

SQL

INJECTION

ATTACK

INVESTIGATION

WITH

EXAMPLE

CODE

Before we go to investigation, here are the SQL injection code examples.

Example 1: Using SQLi to Authenticate as Administrator

This example shows how an attacker can use SQL injection to circumvent an application's authentication and gain administrator privileges.

Consider a simple authentication system using a database table with usernames and passwords. A user's POST request will provide the variables `user` and `pass`, and these are inserted into a SQL statement:

```
sql = "SELECT id FROM users WHERE username='" + user + "' AND password='" + pass + "'"
```

The problem here is that the SQL statement uses concatenation to combine data. The attacker can provide a string like this instead of the `pass` variable:

```
password' OR 5=5
```

The resulting SQL query will be run against the database:

```
SELECT id FROM users WHERE username='user' AND password='pass' OR 5=5'
```

Because `5=5` is a condition that always evaluates to true, the entire `WHERE` statement will be true, regardless of the username or password provided.

The `WHERE` statement will return the first ID from the users table, which is commonly the administrator. This means the attacker can access the application without authentication and has administrator privileges.

A more advanced form of this attack is where the attacker adds a code comment symbol at the end of the SQL statement, allowing them to further manipulate the SQL query. The following will work in most databases including MySQL, PostgreSQL, and Oracle:

```
' OR '5'='5' /*
```

Example 2: Using SQLi to Access Sensitive Data

In this example, the following code obtains the current username, and searches for items matching a certain item name, where the owner is the current user.

```
...
```

```
string userName = ctx.getAuthenticatedUserName();
```

```
string query = "SELECT * FROM items WHERE owner = ""  
+ userName + "" AND itemname = ""  
+ ItemName.Text + """;
```

...

This code has the same weakness as in the previous example, the use of concatenation. After combining the username and item name, the code creates the following query:

```
SELECT * FROM items  
WHERE owner =  
AND itemname = ;
```

If the attacker provides the following string for `itemname`:

```
Widget' OR 5=5
```

The SQL statement becomes:

```
SELECT * FROM items  
WHERE owner = 'Izzmier'  
AND itemname = 'Widget' OR 5=5';
```

Which is the same as: `SELECT * FROM items;`

This means the query will return the data of the entire table, giving the attacker unauthorized access to sensitive data.

Example 3: Injecting Malicious Statements into Form Field

First Name *

Last Name *

Email *

Subject of Your Inquiry *

Inquiry *

This is a simple SQL injection attack based on user input. The attacker uses a form that requires first name and last name as inputs. The attacker inputs:

- First name: `malicious'ex`
- Last name: `Izzmier`

The attacker's first name variable contains a malicious expression, which we denoted as 'ex. The SQL statement that processes the form inputs looks like this:

```
SELECT id, firstname, lastname FROM authors
```

Once the attacker injects a malicious expression into the first name, the statement looks like this:

```
SELECT id, firstname, lastname FROM authors WHERE firstname = 'malicious'ex' and lastname = 'Izzuddin'
```

The database identifies incorrect syntax due to the single apostrophe and tries to execute the malicious statement.

We move to investigation.

Example: Web Server

Apache HTTP Server provides two main log files –

- *access.log*
- *error.log*

The access.log records all requests for files. If a visitor request `www.izzmier.com/main.php`, the following entry will be added to the log file:

```
88.54.124.17 - - [23/Jun/2023:07:44:08 +0100] "GET /main.php HTTP/1.1" 200 203 "-" "Mozilla/5.0 (Windows NT6.0; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0"
```

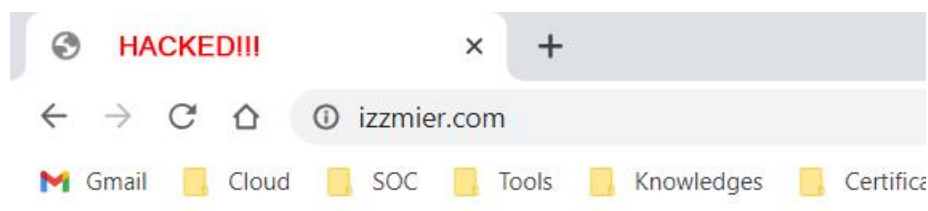
The above log shows that a visitor with an IP address `88.54.124.178` requested the *main.php* file on `23 Jun 2023:07:44:08` and the request was *successful*.

