

[Open in app](#)

Medium

 Search

N

Be part of a better internet. [Get 20% off membership for a limited time](#)

What is a Second-Order SQL Injection and how can you exploit it successfully?

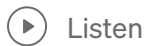
Second-Order SQL Injection with a demonstration



Fiddly Cookie · Follow

Published in InfoSec Write-ups

7 min read · Jan 24, 2019

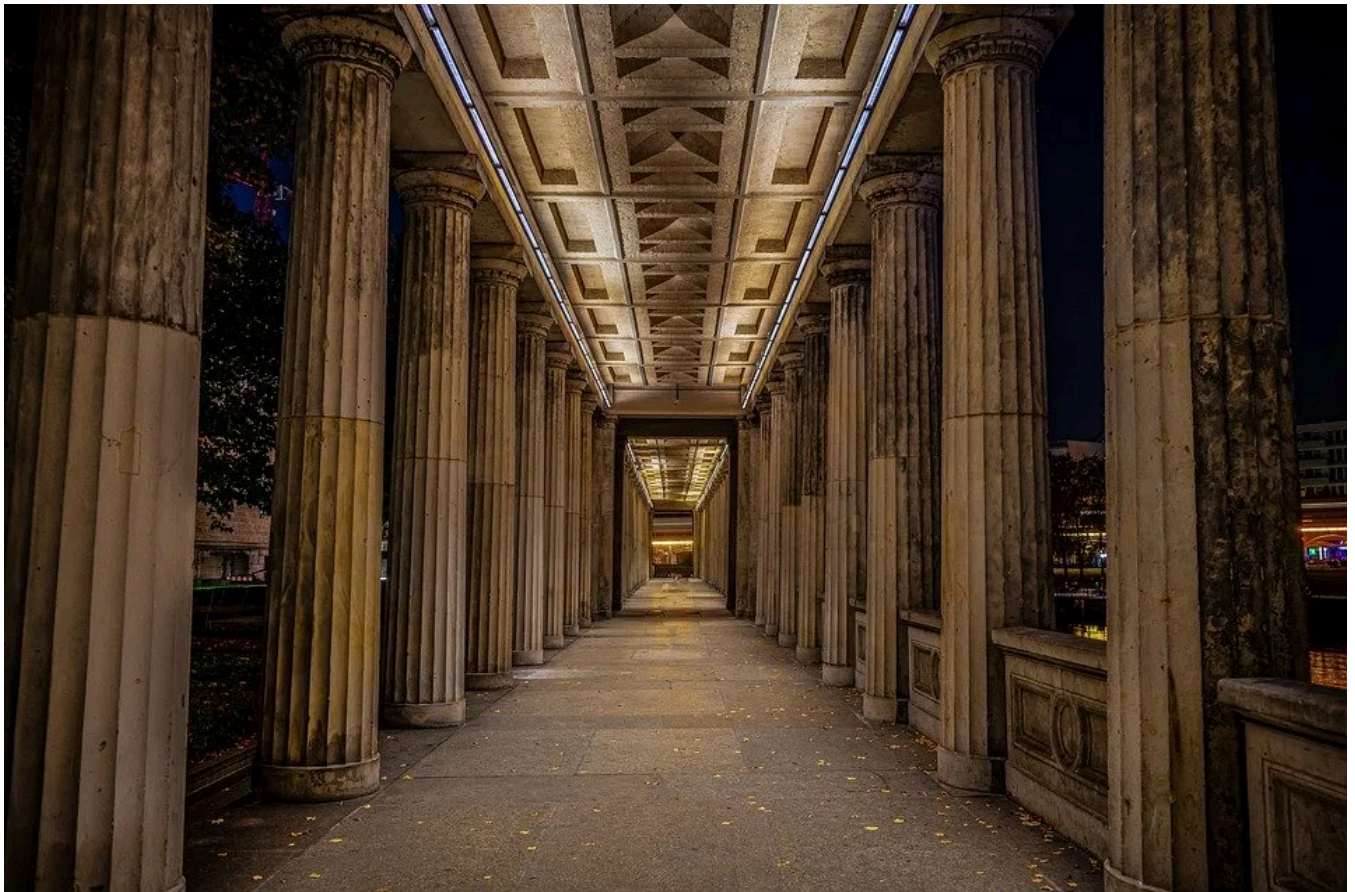


Listen



Share

... More



Credit: pixabay.com

What is SQL Injection

SQL Injection — the process of injecting SQL language code within data requests that result in application backend database server either surrendering confidential data or cause the execution of malicious scripting content on the database that could result in a complete compromise of the host.

