Pune Institute of Computer Technology
Dhankawadi, Pune

SQL Injection and Cross-Site Scripting attacks

SUBMITTED BY

Rajwinder Singh (41152)
Sahil Singh (41155)
Sanchit Raina (41157)

Under the guidance of
Prof. Rutuja A. Kulkarni

DEPARTMENT OF COMPUTER ENGINEERING
Academic Year 2021-22

Title: SQL Injection attacks and Cross-Site Scripting attacks

Problem Statement:
SQL Injection and Cross-Site Scripting attacks SQL Injection attacks and Cross -Site Scripting attacks are the two most common attacks on web application. Develop a new policy based Proxy Agent, which classifies the request as a scripted request or query based request, and then, detects the respective type of attack, if any in the request. It should detect both SQL injection attack as well as the Cross-Site Scripting attacks.

Objectives:

- Learning and demonstrating about SQL injection attack.
- Learning and demonstrating XSS attacks.

Outcomes:
We were able to:

- design a website where SQL Injection works.
- design a website where XSS attack is possible.
- design a website that detects XSS attacks and prevent it as well.

Software and Hardware Packages:

- Django 4.0.2
- SQLite Database
- Python 3.8
- Pycharm IDE
- OS: Ubuntu 20.04 (64 bits)

Theory concept with algorithm:

SQL Injection:
SQL injection is a technique used to exploit user data through web page inputs by injecting SQL commands as statements. Basically, thesestatements can be used to manipulate the application's web server by malicious users.

SQL injection is a code injection technique that might destroy yourdatabase. SQL injection is one of the most common web hacking techniques. SQL injection is the placement of malicious code in SQL statements, via web page input.

Exploitation of SQL Injection in Web Applications:
Web servers communicate with database servers anytime they need to retrieve or store user data. SQL statements by the attacker are designed so that they can be executed while the web-server is fetching content from the application server. It compromises the security of a web application.

Example of SQL Injection
Suppose we have an application based on student records. Any student can view only his or her own records by entering a unique and private student ID.
Suppose we have a field like below:
student_id:
And the student enters the following in the input field:
12222345 or 1=1.

So this basically translates to:
SELECT * FROM students WHERE
student_id = 12222345 or 1 = 1
Now this 1=1 will return all records for which this holds true. So basically, all the student data is compromised. Now the malicious user can also delete the student records in a similar fashion.

Consider the following SQL query.
SELECT * FROM users WHERE
username = "" and password=""
Now the malicious can use the '=' operator in a clever manner to retrieve private and secure user information. So instead of the above-mentioned query the following query when executed, retrieves protected data, not intended to be shown to users.

SELECT * FROM users WHERE
(username = "" OR 1=1) AND (password="" OR 1=1).
Since 1=1 always holds true, user data is compromised.

Impact of SQL Injection
The hacker can retrieve all the user-data present in the database such as user details, credit card information, social security numbers and can also gain access to protected areas like the administrator portal. It is also possible to delete the user data from the tables. Nowadays, all online shopping applications, bank transactions use back- end database servers. So in-case the hacker is able to exploit SQL injection, the entire server is compromised.

Preventing SQL Injection
User Authentication: Validating input from the user by pre-defining length, type of input, of the input field and authenticating the user. Restricting access privileges of users and defining as to how much amount of data any outsider can access from the database. Basically, user should not be granted permission to access everything in the database. Do not use system administrator accounts.

Cross Site Scripting Attack(XSS):
Cross-site Scripting (XSS) is a client-side code injection attack. The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code. The web page or web application becomes a vehicle to deliver the malicious script to the user's browser. Vulnerable vehicles that are commonly used for Cross-site Scripting attacks are forums, message boards, and web pages that allow comments.

A web page or web application is vulnerable to XSS if it uses unsanitized user input in the output that it generates. This user input must then be parsed by the victim's browser. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. However, they are most common in JavaScript, primarily because JavaScript is fundamental to most browsing experiences.

"Isn't Cross-site Scripting the User's Problem?"
If an attacker can abuse an XSS vulnerability on a web page to execute arbitrary JavaScript in a user's browser, the security of that vulnerable website or vulnerable web application and its users has been compromised. XSS is not the user's problem like any other security vulnerability. If it is affecting your users, it affects you.

Cross-site Scripting may also be used to deface a website instead of targeting the user. The attacker can use injected scripts to change the content of the website or even redirect the browser to another web page, for example, one that contains malicious code.

