**Assignment No: 06**

**Title:** Mini Project-I.

**Objectives:**

To know the basics of cryptography.

To acquire knowledge of standard algorithms and protocols employed to provide confidentiality, integrity and authenticity.

**Problem Statement:** 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.

**Outcome:** Build appropriate security solutions against cyber-attacks.

**Software & Hardware Requirments:**

1. Mozilla Firefox

2. Php

3. Notepad/ Notepad++ Editor

**Theory:**

**1. SQL Injection Attacks:**

**SQL injection**is a [code injection](https://en.wikipedia.org/wiki/Code_injection) technique, used to [attack](https://en.wikipedia.org/wiki/Attack_(computing)) data-driven applications, in which malicious [SQL](https://en.wikipedia.org/wiki/SQL) statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).[[1]](https://en.wikipedia.org/wiki/SQL_injection#cite_note-1) SQL injection must exploit a [security vulnerability](https://en.wikipedia.org/wiki/Security_vulnerability) in an application's software, for example, when user input is either incorrectly filtered for [string literal](https://en.wikipedia.org/wiki/String_literal) [escape characters](https://en.wikipedia.org/wiki/Escape_sequence) embedded in SQL statements or user input is not [strongly typed](https://en.wikipedia.org/wiki/Strongly-typed_programming_language) and unexpectedly executed. SQL injection is mostly known as an attack [vector](https://en.wikipedia.org/wiki/Vector_(malware)) for websites but can be used to attack any type of SQL database.

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.

1.1 **Incorrectly filtered escape characters**

This form of injection occurs when user input is not filtered for [escape characters](https://en.wikipedia.org/wiki/Escape_character) and is then passed into an SQL statement. This results in the potential manipulation of the statements performed on the database by the end-user of the application.

The following line of code illustrates this vulnerability:

statement = "**SELECT** \* **FROM** users **WHERE** name = '" + userName + "';"

This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "userName" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "userName" variable as:

' OR '1'='1

or using comments to even block the rest of the query (there are three types of SQL comments[[13]](https://en.wikipedia.org/wiki/SQL_injection#cite_note-13)). All three lines have a space at the end:

' OR '1'='1' --

' OR '1'='1' {

' OR '1'='1' /\*

renders one of the following SQL statements by the parent language:

**SELECT** \* **FROM** users **WHERE** name = '' **OR** '1'='1';

**SELECT** \* **FROM** users **WHERE** name = '' **OR** '1'='1' *-- ';*

If this code were to be used in an authentication procedure then this example could be used to force the selection of every data field (\*) from *all* users rather than from one specific user name as the coder intended, because the evaluation of '1'='1' is always true.

The following value of "userName" in the statement below would cause the deletion of the "users" table as well as the selection of all data from the "userinfo" table (in essence revealing the information of every user), using an [API](https://en.wikipedia.org/wiki/API) that allows multiple statements:

a';**DROP** **TABLE** users; **SELECT** \* **FROM** userinfo **WHERE** 't' = 't

This input renders the final SQL statement as follows and specified:

**SELECT** \* **FROM** users **WHERE** name = 'a';**DROP** **TABLE** users; **SELECT** \* **FROM** userinfo **WHERE** 't' = 't';

While most SQL server implementations allow multiple statements to be executed with one call in this way, some SQL APIs such as [PHP](https://en.wikipedia.org/wiki/PHP)'s mysql\_query() function do not allow this for security reasons. This prevents attackers from injecting entirely separate queries, but doesn't stop them from modifying queries.

**1.2 Blind SQL injection**

Blind SQL injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack has traditionally been considered time-intensive because a new statement needed to be crafted for each bit recovered, and depending on its structure, the attack may consist of many unsuccessful requests. Recent advancements have allowed each request to recover multiple bits, with no unsuccessful requests, allowing for more consistent and efficient extraction.  There are several tools that can automate these attacks once the location of the vulnerability and the target information has been established.

**1.2.1 Conditional responses**

One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen. As an example, a book review website uses a [query string](https://en.wikipedia.org/wiki/Query_string) to determine which book review to display. So the [URL](https://en.wikipedia.org/wiki/URL)http://books.example.com/showReview.php?ID=5 would cause the server to run the query

**SELECT** \* **FROM** bookreviews **WHERE** ID = 'Value(ID)';

from which it would populate the review page with data from the review with [ID](https://en.wikipedia.org/wiki/Identifier) 5, stored in the [table](https://en.wikipedia.org/wiki/Table_(database)) bookreviews. The query happens completely on the server; the user does not know the names of the database, table, or fields, nor does the user know the query string. The user only sees that the above URL returns a book review.

