

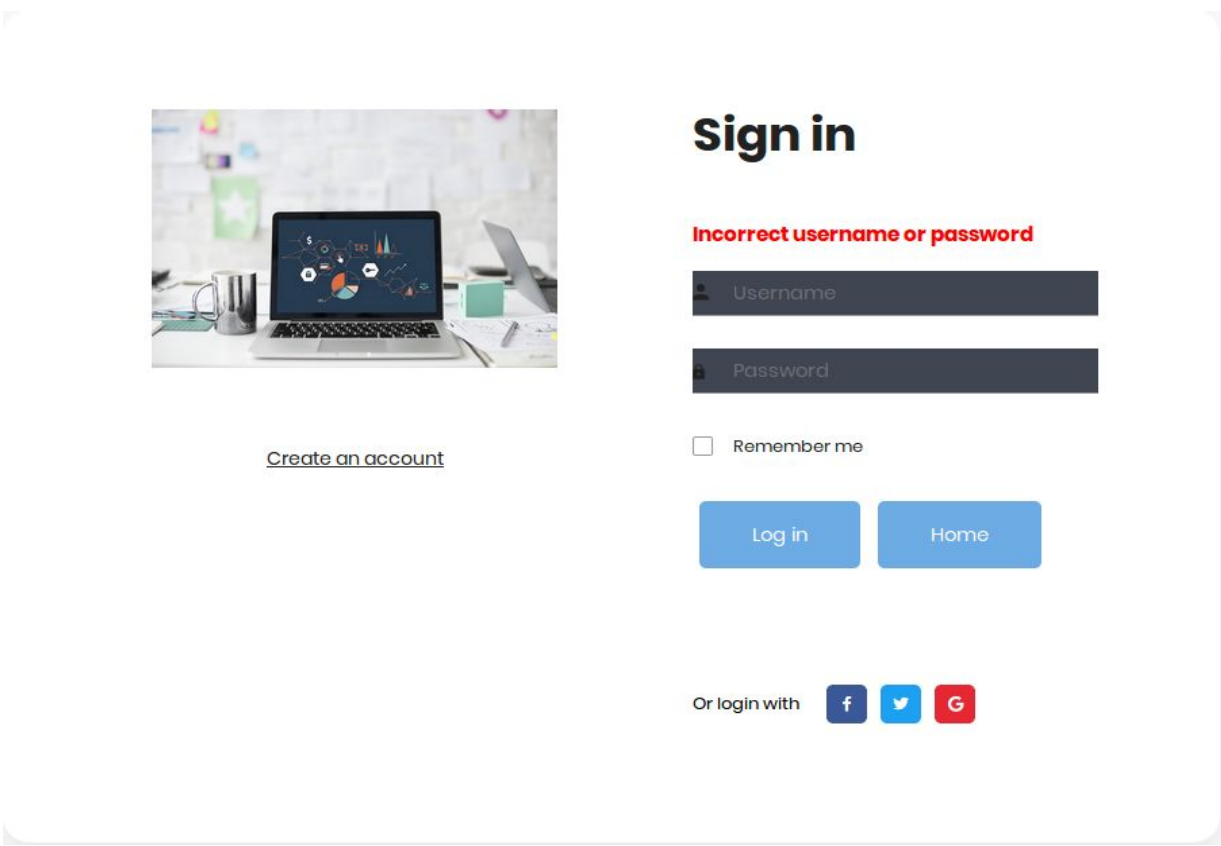
Authentication Bypass (Severity - High/Critical)	1
Description	1
Effects	3
Mitigation	3
Detection	3
Reflected XSS - Using GET (Severity - High/Critical)	5
Description	5
Effects	7
Mitigation	7
Detection	7
SQL Injection (Severity - High/Critical)	9
Description	9
Effects	11
Mitigation	11
Detection	12
Other Possible Vulnerabilities	13
Server Side Template Injection (SSTI)	13
Unrestricted File Upload	13
Cross Site Request Forgery (CSRF)	14

Authentication Bypass (Severity - *High/Critical*)

- **Description**

Authentication bypass vulnerabilities are one of the less common vulnerabilities we see, but they are also one of the easiest to accidentally create.