This information might not be too interesting, but what if the log file has shown that a visitor with IP `88.54.124.178` requested the *dump_database.php* file on `23 Jun 2023:07:44:08` and the request was successful? In the absence of that log file, you might have never known that someone discovered and ran a secret or restricted script that you have on your website and that dumps the database.

Having established that a log file is a critical asset, let's look at an everyday example of how a log file would help identify when, how and by whom a website was hacked.

Investigation

Let's assume that a website that we administer got defaced and assume that the site was a simple and up-to-date WordPress website running on a fully patched Ubuntu Server.



HACKED!!!

After reaching out for help, the forensic team took the server offline to be able to proceed with the investigation.

The server is isolated to preserve the current state of the system and its logs, block remote access to the attacker (in the case a backdoor was installed), as well as prevent interaction with any other machines on the network.

To identify malicious activity on the web server, you often create a forensically sound copy of the server and then proceed with the investigation. However, since there are no plans to pursue legal action against the attacker, in this case, the forensic team can work on original data.

Evidence to Look for in an Investigation.

To start an investigation, the investigator needs to identify what evidence to look for. Usually, evidence of an attack involves direct access to hidden or unusual files, access to the administration area with or without authentication, remote code execution, SQL injection, file inclusion, cross-site scripting (XSS) and other unusual behaviour that might indicate vulnerability scanning or reconnaissance.

Let us assume that in our example, the web server access.log is available.

```
root@secureserver:/var/log/apache2# less access.Log
```

The access.log tends to be quite a large file, often containing thousands of recorded requests.

```
84.55.41.57 - - [23/Jun/2023:20:21:56 +0100] "GET /izzmier/index.php HTTP/1.1" 200 3804 "-"
"Mozilla/5.0 (Windows NT 6.0; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0"
84.55.41.57 - - [23/Jun/2023:20:21:56 +0100] "GET /izzmier/assets/js/skel.min.js HTTP/1.1"
200 3532 "http://www.example.com/izzmier/index.php" "Mozilla/5.0 (Windows NT 6.0; WOW64;
rv:45.0) Gecko/20100101 Firefox/45.0"
84.55.41.57 - - [23/Jun/2023:20:21:56 +0100] "GET /izzmier/images/pic01.jpg HTTP/1.1" 200
9501 "http://www.example.com/izzmier/index.php" "Mozilla/5.0 (Windows NT 6.0; WOW64;
rv:45.0) Gecko/20100101 Firefox/45.0"
84.55.41.57 - - [23/Jun/2023:20:21:56 +0100] "GET /izzmier/images/pic03.jpg HTTP/1.1" 200 5593
"http://www.example.com/izzmier/index.php" "Mozilla/5.0 (Windows NT 6.0; WOW64; rv:45.0)
Gecko/20100101 Firefox/45.0"
```

Checking every single line would be impractical, so we need to filter out data that would most probably be of no interest. That usually includes resources such as images and CSS stylesheets. Some investigators also prefer to strip out JavaScript files too.

In this case, however, since the website is running the WordPress web application, we will use a slightly different approach. Instead of ruling out some data, we will filter *access.log* for WordPress-specific characteristics.

```
root@secureserver:~#cat /var/log/apache2/access.Log | grep -E "wp-admin|wp-login|POST /"
```

The above command filters *access.log* and shows only records with strings containing *wp-admin*, which is the default administration folder of WordPress, *wp-login*, which is part of the login file of WordPress (*wp-login.php*), and finally, **POST**, which will show HTTP requests sent to the server using the POST method, which are most likely login form submissions.

