Exploiting (pretty) blind SQL injections

Published: 2016-02-15
Last Updated: 2016-02-15 22:53:40 UTC
by Bojan Zdrnja (Version: 1)

0 comment(s)

Although a lot has been written about SQL injection vulnerabilities, they can still be found relatively often. In most of the cases I've seen in last couple of years, I had to deal with blind SQL injection vulnerabilities. Typically, they can be exploited relatively easy, by carefully modifying the resulting SQL queries and deducing the answer – either by monitoring the resulting web page or, for example, by measuring the time it took for the answer to arrive.

Long time ago (phew, it's been almost 7 years!) I wrote a diary where I gave an example of exploiting a blind SQL injection vulnerability that needed a specific value in a parameter (see https://isc.sans.edu/diary/Advanced+blind+SQL+injection+%28with+Oracle+examples%29/6409). In this particular case, the SQL query's result had to be either true or false (literally) – by careful modification of the query this can be exploited to retrieve any data from the database (as long as the user we run as has permissions to access the required data, of course).

This time I stumbled upon a bit more difficult SQL injection. I was able to control a single parameter that was used in a SQL query through a request such as the one below:

```
http://10.10.10.10/application.aspx?queueID=743994
```

When the URL was opened regularly, in a browser, it was supposed to list certain transactions belonging to the numerical `queueID` in shown in the URL. The first issue was that the `queueID` I had to work with had no results! It just returned an empty page (and there was no Direct Object Reference here, I had to use that particular `queueID`).

What's the problem you might say? Simply put `743994 OR 1=1` and off you go. Well, it wasn't that easy – due to a complex SQL query in the backend, as well as application handling of the received data I did not manage to exploit it with such simple queries. A `UNION` didn't work either – the application expected exactly certain values in the columns (not only column types) and all I could get with both of these cases was a nice, generic error screen (hey, no insufficient error handling either!).

The idea here was to prove that this is exploitable – and although it looks a bit more complex, it is certainly exploitable – here's one way (if you have idea of other ways on how to exploit this please do share!).
So, to prove that there was a SQL injection vulnerability, the following two queries can satisfy our needs:

```
queueID = 743994 AND 1=1
queueID = 743994 AND 1=2
queueID = 743995
queueID = 743994'
```

The first 3 queries all return a no results page. The last one returns the generic error page due to the SQL error that happened in the background. A tough cookie.

As with any blind SQL injection vulnerability, we need a true/false cases. And if you take a look we actually have them: the true case can be the no results page, while the false case can be the error page. The only question that remains is how to provoke the error page – it's easy to do manually by entering a ' character, but how to do it in SQL, programmatically?

Division by zero come to the rescue! If we divide something by zero, we can cause an SQL error, which will result with the generic error page.

Our exploitation now becomes the following:

```
queueID = 743994 AND 1 = 1 / (select case when substr(banner, 1, 1) = 'A' then 1
else 0 end from (select banner from v$version where banner like '%Oracle%'))
```

As in the previous diary, the query takes the database banner from `v$version` (where it has string Oracle in it). Then, from that line the first character is examined (specified by the `substr()` call) and compared to the letter 'A'. If it is 'A', the query returns 1, otherwise it returns 0.

This will either result in `1 = 1 / 1` or `1 = 1 / 0`. The former will display the no results page while the latter will display the error page. Game over!
In this particular case the goal was to show what an attacker can do with the vulnerability. While it was relatively easy to confirm its existence, by showing that arbitrary data can be retrieved from the database, the resulting risk indeed gets pretty high.

Have more ideas on how to exploit this? Let us know!

--
Bojan
@bojanz
INFIGO IS

Keywords: blind sql injection

0 comment(s)

previous

Top of page

Diary Archives