The application allows a login and registration feature for the user. In backend it uses a MySQL database for storing and retrieving user information. For login when a user hits submit, the values for username and password input field are sent to the server via POST method. On server side the user is authenticated by concatenating the credentials to a SQL query that is to be executed. Doing so allows the attacker to easily bypass the login mechanism by manipulating the user inputs. Let's take a look at how this works.



[Create an account](#)

Sign in




Incorrect username or password

Username

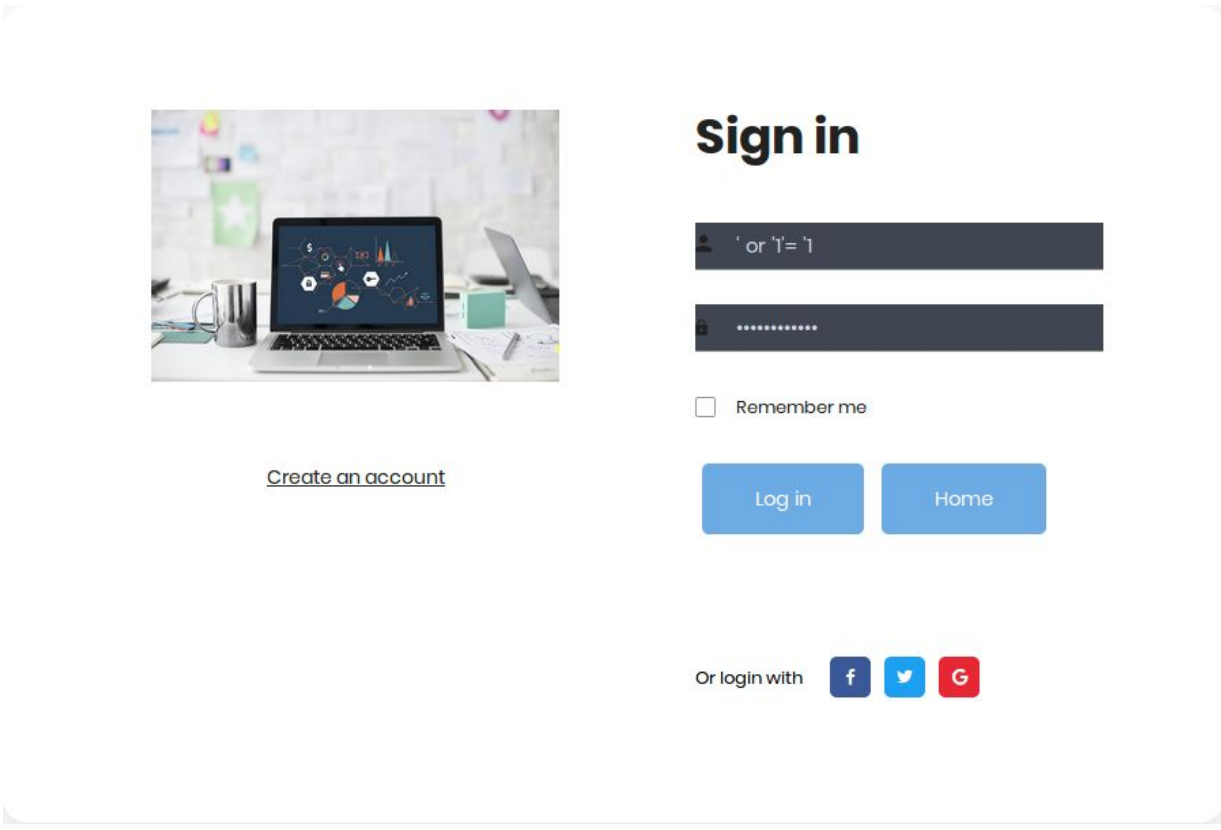
Password

☐ Remember me

Log in Home

Or login with   

- As you can see if the user enters Incorrect credentials he/she is not authenticated. Under normal scenarios the SQL query would be:
SELECT * FROM table_name WHERE username='user' AND password='pass' . This would return nothing because no such user is registered.
- Now let's see a scenario where an attacker can manipulate the input such that the resulting statement is always true. This results in authentication bypass.