The output returns several results. After sifting through them, we'll concentrate on the following single record:

```
84.55.41.57 - - [24/Jun/2023:06:52:07 +0100] "GET /wordpress/wp-admin/ HTTP/1.1" 200 12349
"http://www.example.com/wordpress/wp-login.php" "Mozilla/5.0 (Windows NT 6.0; WOW64; rv:45.0)
Gecko/20100101 Firefox/45.0"
```

We see that the IP `84.55.41.57` accessed the WordPress administration interface successfully. Let's see what else the user with this IP address did. We'll use `grep` once again to filter the `access.log` with that IP.

```
root@secureserver:~#cat /var/Log/apache2/access.Log | grep 84.55.41.57
```

This results in the following interesting records.

```
84.55.41.57 - - [24/Jun/2023:06:57:24 +0100] "GET /wordpress/wp-login.php HTTP/1.1" 200 1568 "-"
84.55.41.57 - - [24/Jun/2023:06:57:31 +0100] "POST /wordpress/wp-login.php HTTP/1.1" 302 1150
"http://www.example.com/wordpress/wp-login.php"
84.55.41.57 - - [24/Jun/2023:06:57:31 +0100] "GET /wordpress/wp-admin/ HTTP/1.1" 200 12905
"http://www.example.com/wordpress/wp-login.php"
84.55.41.57 - - [24/Jun/2023:07:00:32 +0100] "POST /wordpress/wp-admin/admin-ajax.php HTTP/1.1"
200 454 "http://www.example.com/wordpress/wp-admin/"
84.55.41.57 - - [24/Jun/2023:07:00:58 +0100] "GET /wordpress/wp-admin/theme-editor.php HTTP/1.1"
200 20795 "http://www.example.com/wordpress/wp-admin/"
84.55.41.57 - - [24/Jun/2023:07:03:17 +0100] "GET /wordpress/wp-admin/theme
editor.php?file=404.php&theme=twenty-sixteen HTTP/1.1" 200 8092
"http://www.example.com/wordpress/wp-admin/theme-editor.php"
84.55.41.57 - - [24/Jun/2023:07:11:48 +0100] "GET /wordpress/wp-admin/plugin-install.php
HTTP/1.1" 200 12459 "http://www.example.com/wordpress/wp-admin/plugin-install.php?tab=upload"
84.55.41.57 - - [24/Jun/2023:07:16:06 +0100] "GET /wordpress/wp-admin/update.php?action=install-
plugin&plugin=file-manager&_wpnonce=3c6c8a7fca HTTP/1.1" 200 5698
"http://www.example.com/wordpress/wp-admin/plugin-install.php?tab=search&s=file+permission"
84.55.41.57 - - [24/Jun/2023:07:18:19 +0100] "GET /wordpress/wp-
admin/plugins.php?action=activate&plugin=file-manager%2Ffile-manager.php&_wpnonce=bf932ee530
HTTP/1.1" 302 451 "http://www.example.com/wordpress/wp-admin/update.php?action=install-
plugin&plugin=file-manager&_wpnonce=3c6c8a7fca"
84.55.41.57 - - [24/Jun/2023:07:21:46 +0100] "GET /wordpress/wp-admin/admin-
ajax.php?action=connector&cmd=upload&target=11_d3AtY29udGVudA&name%5B%5D=r57.php&FILES=&_id=1460873
968131 HTTP/1.1" 200 731 "http://www.example.com/wordpress/wp-admin/admin.php?page=file-
manager_settings"
84.55.41.57 - - [24/Jun/2023:07:22:53 +0100] "GET /wordpress/wp-content/r57.php HTTP/1.1" 200
9036 "-"
84.55.41.57 - - [24/Jun/2023:07:32:24 +0100] "POST /wordpress/wp-content/r57.php?14 HTTP/1.1" 200
8030 "http://www.example.com/wordpress/wp-content/r57.php?14"
84.55.41.57 - - [24/Jun/2023:07:29:21 +0100] "GET /wordpress/wp-content/r57.php?29 HTTP/1.1" 200
8391 "http://www.example.com/wordpress/wp-content/r57.php?28"
84.55.41.57 - - [24/Jun/2023:07:57:31 +0100] "POST /wordpress/wp-admin/admin-ajax.php HTTP/1.1"
200 949 "http://www.mywebsite.com/wordpress/wp-admin/admin.php?page=file-manager_settings"
```