Understanding Second-Order Code Injection

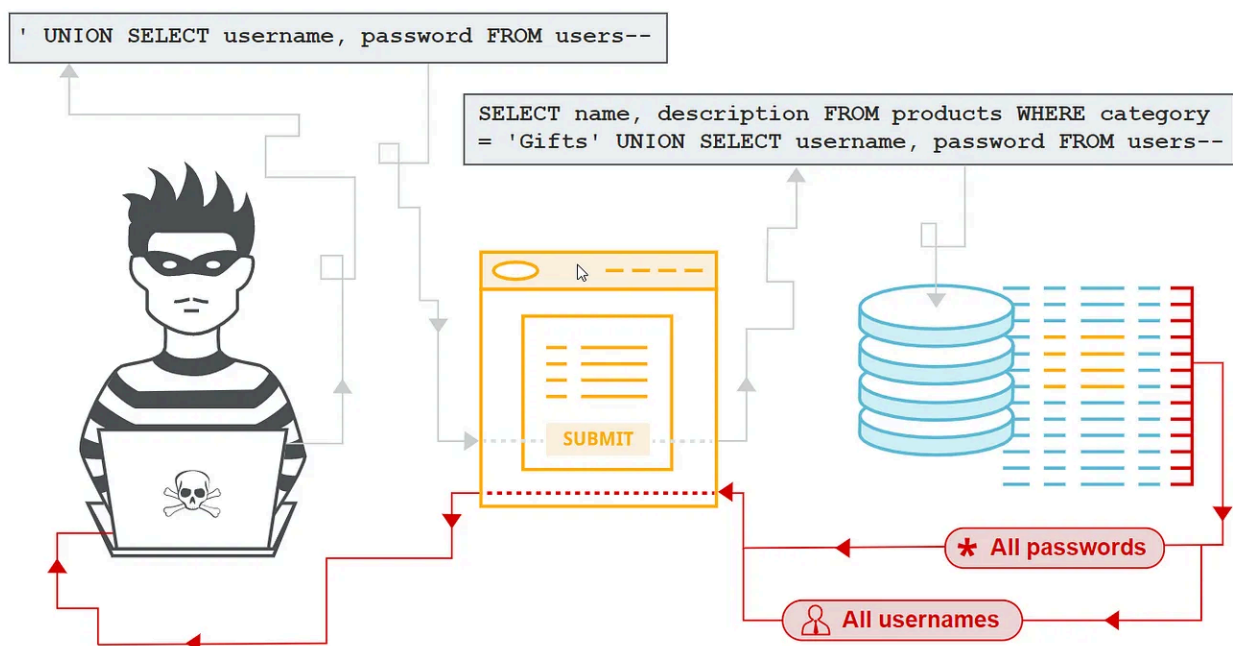
Imagine a scenario wherein a malicious code injected into an application by an attacker which does not get immediately get executed in the application.

Yes, you read that right. It's a familiar story and it usually goes like this, user-provided data becomes a threat when it is utilized by the application or any other application wherein the injected code provided by the attacker gets activated resulting in successful exploitation.

First -order and Second-order SQL Injection differ primarily in the way that the attacker can simply enter a malicious string and cause the modified code to be executed immediately.

See the difference?

The attacker injects into persistent storage (such as a table row) which is deemed as a trusted source. An attack is subsequently executed by another activity.



Credit: portswigger.net

Testing Challenge

The attacking nature of common code injection allows an attacker to discover the vulnerability by observing the application response.

Testing for Second Order SQL Injection is slightly difficult because it requires the attacker to have the knowledge of backend operation of the application.

How can you beat that?

Automated web-application assessment tools are not adequate to identify these vulnerabilities. An automated tool is not smart enough to identify the change in application behavior in any of the subsequent responses caused by the malicious injection in one of the previous queries.