- The resulting SQL query in this case would be:
SELECT * FROM table_name WHERE username='' or '1'='1' AND password='' or '1'='1' which will be true in all cases. Hence, allowing an attacker to easily bypass authentication mechanism.

In general, authentication bypass vulnerability is generally caused when it is assumed that users will behave in a certain way and failing to foresee the consequences of users doing the unexpected.

- **Effects**

- Successful exploitation of this vulnerability allows an attacker to bypass the authentication mechanisms and gain unauthorized access to the web application.
- **Worst-Case Scenario:** Unauthorized access to the web application, possibly as an Administrator.

- **Mitigation**

The authentication bypass vulnerability is a special case of SQL injection, specifically located in your authentication routines. The following recommendations will help to mitigate the risk of Authentication Bypass attacks:

- Use comparison technique to authenticate a user (i.e retrieve credentials pair from the database and compare against user provided pair).
- Use prepared Statements instead of concatenating user input to a SQL query.
- Ensure proper input validation is performed wherever user supplied data can be supplied, regardless of the application's relationship to a back-end database.

- **Detection**

- **Static Analysis:**
 1. Search for database queries in the code.
 2. Check if the query is SQL.
 3. If it is a SQL query, check if it uses a prepared statement or generates query by concatenating user input.

4. If it generates query by concatenation, check if it validates the user based on comparison technique.
5. If it does not uses comparison for validation then the application might be vulnerable to Authentication bypass.

- **Runtime Analysis:**

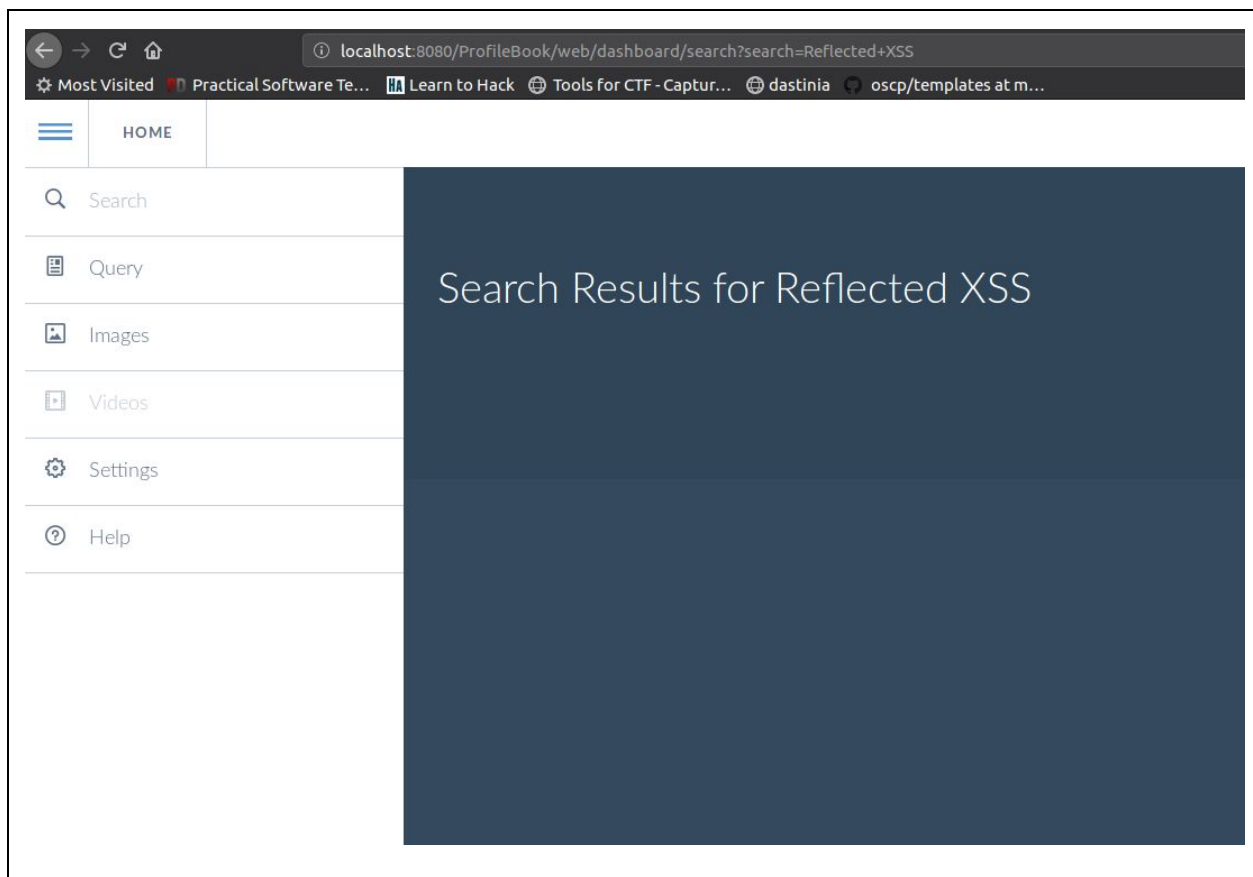
1. Keep track of user input variable to find if it reaches the final query without any validation.
2. If it does, check if it is concatenated to the string.
3. If it is then the application is vulnerable to Authentication bypass