Let's analyse these records a bit further. The attacker accessed the login page.

```
84.55.41.57 - GET /wordpress/wp-login.php 200
```

The attacker submitted the login form (HTTP request using the POST method) and was redirected (302 HTTP status code).

```
84.55.41.57 - GET /wordpress/wp-login.php 302
```

The attacker was redirected to `wp-admin` (the WordPress dashboard), which means that authentication was successful.

```
84.55.41.57 - GET /wordpress/wp-admin/ 200
```

The attacker navigated to the theme editor.

```
84.55.41.57 - GET /wordpress/wp-admin/theme-editor.php 200
```

The attacker tried to edit the file *404.php*, which is a very common tactic used to inject malicious code into the file. The attacker most probably failed in doing so due to a lack of write permissions.

```
84.55.41.57 - GET /wordpress/wp-admin/theme-editor.php?file=404.php&theme= twenty sixteen 200
```

The attacker accessed the plugin installer.

```
84.55.41.57 - GET /wordpress/wp-admin/plugin-install.php 200
```

The attacker installed and activated the *file-manager* plugin.

```
84.55.41.57 - GET /wordpress/wp-admin/update.php?action=install-plugin&plugin= file-manager
&_wpnonce=3c6c8a7 fca 200
84.55.41.57 - GET /wordpress/wp-admin/plugins.php?action=activate&plugin=file-manager%2Ffile-
manager.php&_wpnonce=bf932ee530 200
```

The attacker used the file-manager plugin to upload *r57.php*, which is a PHP **web shell** script.

```
84.55.41.57 - GET /wordpress/wp-admin/admin-ajax.php?action=connector& cmd=
upload&target=l1_d3AtY29udGVudA&n ame%5B%5D=r57.php&FILES=&_ =1460873968131 200
```

The log indicates that the attacker ran the *r57* shell script. The query strings **?1** (the attacker ran *phpinfo();*) and **?28** (the attacker got a list of services) indicate navigation through the different sections of the shell script. It appears that they didn't find anything interesting.

```
84.55.41.57 - GET /wordpress/wp-content/r57.php 200
84.55.41.57 - POST /wordpress/wp-content/r57.php?1 200
84.55.41.57 - GET /wordpress/wp-content/r57.php?28 200
```

The attacker's last action was to edit the index file of the theme through the file-manager plugin and replace its contents with the word *HACKED!*

```
84.55.41.57 - POST /wordpress/wp-admin/admin-ajax.php 200 - http://www.
example.com/wordpress/wp-admin/admin.php?page=file-manager_settings
```

Based on the above information, we now have a timeline of the attacker's actions that led to the defacement of the website. However, there is a missing piece in the puzzle. How did the attacker get the login credentials in the first place or did they bypass authentication?

Assuming that we are certain that the administrator password was not leaked or brute-forced, let's go back and see if we can find anything regarding this matter.

The current access.log did not contain any clues on what might have happened. However, there is more than just the one access.log file that we can investigate. The Apache HTTP Server log rotation algorithm archives old log files. Listing the */var/log/apache2/* directory shows four additional log files.