What makes an application vulnerable to Second-order SQL Injection

This kind of vulnerability happens because a good programmer maybe will patch his code to prevent SQL injections in forms where the user can input something BUT he will not do the same thing where a user doesn't have any sort of interaction with the application database.

Exploit Scenario

A second-order SQL Injection, on the other hand, is a vulnerability exploitable in two different steps:

1. Firstly, we STORE a particular user-supplied input value in the DB and
2. Secondly, we use the stored value to exploit a vulnerability in a vulnerable function in the source code which constructs the dynamic query of the web application.

So let's get down to business and look at how a vulnerable application could be exploited in more detail with the help of a hypothetical scenario:

Example 1

```
CREATE TABLE USERS ( userId serial PRIMARY KEY, firstName TEXT )
```

Suppose you have some SAFE code like this, receiving *firstName* from a form:

```
$firstname = someEscapeFunction($_POST["firstName"]);
```

```
$SQL = "INSERT INTO USERS (firstname) VALUES ('{$firstName}');"
someConnection->execute($SQL);
```

So far so good, assuming that someEscapeFunction() does a fine job. It isn't possible to inject SQL. If I would now send my payload as a value for firstname, you wouldn't mind:

```
Payload : bla'); DELETE FROM USERS; //
```

Now, suppose somebody on the same system wants to transport firstName from USERS to SOME, and does that like this:

```
$userid = 42; $SQL = "SELECT firstname FROM USERS WHERE (userId=
{$userid})"; $RS = con->fetchAll($SQL); $firstName = $RS[0]
["firstName"];
```

And then inserts it into SOME table without escaping:

```
$SQL = "INSERT INTO SOME VALUES ('{$firstName}');";
```

Malicious query becomes like this:

```
INSERT INTO SOME VALUES (' bla'); DELETE FROM USERS; //
```

At this point you realise that if the firstname contains some delete command, it will still be executed.

Example 2

It could be possible to exploit some functions that don't need user input and uses data already saved in the DB, that retrieves when needed. The password reset functionality!

A victim user "User123" could be registered on the website with a very strong and secure password but we still really want to get his account. In a second order SQL Injection we should be able to do something like:

Register a new account. We want to name this new user as “User123’ — “ and password “UserPass@123”

Payload: “User123’ — “

Then we can reset our password and set a new one in the appropriate form.

The legit query will be:

```
$pwdreset = mysql_query(“UPDATE users SET password=’getrekt’ WHERE username=’User123’ — ‘ and password=’UserPass@123’”);
```

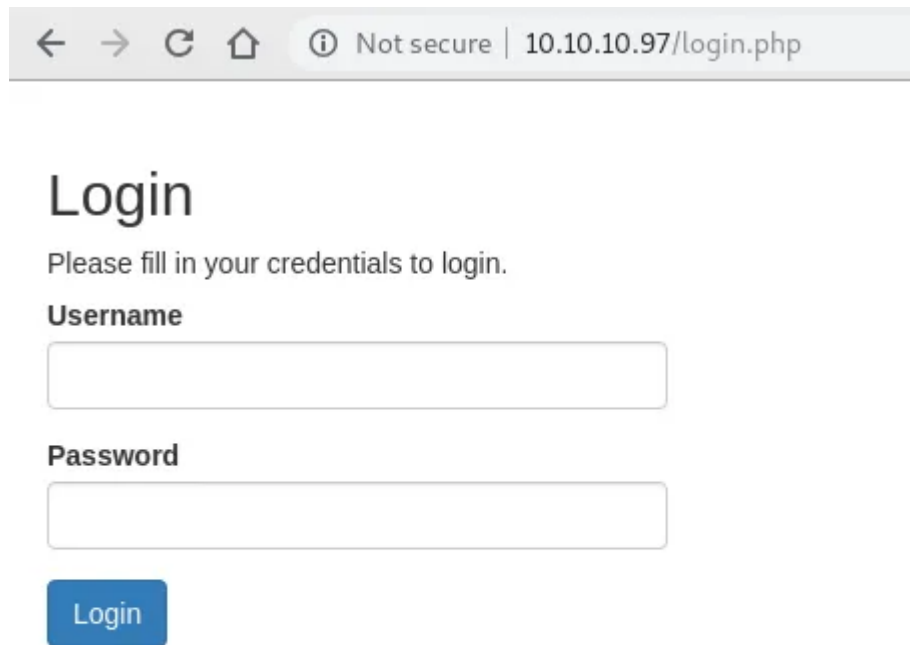
BUT since — is the character used to comment in SQL, the query will result being this:

```
$pwdreset = mysql_query(“UPDATE users SET password=’getrekt’ WHERE username=’User123’”);
```

And boom! You’re there. This will set a new password chosen by us for the victim user account!