Reflected XSS - Using GET (Severity - *High/Critical*)

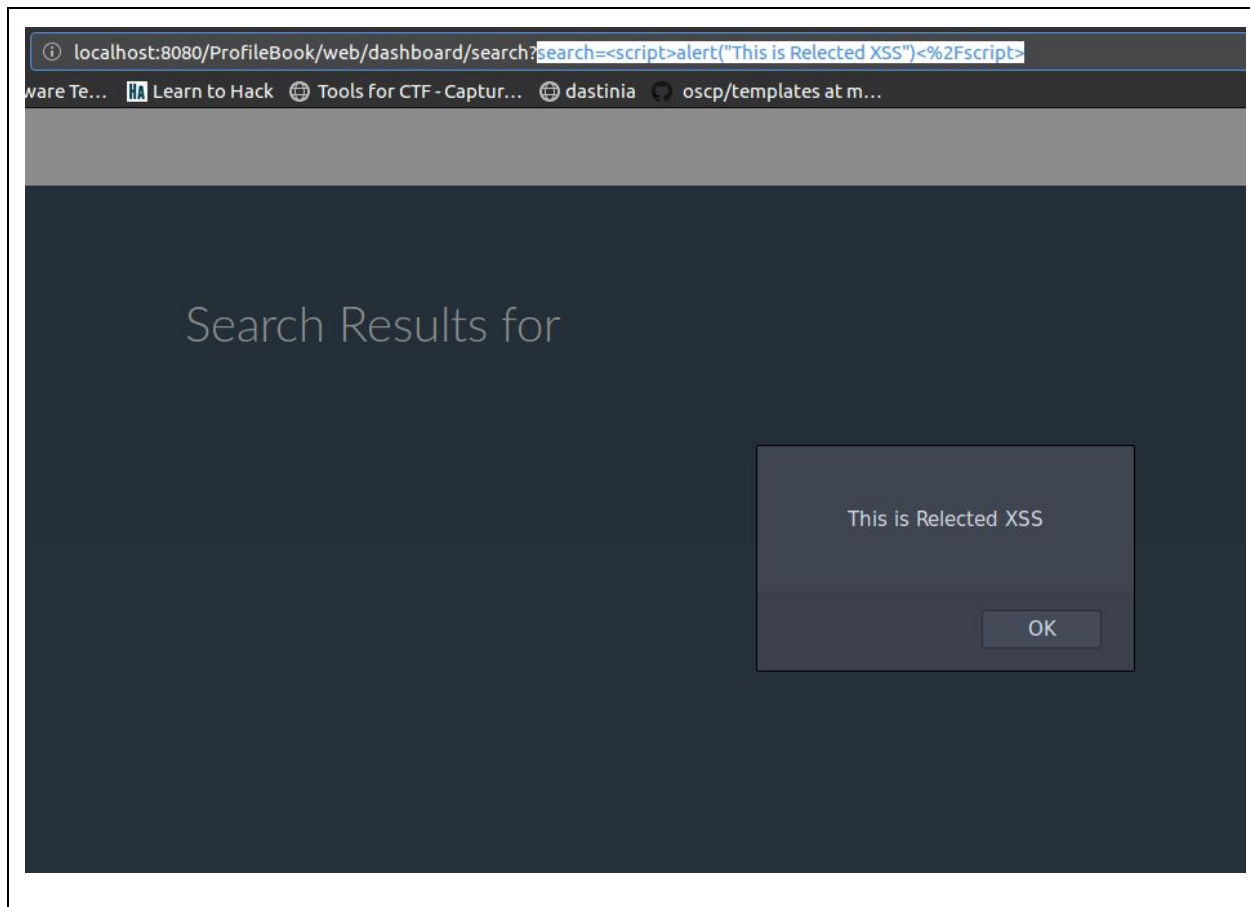
- **Description**

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

- Let's see how the current application is vulnerable to Reflected XSS using GET method.



- You can see a search bar in the above image, that takes a user input and displays search response.
- To exploit this an attacker can craft a string which contains malicious code and make the browser execute it.



- If you look closely at the URI you can see the malicious code injected by an attacker `<script>alert("This is Reflected XSS")</script>`.
- The script tag tricks the browser into executing the code. It uses a GET request which makes it even more easier for an attacker to fool a victim.

To generalize this, Cross Site Scripting (XSS) attacks occurs when data enters a Web application through an untrusted source (like unsanitized input fields), the data is included in dynamic content that is sent to a user without being validated for malicious content.

- **Effects**

- XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise.
- **Worst-Case Scenario:** The most severe XSS attacks involve disclosure of the user's session cookie, allowing an attacker to hijack the user's session and take over the account.
- Other damaging attacks include disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content.

- **Mitigation**

XSS attacks are results of untrusted input data being used directly. The following recommendations will help to mitigate the risk of Cross Site Scripting attacks:

- Ozark has an inbuilt functionality that allows encoding untrusted data, using the methods defined by `javax.mvc.security.Encoders`, to prevent XSS attacks.
- Ensure that the data submitted by clients is properly sanitized before it is manipulated, stored in a database, returned to the client, etc.
- Use validation functions to validate untrusted data, but remember validation is not a replacement for sanitization or escaping.
- Another mitigation technique is to use a Web Application Firewall (WAF).

- **Detection**

- **Static Analysis:**
 1. Check for user input fields in the code.
 2. Check if the input is stored in database, or displayed to the user without sanitization / validation.