What Can the Attacker Do with JavaScript? XSS vulnerabilities are perceived as less dangerous than for example SQL Injection vulnerabilities. Consequences of the ability to execute JavaScript on a web page may not seem dire at first. Most web browsers runJavaScript in a very tightly controlled environment. JavaScript has limited access to the user's operating system and the user's files. However, JavaScript can still be dangerous if misused as part of malicious content.

How Cross-site Scripting Works
There are two stages to a typical XSS attack, To run malicious JavaScript code in a victim's browser, an attacker must first find a way to inject malicious code (payload) into a web page that the victim visits.
After that, the victim must visit the web page with the malicious code. If the attack is directed at particular victims, the attacker can use social engineering and/or phishing to send a malicious URL to the victim.
For step one to be possible, the vulnerable website needs to directly include user input in its pages. An attacker can then insert a malicious string that will be used within the web page and treated as source code by the victim's browser. There are also variants of XSS attacks where the attacker lures the user to visit a URL using social engineering and the payload is part of the link that the user clicks.

The following is a snippet of server-side pseudocode that is used to display the most recent comment on a web page:
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"

The above script simply takes the latest comment from a database and includes it in an HTML page. It assumes that the comment printed out consists of only text and contains no HTML tags or other code. It is vulnerable to XSS, because an attacker could submit a comment that contains a malicious payload.
for example:
<script>doSomethingEvil();</script>

The web server provides the following HTML code to users that visit this web page:
<html>
<h1>Most recent comment</h1>
<script>doSomethingEvil();</script>
</html>

When the page loads in the victim's browser, the attacker's malicious script executes. Most often, the victim does not realize it and is unable to prevent such an attack.

How to Prevent Cross-site Scripting (XSS)
Preventing Cross-site Scripting (XSS) is not easy. Specific prevention techniques depend on the subtype of XSS vulnerability, on user input usage context, and on the programming framework. However, there are certain general strategic principles that you should follow to keep your web application safe.

1. Train and maintain awareness
   To keep your web application safe, everyone involved in building the web application must be aware of the risks associated with XSS vulnerabilities. You should provide suitable security training to all your developers, QA staff, DevOps, and SysAdmins. You can start by referring them to this page.

2. Don't trust any user input
   Treat all user input as untrusted. Any user input that is used as part of HTML output introduces a risk of an XSS. Treat input from authenticated and/or internal users the same way that you treat public input.

3. Use escaping/encoding
   Use an appropriate escaping/encoding technique depending on where user input is to be used: HTML escape, JavaScript escape, CSS escape, URL escape, etc. Use existing libraries for escaping, don't write your own unless absolutely necessary.

4. Sanitize HTML
   If the user input needs to contain HTML, you can't escape/encode it because it would break valid tags. In such cases, use a trusted and verified library to parse and clean HTML. Choose the library depending on your development language, for example, Html-Sanitizer for .NET or SanitizeHelper for Ruby on Rails.

5. Set the HttpOnly flag
   To mitigate the consequences of a possible XSS vulnerability, set the HttpOnly flag for cookies. If you do, such cookies will not be accessible via client-side JavaScript.

6. Use a Content Security Policy
   To mitigate the consequences of a possible XSS vulnerability, also use a Content Security Policy (CSP). CSP is an HTTP response header that lets you declare the dynamic resources that are allowed to load depending on the request source.

7. Scan regularly (with Acunetix)
   XSS vulnerabilities may be introduced by your developers or through external libraries/modules/softw
   You should regularly scan your web applications using a web vulnerability scanner such as Acunetix. If you use Jenkins, you should install the Acunetix plugin to automatically scan every build.

Test Cases:

1. SQL injection:

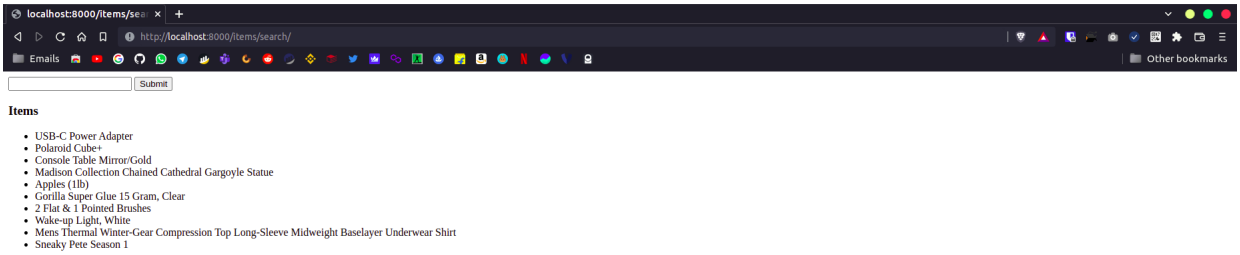

Figure 1: Item Search Page

SQL Injection query:
'UNION SELECT password FROM auth_user WHERE first_name LIKE'

