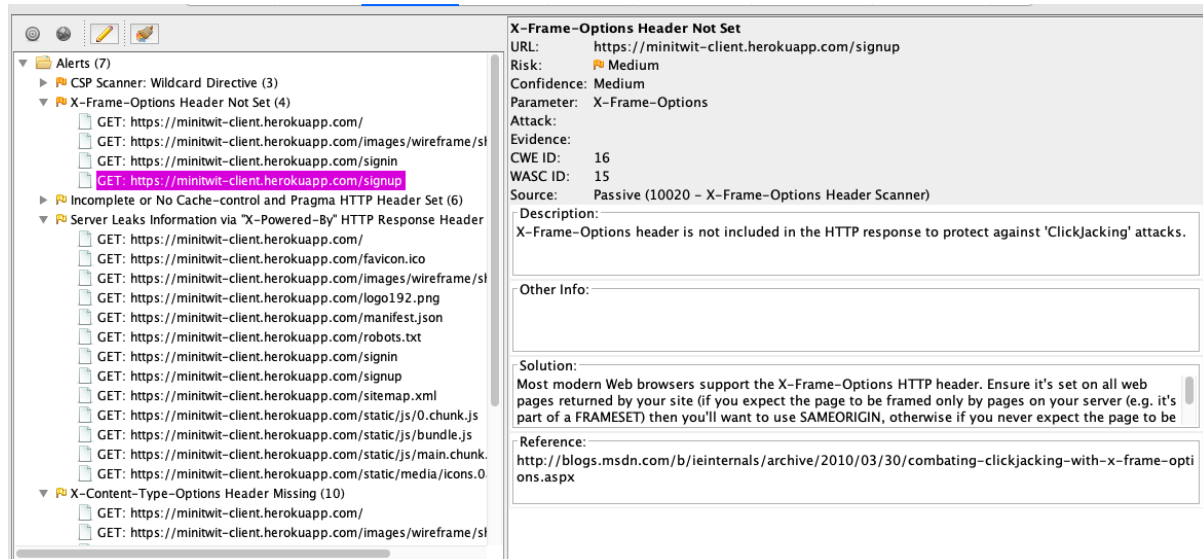
In order to fix the vulnerability, we used a tool called `helmet` which prevents some of Express' security issues (Express is the popular framework we use to create our server). OWASP ZAP reported that our `http` responses exposed which framework we are powered by (see the pink rectangle) which could be valuable information for attackers. We fixed these issues by implementing `helmet` in our server which prevents this information from being sent with our `http` responses. This mitigates risk number 2.

Result of Test of Security of Monitored Team

Running OWASP ZAP on Group L's endpoints exposed some of the same vulnerabilities as we had in our own service. They had low and medium risk issues but no high-risk issues.

The screenshot shows the OWASP ZAP Alerts window. On the left, a tree view lists various alerts, with 'Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)' selected. The right pane displays the details for this alert. The URL is `https://minitwit-client.herokuapp.com/`, the risk is 'Low', and the confidence is 'Medium'. The parameter is 'X-Powered-By: Express'. The attack evidence shows the 'X-Powered-By: Express' header. The CWE ID is 200 and the WASC ID is 13. The source is 'Passive (10037 - Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s))'. The description explains that the web/application server is leaking information via one or more 'X-Powered-By' HTTP response headers, which could facilitate attackers identifying other frameworks/components. The solution is to ensure that the web server, application server, load balancer, etc. is configured to suppress 'X-Powered-By' headers. The reference links provided are <http://blogs.msdn.com/b/varunm/archive/2013/04/23/remove-unwanted-http-response-headers.aspx> and <http://www.troyhunt.com/2012/02/shhh-dont-let-your-response-headers.html>.

Through the tool we can see that their frontend is vulnerable to clickjacking. Clickjacking is when a hacker loads your page in a frame and puts an invisible overlay on top of it so the page looks exactly like yours. The hacker may then trick the user into clicking hidden buttons.



A common type of attack is a denial-of-service attack overloading endpoints to the extent that the targeted service becomes unavailable. This can be a serious issue and needs to be tested in a controlled environment. We attempted to throttle the `/msgs` -endpoint which resulted in a temporary outage of every endpoint besides `/latest`. We suspect that something happened in the database since `/latest` is likely the only endpoint not connected to the database. The service recovered quickly. We monitored our attack using Datadog (see image).