3. If the input is used directly then the application might be vulnerable to XSS attack.

- **Runtime Analysis:**

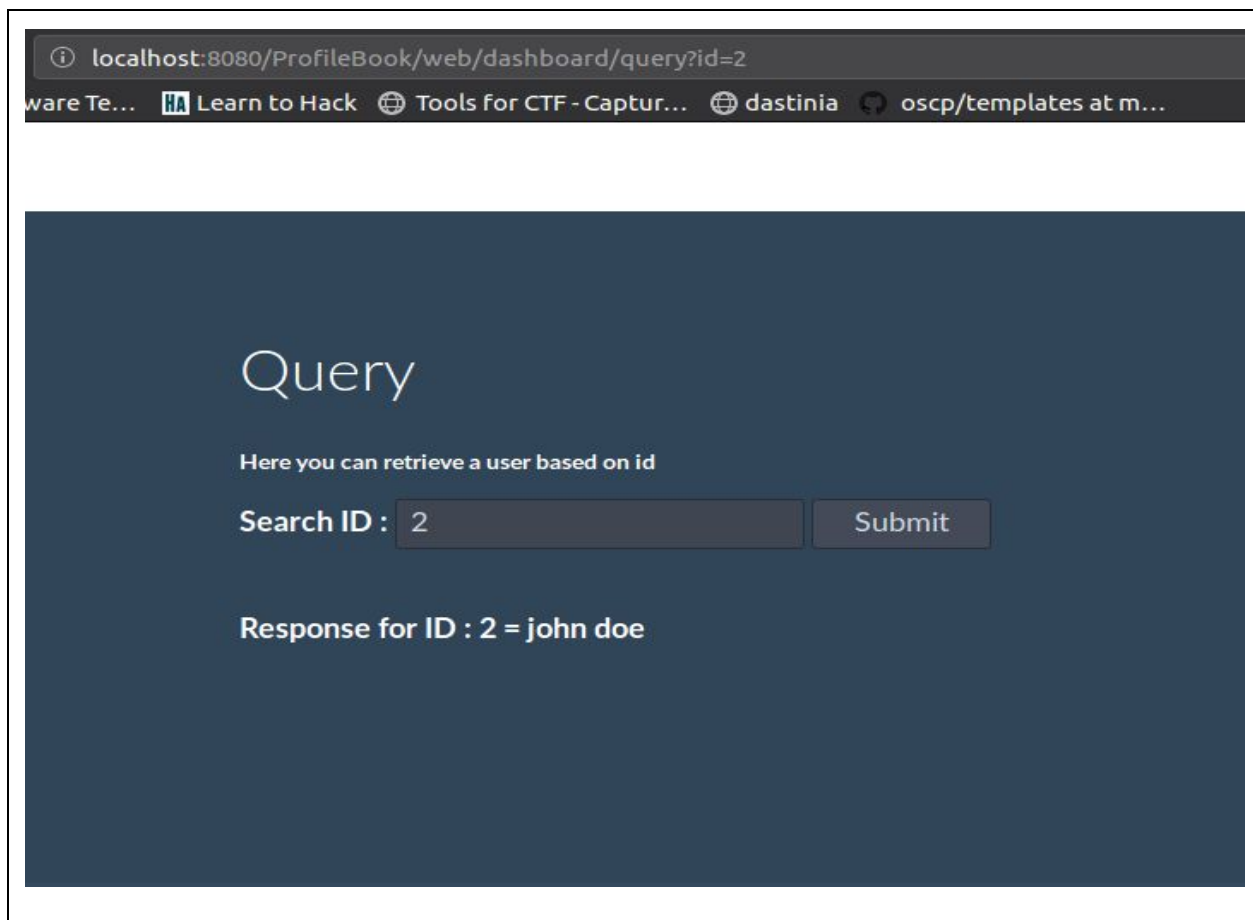
1. Track any user input in the application to check for any additional operation performed on the input (like encoding / validation etc.)
2. If no additional operations is performed and the input is displayed to the on the browser the application might be vulnerable to XSS.

SQL Injection (Severity - *High/Critical*)