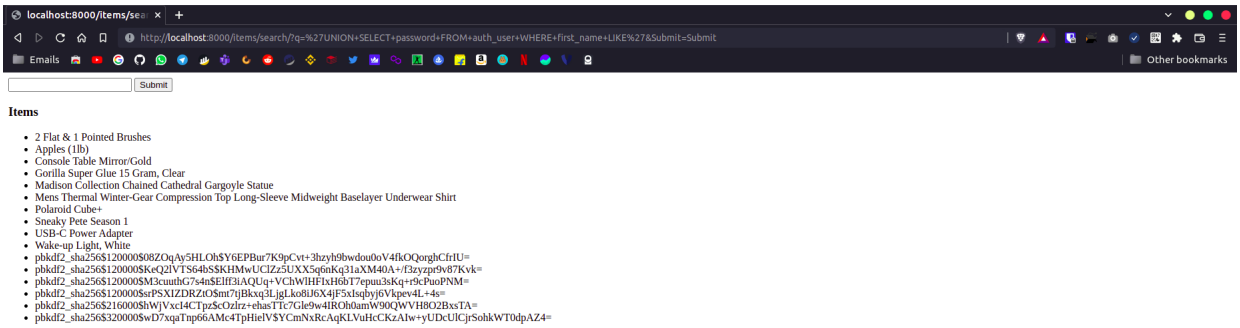

Figure 2: Unauthorised access of data

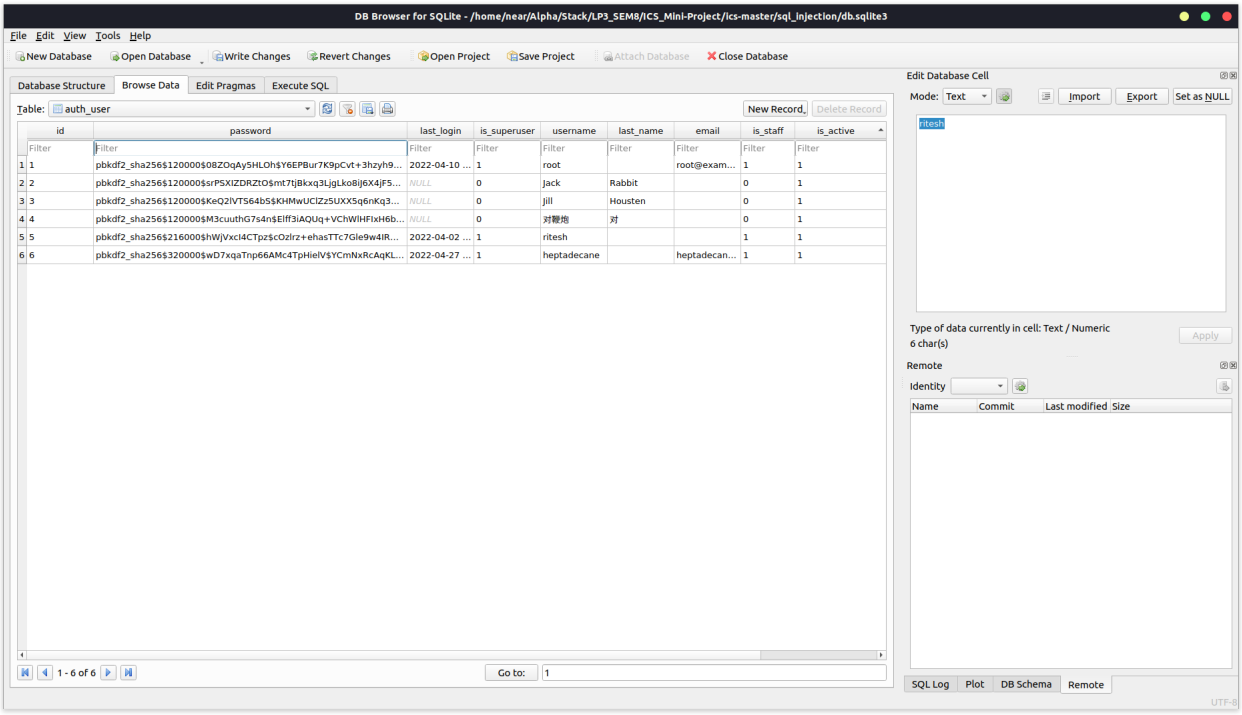

Figure 3: SQLite: auth_user table

2. XSS Attack:

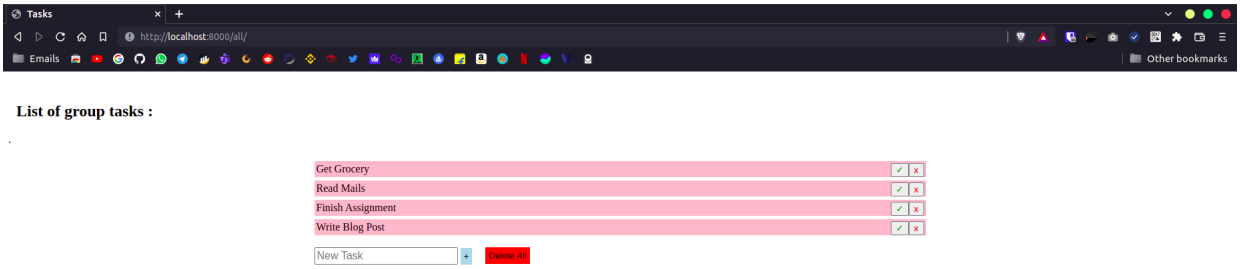

Figure 4: ToDo List

XSS Script:
<pre><code>
Malicious Code
<form id="myform" action="http://127.0.0.1:8000/deleteall" method="post"></form>
<script>setTimeout(() => document.getElementById("myform").submit();, 5000);</script>