Demonstration

I was working on SecNotes machine on HTB and encountered a login form as shown in Figure 1. Gotta try SQL Injection, right? Duh! I tried inserting SQL injection queries in the login parameters but nothing showed up.



← → ↻ 🏠 ⓘ Not secure | 10.10.10.97/login.php

Login

Please fill in your credentials to login.

Username

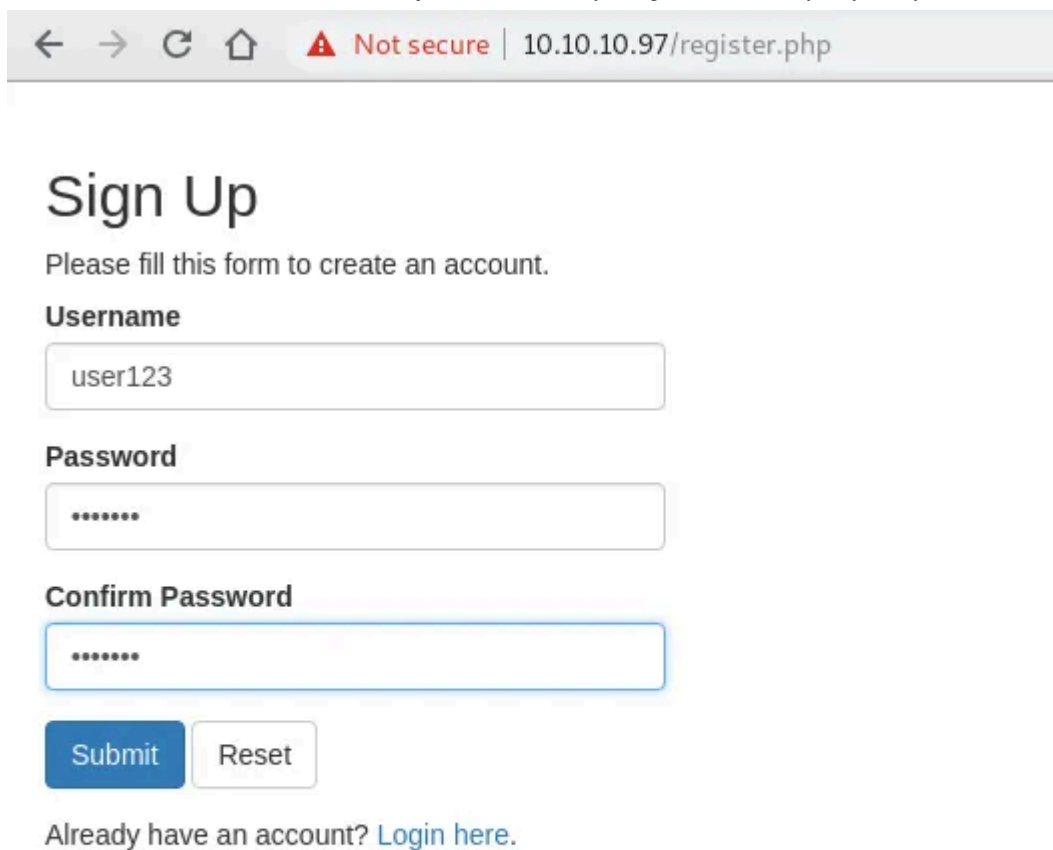
Password

Login

Don't have an account? [Sign up now.](#)

Figure 1

I created a user user123 and logged into the account wherein I could see some notes as shown in Figure 2 and Figure 3.



← → ↻ 🏠 ⚠ Not secure | 10.10.10.97/register.php

Sign Up

Please fill this form to create an account.

