Time-based test for blind SQL injections

Detectify scanner performs fully automated tests to identify vulnerabilities on web applications. Some vulnerabilities can only be spotted by treating the web application as a black-box, and thus by analysing its *response* to a controlled set of sollicitations (i.e. *http requests*).

Blind SQL injections are one of such vulnerabilities and the Detectify scanner implements a time-based test for it[1]. In short, a time based blind SQL injection is a type of SQL attack that asks to a database behind a web application to sleep for a specified amount of time before responding: if the application is vulnerable, then there will be a considerable delay in the response, otherwise there will be no delay. Due to its time-based nature though, such test is prone to false positives - i.e. results that indicate that a vulnerability exists which in reality does not - and that is why we need your help.

In order to be able to consistently deliver high quality results to our users in fact, we must guarantee a false positive rate of less than 10^{-6} for each test. Moreover, because we run such tests against thousand of urls in each application, we need to make sure that the length of each test is limited to a few seconds in order to keep the overall length of the scan under control.

We would thus like you to write a time-based algorithm to identify urls vulnerable to blind SQL injections that completes within a few seconds and that returns less than one false positive result in ten thousand.

In the next page, you will find a detailed explanation how to do a blind SQL injection attack so that you will be able to try it on a web application that we have built for you, that mimics the behaviour of a real application on the internet. By asking the database behind such web application to sleep for a given amount of time, you will get responses that will or will not be affected by such sleep time, thus revealing whether the application is vulnerable or not. Beware that the application also mimics the network delay which typically overlaps to any internet communication between a client and a server, thus introducing noise to all response times.

We would like you to get back to us with:

- a summary of how you interpreted the problem at hand and your approach to it
- a log/dataset of urls and response times that you measured to understand the network delay and a short description of it
- a testing procedure that evaluates response times and produces a false positive rate smaller than one in ten thousand
- a description of the algorithm that you built and the reasoning behind your choice
- readable and well documented code that you use to solve this challenge

and any additional information you think is worth sharing. Good luck!

[1] cf. https://support.detectify.com/customer/en/portal/articles/2529264-blind-sql-injections

What a blind SQL injection attack looks like

Very often websites use template pages that are filled with content according to what users request. Such content can, for instance, be stored in a database from which it is fetched on-the-fly when needed.

Let's assume, for instance, that the website `www.example.com` has a database where several items are stored with a unique id[2]. If a user, for instance, navigates to

```
www.example.com/item?id=1
```

the server will execute the following SQL query towards the database

```
SELECT content FROM t_items WHERE id = 1
```

which will instruct the database to fetch the columns called *content* that correspond to the item with *id = 1* in the table called *t_items*.

If an attacker wants to check if the website is vulnerable, he can change the *?id=1* parameter in the URL to *?id=2* and check the response. If some content is returned then the website is vulnerable, whereas if an error is returned the website is not vulnerable.

If the website is vulnerable, the attacker can try to inject an SQL instruction instead of an id number. For example, he can try to inject a SQL command that forces the database to sleep before returning, as follows:

```
www.example.com/item?id=SLEEP(3)
```

which will execute the query

```
SELECT content FROM t_items WHERE id = SLEEP(3)
```

which will make the server wait for 3 seconds before returning a page or a some error. Irrespective of what the server returns though, observing a delay of 3 seconds in the response is the prove that the website is vulnerable. Note that because of network delay, the response would actually returned after a time of 3 second + network delay (i.e. noise).

[2] Note that this is a made up website that does not exists on the internet, and serve solely the purpose of depicting how a blind SQL injection attack looks like.

How to launch an attack against our web application

To facilitate your work, we built a python web application that you can use as a test bed for this challenge. The application serves several pages, some of which are vulnerable and some of which are not vulnerable to blind SQL injections. Moreover, a delay, which typically overlaps to any internet communication between a client and a server, is randomly added to each response.

The application can be started by invoking the python script `app.py`[3].

Once started, the application can be reached at the url `http://localhost:5000`.

Vulnerable pages can be reached at urls like

```
http://localhost:5000/vulnerable/1/page?id=1
```

where the number in the last path element of the url, points to one specific page and can assume any numerical values. For example, the following are all legitimate vulnerable urls:

```
http://localhost:5000/vulnerable/1/page?id=1
http://localhost:5000/vulnerable/7/page?id=1
http://localhost:5000/vulnerable/12/page?id=1
```

Non-vulnerable pages, instead, are reachable at urls like

```
http://localhost:5000/safe/1/page?id=1
```

where the url structure is the same as the vulnerable one.

Instead of using a browser, you can for example use the python requests module[4] to send requests to the application and measure response times.

[3] A reference to how to run a Flask application can be found at http://flask.pocoo.org/docs/1.0/quickstart/
[4] http://docs.python-requests.org/en/master/