</code></pre>

Figure 5: Malicious snippet to POST 'deleteall'

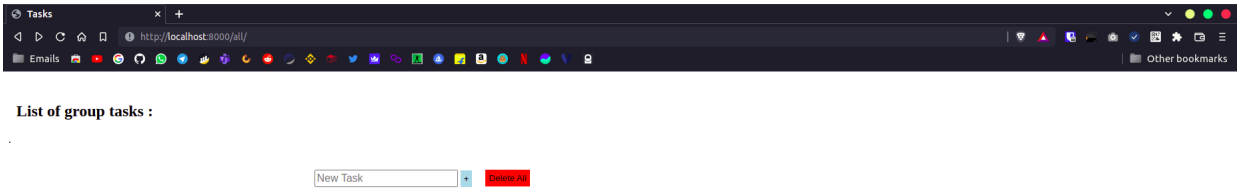

Figure 6: Items deleted due to Malicious script

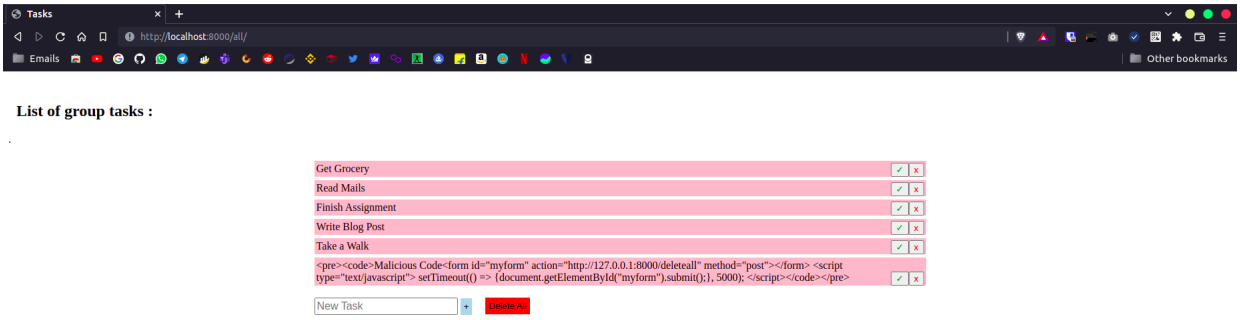

Figure 7: Preventing XSS sxripts with escaping/encoding

Conclusion:

With this study, we have successfully demonstrated SQL Injection and Cross-Site Scripting attacks. Also, designed a website demonstrating prevention and detection of the discussed attacks.