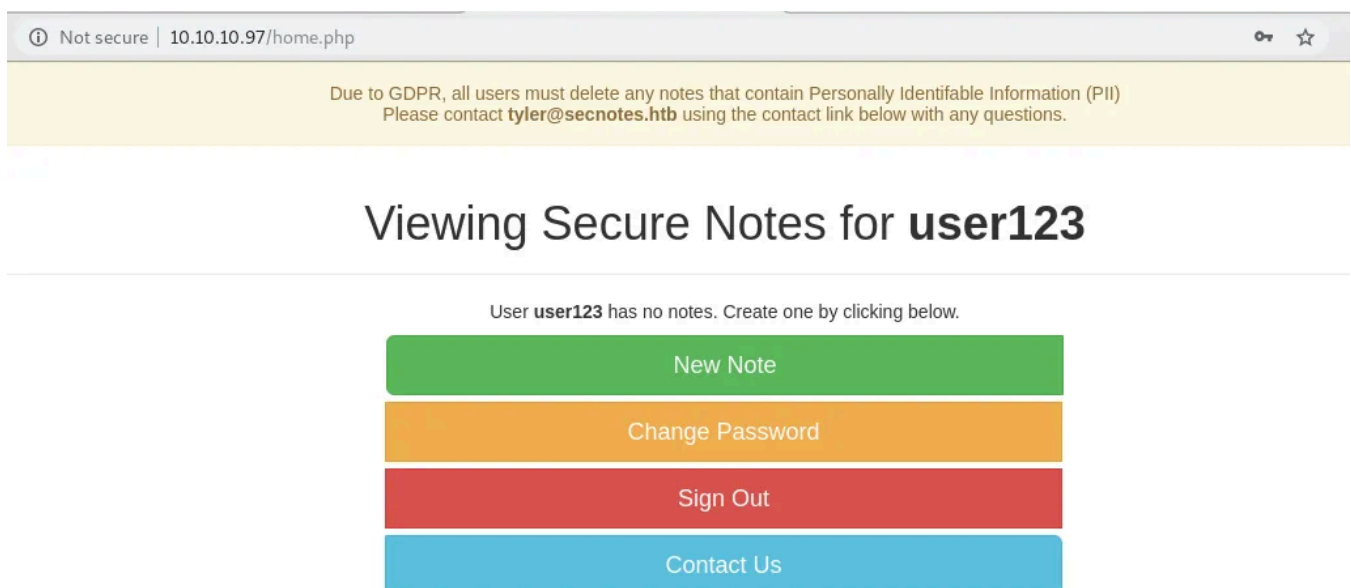
Username

Password

Confirm Password

Already have an account? [Login here.](#)

Figure 2



ⓘ Not secure | 10.10.10.97/home.php 🔑 ☆

Due to GDPR, all users must delete any notes that contain Personally Identifiable Information (PII)
Please contact tyler@secnotes.htb using the contact link below with any questions.

Viewing Secure Notes for **user123**

User **user123** has no notes. Create one by clicking below.

Figure 3

If we recall how SQL Injection exploits works, we STORE a particular value in the DB and the stored value becomes a part of the query in an unfiltered or bugged function in the source code of the web application.

What if the application is fetching the notes from the database using the username of the application. Let's create a user with a username containing ' and hoping that we might encounter an SQL error

Figure 4

Woah! The server responds with 500 internal server error as shown in Figure 5. Once I was able to make the server respond with an error (mostly HTTP 500 status), I had to confirm that it is the SQL command that is causing the error and not something else.