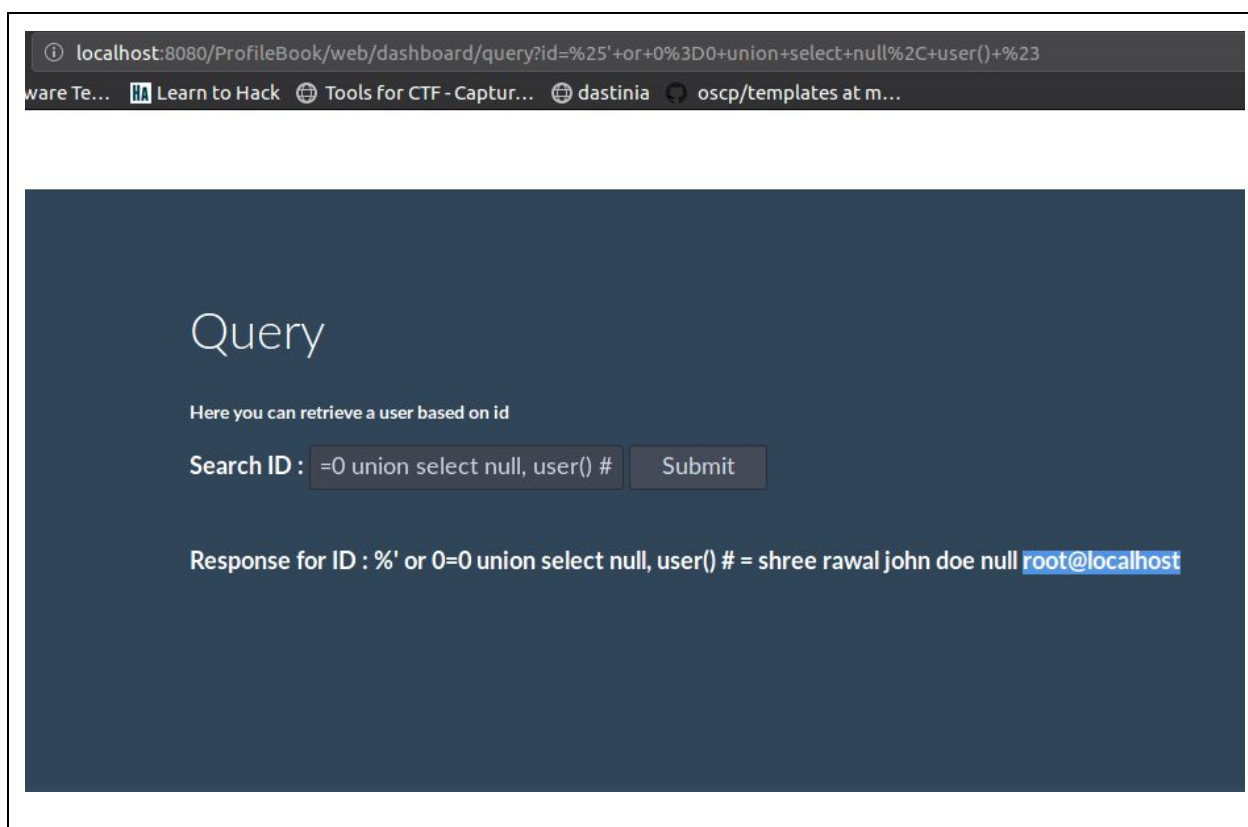
- **Description**

SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands. SQL Injection has become a common issue with database-driven web sites. This attack consists of insertion or "injection" of a SQL query via the input data from the client to the application.

- Let's see how the current application is vulnerable to a SQL Injection attack.
- The application has a query page which allows a user to retrieve first name and last name of a person from the database based on an associated integer user id.