First, we need to filter the logs to see if any actions were taken by the IP *84.55.41.57*. One of the logs was bombarded with records containing a lot of SQL commands that clearly indicate an **SQL injection** attack on what seems to be a custom plugin that works with the SQL server.


```

84.55.41.57- - [20/Jun/2023:08:22:13 0100] "GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=1 AND (SELECT 6810 FROM(SELECT
COUNT(*),CONCAT(0x7171787671,(SELECT (ELT(6810=6810,1))),0x71707a7871,FLOOR(RAND(0)*2))x FROM
INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) HTTP/1.1" 200 166 "-" "Mozilla/5.0 (Windows; U;
Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR 3.5.30729)"
84.55.41.57- - [20/Jun/2023:08:22:13 0100] "GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=(SELECT 7505 FROM(SELECT
COUNT(*),CONCAT(0x7171787671,(SELECT (ELT(7505=7505,1))),0x71707a7871,FLOOR(RAND(0)*2))x FROM
INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) HTTP/1.1" 200 166 "-" "Mozilla/5.0 (Windows; U;
Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR 3.5.30729)"
84.55.41.57- - [20/Jun/2023:08:22:13 0100] "GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=(SELECT CONCAT(0x7171787671,(SELECT
(ELT(1399=1399,1))),0x71707a7871)) HTTP/1.1" 200 166 "-" "Mozilla/5.0 (Windows; U; Windows NT
6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR 3.5.30729)"
84.55.41.57- - [20/Jun/2023:08:22:27 0100] "GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=1 UNION ALL SELECT
CONCAT(0x7171787671,0x537653544175467a724f,0x71707a7871), NULL, NULL-- HTTP/1.1" 200 182 "-"
"Mozilla/5.0 (Windows; U; Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR
3.5.30729)"

```

Let's assume that this plugin was created by copy-and-pasting some code that the system administrator found online. The script was meant to check user validity based on a given ID. The plugin had a form exposed on the main web page, which was sending an AJAX GET request to `/wordpress/wp-content/plugins/custom_plugin/check_user.php`.

When we analyse `check_user.php`, it is immediately obvious that the script is poorly written and vulnerable to an SQL injection attack.

```

<?php
//Include the WordPress header
include('/wordpress/wp-header.php');

global $wpdb;

// Use the GET parameter 'userid' as user input
$id=$_GET['userid'];

// Make a query to the database with the value the user supplied in the SQL statement
$users = $wpdb->get_results("SELECT * FROM users WHERE user_id=$id");
?>

```

The number of records in the access.log and the pattern indicate that the attacker used an SQL injection exploitation tool to exploit an SQL injection vulnerability. The logs of the attack that may look like gibberish, however, they are SQL queries typically designed to extract data via an SQL injection vulnerability. The exploitation tool tries various SQL injection techniques to find the database name, table name, and columns as part of the enumeration process.

We will not dig deeper into the SQL injection attack, or how to fix SQL injection vulnerabilities (for example, using prepared statements) as this is outside the scope of this article. However, the records in the log would resemble the following:

```
/wordpress/wp-content/plugins/my_custom_plugin/check_user.php?userid=-6859 UNION ALL SELECT
(SELECT CONCAT(0x7171787671,IFNULL(CAST(ID AS CHAR),0x20),0x616474686c76,IFNULL(CAST(display_name
AS CHAR),0x20),0x616474686c76,IFNULL(CAST(user_activation_key AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_email AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_login AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_nicename AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_pass AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_registered AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_status AS
CHAR),0x20),0x616474686c76,IFNULL(CAST(user_url AS CHAR),0x20),0x71707a7871) FROM wp.wp_users
LIMIT 0,1),NULL,NULL--
```

The above SQL code is a very strong indication that the WordPress database has been compromised and that all sensitive information in that SQL database has potentially been stolen.

Some questions remain, such as who was behind the attack. At this point, it is only possible to know the attacker's IP address. It is very difficult, and probably infeasible to attempt to attribute most attacks unless the attacker left concrete evidence that ties to a real person's identity. Bear in mind that attackers frequently make use of proxies and anonymity networks such as Tor to conduct most attacks to mask their real location.

The bottom line is that unsafe code that led to an SQL injection attack was present in a custom WordPress plugin. Had the site been tested for security vulnerabilities before being deployed in a production environment, it would have not been possible for the attacker to take advantage of the security vulnerability that caused the defacement.

The attacker in the above fictitious example was very sloppy and left a significant amount of evidence and tracks, which made the investigation very easy. Bear in mind, however, that it is not always the case, especially when dealing