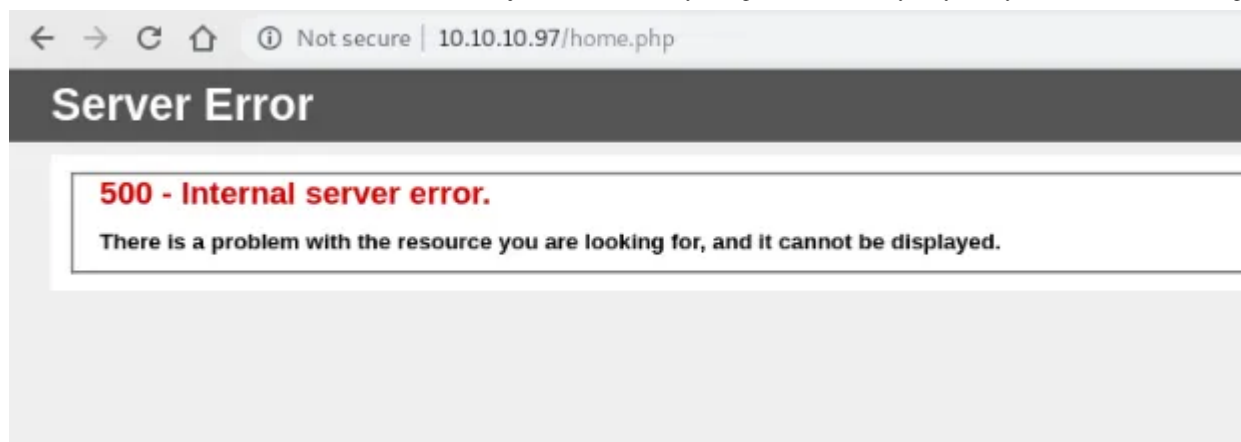


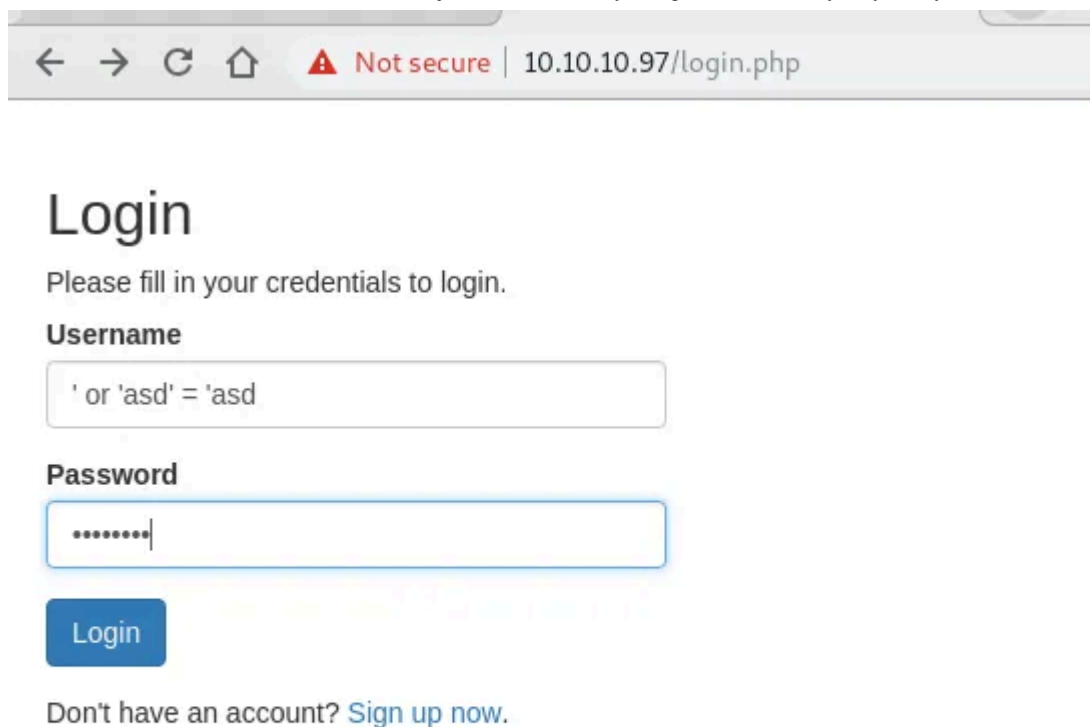
Figure 5

I created an account with username 'or 'asd'='asd as shown in Figure 6. So, the username 'or 'asd'='asd gets STORED in the database.

A screenshot of a web browser window showing a "Sign Up" form. The address bar shows "10.10.10.97/register.php" with a "Not secure" warning. The form has a title "Sign Up" and a subtitle "Please fill this form to create an account." There are three input fields: "Username" with the value "' or 'asd' = 'asd", "Password" with masked characters ".....", and "Confirm Password" with masked characters ".....". Below the fields are "Submit" and "Reset" buttons. At the bottom, there is a link "Login here." for users who already have an account.