Synthetics > GET /msgs

ALERT Last ran 2 mins ago (Apr 8, 2020, 5:40 pm)

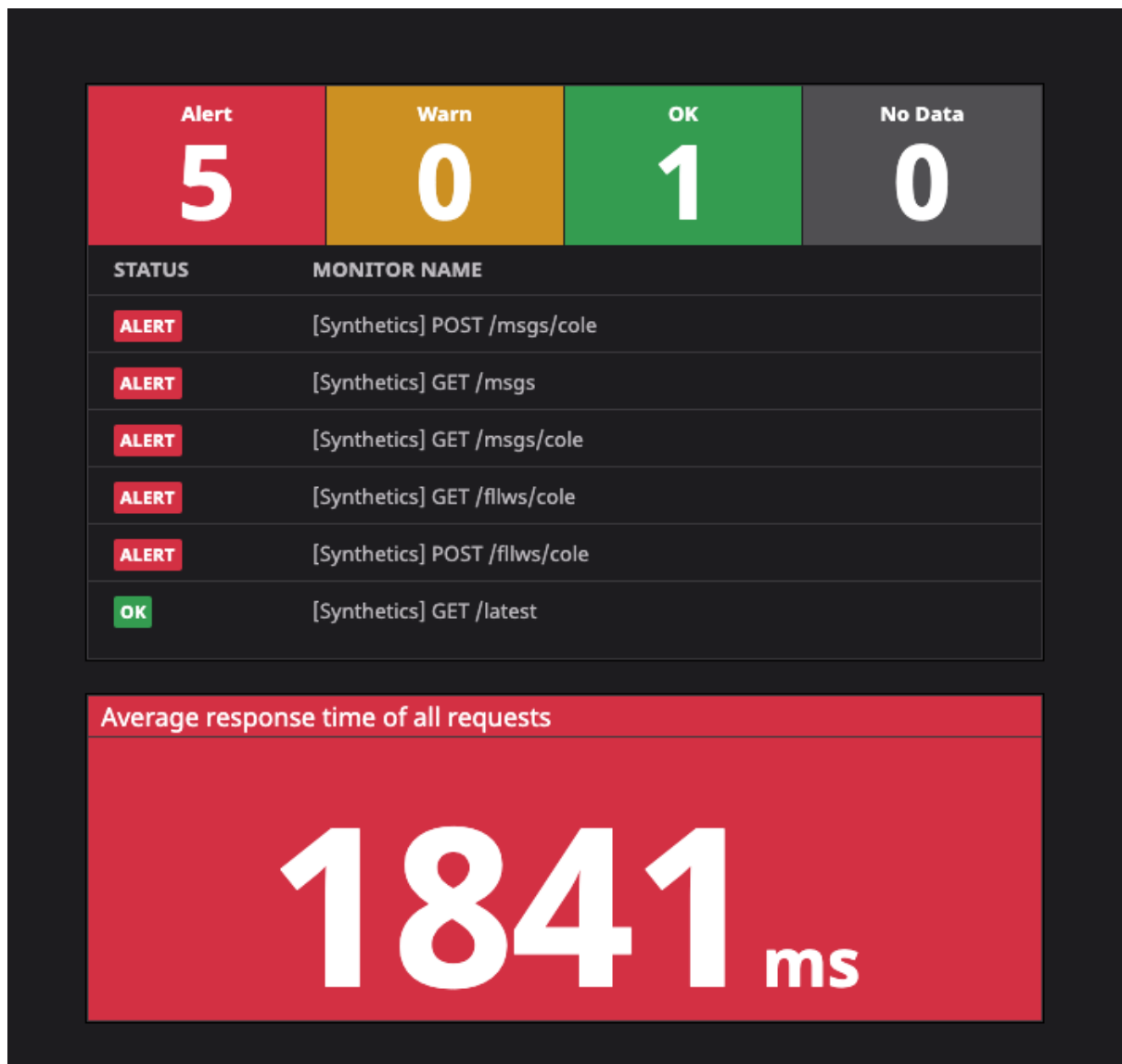
GET /msgs

All locations 4h Past 4 Hours

Last updated less than a minute ago Refresh

STATUS	DATE	DURATION	LOCATION
ALERT	2 mins ago Apr 8, 2020, 5:40 pm	-	Ireland (AWS)
ALERT	2 mins ago Apr 8, 2020, 5:40 pm	-	London (AWS)
ALERT	2 mins ago Apr 8, 2020, 5:40 pm	-	Frankfurt (AWS)
ALERT	7 mins ago Apr 8, 2020, 5:35 pm	-	Ireland (AWS)
ALERT	7 mins ago Apr 8, 2020, 5:35 pm	-	London (AWS)
ALERT	7 mins ago Apr 8, 2020, 5:35 pm	-	Frankfurt (AWS)
OK	12 mins ago Apr 8, 2020, 5:30 pm	1651ms	Ireland (AWS)
OK	12 mins ago Apr 8, 2020, 5:30 pm	1643ms	London (AWS)
OK	12 mins ago Apr 8, 2020, 5:30 pm	1835ms	Frankfurt (AWS)

Picture 1: `/msgs` endpoint does not keep their SLA. Note how duration does not have a value for the endpoint. That means the service did not respond to the request at all.