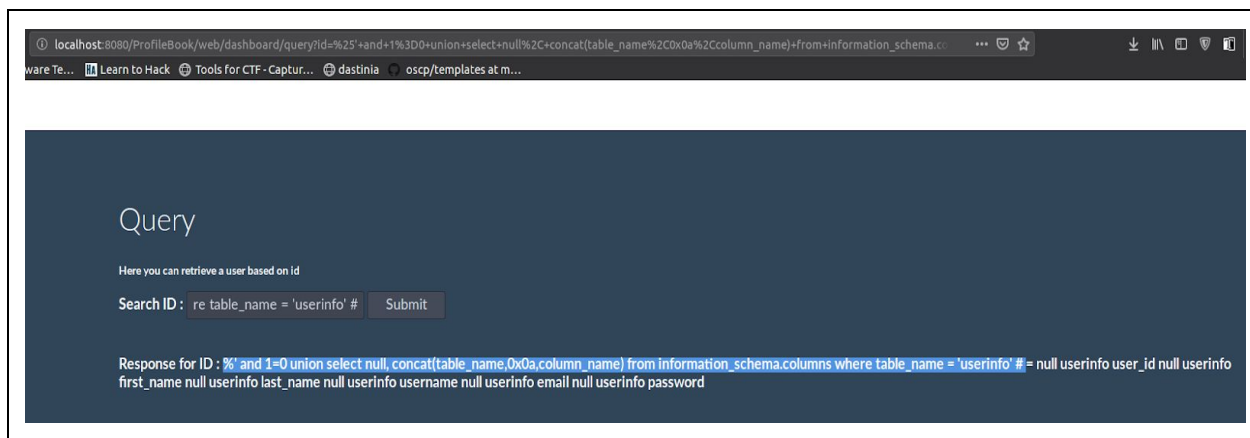


- For a valid user id the SQL query would look something like:
`SELECT first_name,last_name FROM table_name WHERE user_id='2' .`
- As you can see in the image above the first name and last name for user id 2 is displayed as response for the query
- To find if this is vulnerable an attacker will input SQL query in the input field and see if they get executed.



- If you take a look at the query `%' or 0=0 union select null, user() #` in the URI / Input field, we can tell it's not an integer input but still it gets executed and the current database user is displayed on the browser.

- Now since the attacker knows this is vulnerable to SQL injection he/she can execute commands to get database name, table name and also credentials for all the users.



Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. SQL Injection occurs when Data enters a program from untrusted source, and it is used to dynamically construct a SQL query.

● Effects

- A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system.
- This results in loss of Confidentiality, Integrity, Authentication and Authorization.

● Mitigation

There are several effective ways to prevent SQLI attacks from taking place, as well as protecting against them, should they occur.

- Treat every user input as untrusted and sanitize / validate it.
- Use a Web Application Firewall (WAF) to prevent SQLi.
- Use prepared Statements instead of concatenating user input to a SQL query.

- **Detection**

- **Static Analysis:**
 1. Search for database queries in the code.
 2. Check if the query is built using dynamic prepared statements or concatenation.
 3. If the query concatenates user input, the application might be vulnerable to SQL Injection.
- **Runtime Analysis:**
 1. Track user input to check if it is used in a SQL query without validation or sanitization.
 2. If the input data is used as it is the application might be vulnerable to SQL injection.

Other Possible Vulnerabilities

This section explains few other vulnerabilities that may (not necessarily) be possible in the MVC 1.0 (aka ozark) framework.

- **Server Side Template Injection (SSTI)**
 - The web application uses templates to make the web pages look more dynamic. Server Side Template Injection occurs when user input is embedded in a template in an unsafe manner.
 - However, in the initial observation, this vulnerability is easy to mistake for XSS attacks. But SSTI attacks can be used to directly attack web servers' internals and leverage the attack more complex such as running remote code execution and complete server compromise.
 - Template Injection can arise both through developer error, and through the intentional exposure of templates in an attempt to offer rich functionality, as commonly done by wikis, blogs, marketing applications and content management systems.
 - There are a couple of templates in java (such as FreeMarker and Velocity) which are vulnerable to SSTI. If any of these template is used with the framework, the overall application might be vulnerable to Template Injection attack.
- **Unrestricted File Upload**
 - Many websites require file upload functionality for their users. Social networking websites, such as Facebook and Twitter allow their users to upload profile pictures. Job portals allow their users to upload their resumes.
 - File upload functionality is crucial for many web applications. At the same time, it is a big risk to the application as well as to the server if proper security controls are not implemented on file uploads.

- This being said, if any application uses MVC 1.0 and has a file upload feature but does not implements proper security checks, an attacker can upload a backdoor which would allow complete compromise of the web application.
- **Cross Site Request Forgery (CSRF)**
 - Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
 - If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.
 - MVC 1.0 has an inbuilt security feature `@CsrfValid` which helps prevent CSRF attacks, but this needs to be added to code by the developer.
 - If `CsrfOptions.IMPLICIT` is used, all controller methods annotated with `@POST` and that consumes the media type `x-www-form-urlencoded` will be automatically checked for a valid CSRF token.
 - If `CsrfOptions.EXPLICIT` is used, then the `@CsrfValid` annotation must be added explicitly to the methods you want the CSRF token to be validated.
 - Nevertheless, if the CSRF security mechanism is not properly implemented, it doesn't matter if the framework protects against CSRF, the web application will still be vulnerable to CSRF attack.