Operator API Security Analysis

As identified in the project requirements, the APIs must be secure. Arguably, this is not as important as the customer API, since this returns readily available information such as company details and the modes of transport they use whereas the customer API returns personal details which should be protected. As we begin to expand the function of our API, however, to include the possibility of post requests to change the data, this becomes much more important.

The first vulnerability that I identified was the possibility of a man in the middle attack, with some third-party gaining access to the data being transferred from the API to the client. To combat this, we hosted the API using Pythonanywhere, forcing the client to use HTTPS when making a request which includes the TLS certificate to ensure a secure, encrypted connection.

The second vulnerability that I considered was the fact that anybody could access the API – there was no authentication. Therefore, we used the Django rest framework authorization protocols to prevent access to the API without a verified token. If the API is queried from any application other than the operator query tool we have built, the request will not be authorized. This then leads to the problem of ensuring that the users of the query tool are well authenticated. We have used the built in Django AllAuth framework to ensure secure logins. I will be looking into the possibility of using email to verify an account.

Also, I would suggest that when we begin to use the API to connect one account to another, the linking of accounts must be very secure. We need to know that both accounts belong to the same user. So, if account A wants to link with account B, we need to ensure that an email is sent to the address of account B, asking them to verify that account A is indeed their account.

The next issue which I identified was the issue of a CSRF attack. So, when a user is logged into the query tool, they can perform both queries and then log out. However, there was a flaw in the code which allowed access afterwards to these queries when the user had logged out. If somebody were to have access to the history, or maybe the favourites, they could go back to the query that the use was viewing earlier and see the data, even though they are not logged in. This is a serious security flaw, which I easily fixed in the templates by only displaying the data if the user is authenticated.

In terms of SQL injection attacks, I am happy that this should not cause a large issue. Django uses parameterization to automatically generate SQL to manipulate the database. There are no sections in our code where we use raw SQL so if someone were to inject it into the form, it would not cause an issue.

Another issue we have considered is the possibility of a denial-of-service attack. This is perhaps especially worrying since pythonanywhere can be somewhat temperamental and a relatively small amount of traffic may overwhelm the server. To combat this, we have implemented the use of Django throttle to limit the number of requests per day.

In the case that there were to be unidentified flaws exploited by attackers, we will have a log of the traffic to the API. This is because pythonanywhere automatically keeps access logs. Therefore, we will be able to identify the IP address of any malicious third parties and put them on a blacklist.