Figure 6

Then, I logged into the account with the same username as shown in Figure 7.



← → ↻ 🏠 ⚠ Not secure | 10.10.10.97/login.php

Login

Please fill in your credentials to login.

Username

Password

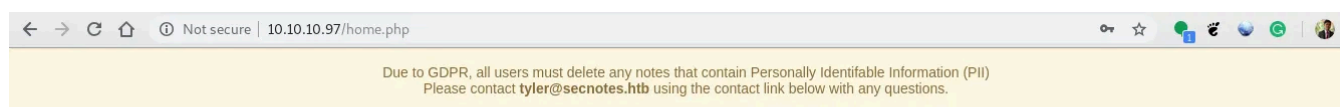
Login

Don't have an account? [Sign up now.](#)

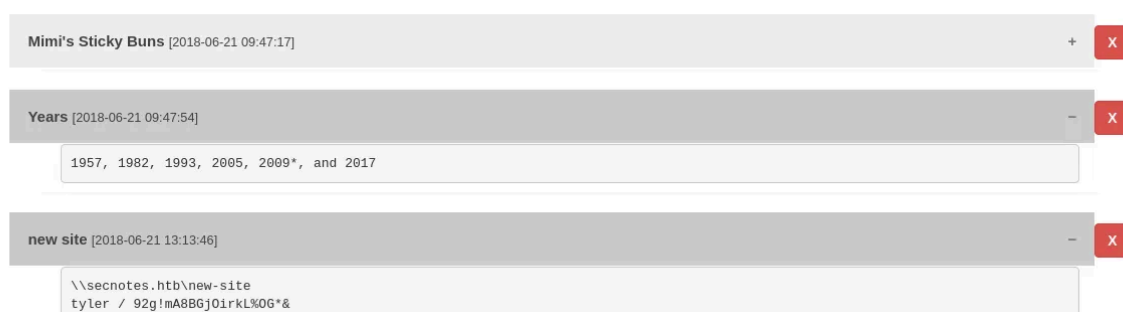
Figure 7

Bingo! Now I was able to see three notes from the database as shown in Figure 8. It was confirmed that the heading was causing this 2nd order SQL injection vulnerability. The dynamic query that was constructed would look something like this,

```
SELECT * from notes WHERE username = '' or 'asd' = 'asd';
```



Viewing Secure Notes for ' or 'asd' = 'asd'



Mimi's Sticky Buns [2018-06-21 09:47:17] + X

Years [2018-06-21 09:47:54] - X

1957, 1982, 1993, 2005, 2009*, and 2017

new site [2018-06-21 13:13:46] - X

\\secnotes.htb\\new-site
tyler / 92g!mABBGj01rkL%0G*&

Figure 8

Attack Probability

The success rate of identifying a classical (first-order) SQL Injection is common in comparison with the second-order SQL injection. The First-order Injections often referred to as ‘low hanging fruit’ can be observed directly whereas the relative probability of second-order SQL Injection is low.

The Second-order SQL Injection attack has to be performed “blind” in a majority of the cases because the attacker performs the attack on the backend functionality without any prior knowledge of the system.

Protection Against Second-Order SQL Injection

Use a white-list approach to sanitizing data (i.e. disallow everything by default, and explicitly enumerate data characters that are allowed or deemed “safe”).

Beware that data marked “SAFE” for one application may not be safe for another application/component.

- Each application that retrieves stored data (especially if the data is likely to have been supplied by users) must apply its own data sanitization processes before processing it further. Before any user-supplied data should undergo sanitization process before it is being processed further
- All data processed within and between the application components should be validated.

Language specific recommendations:

- Java EE — use `PreparedStatement()`
- .NET — use parameterized queries like `SqlCommand()` or `OleDbCommand()`
- PHP — use PDO with strongly typed parameterized queries (using `bindParam()`)
- Hibernate — use `createQuery()` (called named parameters in Hibernate)
- SQLite — use `sqlite3_prepare()`

Now I’m going to stop right there and leave you to discover more about Second-order SQL Injection.