Picture 2: We see that all other endpoints has been affected by overwhelming the `/msgs` endpoint except for `/latest`.

The code used for the attack can be seen below. It starts 1000 processes each calling the `/msgs` endpoint over and over again.

```
const axios = require('axios');
const route = 'https://minitwit-simulator-api.herokuapp.com/msgs';

const attack = async () => axios.get(route).then(attack);

let attacks = [];

let n = 1000;
while (n-->0) attacks.push(attack());

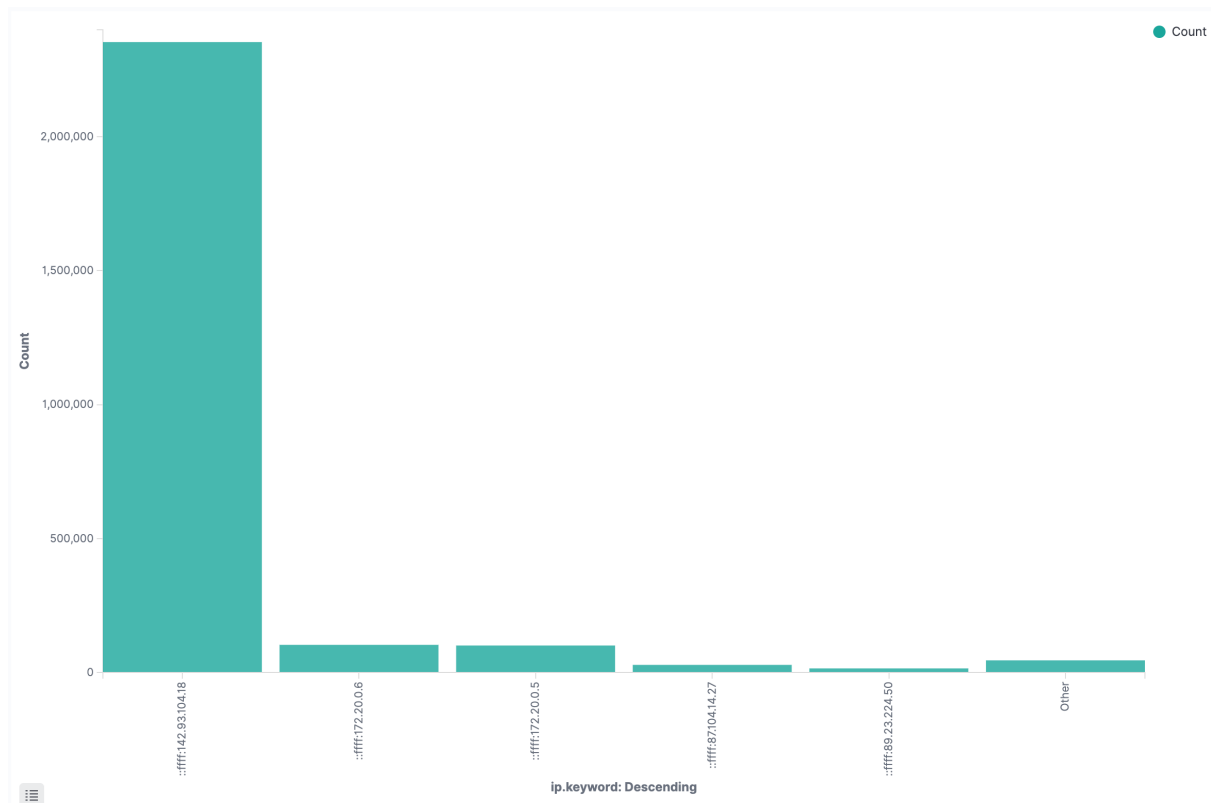
Promise.all(attacks).catch(console.error);
```

Detecting Attacks on Own System

As of April 11, we have not detected any attacks or irregular activity in our simulator API apart from a small irregularity concerning special characters.

Traffic

As shown in the figure below, our simulator API receives traffic from the simulator client (with IP `142.93.104.18`), our Prometheus container (with the private IP `127.20.0.X`), and two other clients that is tracked to be located in Denmark based on a lookup on their IP. These two clients is assumed to be monitoring agents of group J that is monitoring us, and based on the request amount and the payload of their requests there seems to be no sign of denial of service or injection attacks.



Logged errors

As shown in the figure below, we have logged 353 errors from March 28th to April 11th.



338 of these errors were bad requests from the client, while the rest of the errors (15) were server errors. Most of the server errors were logged during a crash in our production environment which was not related to the bad requests received by the client. Though, we also detected errors where our application could not decode a client's payload containing special characters as shown in the figure below.



This issue was resolved on the same day that these errors were logged.