A [hacker](https://en.wikipedia.org/wiki/Hacker_(computer_security)) can load the URLs http://books.example.com/showReview.php?ID=5 **OR** 1=1 and http://books.example.com/showReview.php?ID=5 **AND**1=2, which may result in queries

**SELECT** \* **FROM** bookreviews **WHERE** ID = '5' **OR** '1'='1';

**SELECT** \* **FROM** bookreviews **WHERE** ID = '5' **AND** '1'='2';

respectively. If the original review loads with the "1=1" URL and a blank or error page is returned from the "1=2" URL, and the returned page has not been created to alert the user the input is invalid, or in other words, has been caught by an input test script, the site is likely vulnerable to a SQL injection attack as the query will likely have passed through successfully in both cases. The hacker may proceed with this query string designed to reveal the version number of [MySQL](https://en.wikipedia.org/wiki/MySQL) running on the server: http://books.example.com/showReview.php?ID=5 **AND** substring(@@version, 1, INSTR(@@version, '.') - 1)=4, which would show the book review on a server running MySQL 4 and a blank or error page otherwise. The hacker can continue to use code within query strings to glean more information from the server until another avenue of attack is discovered or his goals are achieved.

**2. Cross -Site Scripting attacks**

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

**2.1** [**Stored and Reflected XSS Attacks**](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)#Stored_and_Reflected_XSS_Attacks)

XSS attacks can generally be categorized into two categories: stored and reflected. There is a third, much less well-known type of XSS attack called [DOM Based XSS](https://www.owasp.org/index.php/DOM_Based_XSS) that is discussed separately [here](https://www.owasp.org/index.php/DOM_Based_XSS).

**2.2 Stored XSS Attacks**

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

**2.3 Reflected XSS Attacks**

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other website. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user’s browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

**2.4 Other Types of XSS Vulnerabilities**

In addition to Stored and Reflected XSS, another type of XSS, [DOM Based XSS](https://www.owasp.org/index.php/DOM_Based_XSS) was identified by [Amit Klein in 2005](http://www.webappsec.org/projects/articles/071105.shtml). OWASP recommends the XSS categorization as described in the OWASP Article: [Types of Cross-Site Scripting](https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting), which covers all these XSS terms, organizing them into a matrix of Stored vs. Reflected XSS and Server vs. Client XSS, where DOM Based XSS is a subset of Client XSS.

**2.5 XSS Attack Consequences**

The consequence of an XSS attack is the same regardless of whether it is stored or reflected ([or DOM Based](https://www.owasp.org/index.php/DOM_Based_XSS)). The difference is in how the payload arrives at the server. Do not be fooled into thinking that a “read-only” or “brochureware” site is not vulnerable to serious reflected XSS attacks. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. 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 the disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content. An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company’s stock price or lessen consumer confidence. An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose. For more information on these types of attacks see [Content\_Spoofing](https://www.owasp.org/index.php/Content_Spoofing).

**2.6 How to Determine If You Are Vulnerable**

XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output. Note that a variety of different HTML tags can be used to transmit a malicious JavaScript. Nessus, Nikto, and some other available tools can help scan a website for these flaws, but can only scratch the surface. If one part of a website is vulnerable, there is a high likelihood that there are other problems as well.

**2.7 How to Protect Yourself**

The primary defenses against XSS are described in the [OWASP XSS Prevention Cheat Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet). Also, it's crucial that you turn off HTTP TRACE support on all web servers. An attacker can steal cookie data via Javascript even when document.cookie is disabled or not supported by the client. This attack is mounted when a user posts a malicious script to a forum so when another user clicks the link, an asynchronous HTTP Trace call is triggered which collects the user's cookie information from the server, and then sends it over to another malicious server that collects the cookie information so the attacker can mount a session hijack attack. This is easily mitigated by removing support for HTTP TRACE on all web servers.

The [OWASP ESAPI project](https://www.owasp.org/index.php/ESAPI) has produced a set of reusable security components in several languages, including validation and escaping routines to prevent parameter tampering and the injection of XSS attacks. In addition, the [OWASP WebGoat Project](https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project)training application has lessons on Cross-Site Scripting and data encoding.

**2.8 Alternate XSS Syntax**

**2.8.1 XSS using Script in Attributes**

XSS attacks may be conducted without using <script></script> tags. Other tags will do exactly the same thing, for example:

<body onload=alert('test1')>

or other attributes like: onmouseover, onerror.

onmouseover

<b onmouseover=alert('Wufff!')>click me!</b>

onerror

<img src="http://url.to.file.which/not.exist" onerror=alert(document.cookie);>

**2.8.2 XSS using Script Via Encoded URI Schemes**

If we need to hide against web application filters we may try to encode string characters, e.g.: a=&#X41 (UTF-8) and use it in IMG tag:

<IMG SRC=j&#X41vascript:alert('test2')>

There are many different UTF-8 encoding notations what give us even more possibilities.

**2.8.3 XSS using code encoding**

We may encode our script in base64 and place it in META tag. This way we get rid of alert() totally. More information about this method can be found in [RFC 2397](https://tools.ietf.org/html/rfc2397)

<META HTTP-EQUIV="refresh"

CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">

These and others examples can be found at the OWASP [XSS Filter Evasion Cheat Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet) which is a true encyclopedia of the alternate XSS syntax attack.

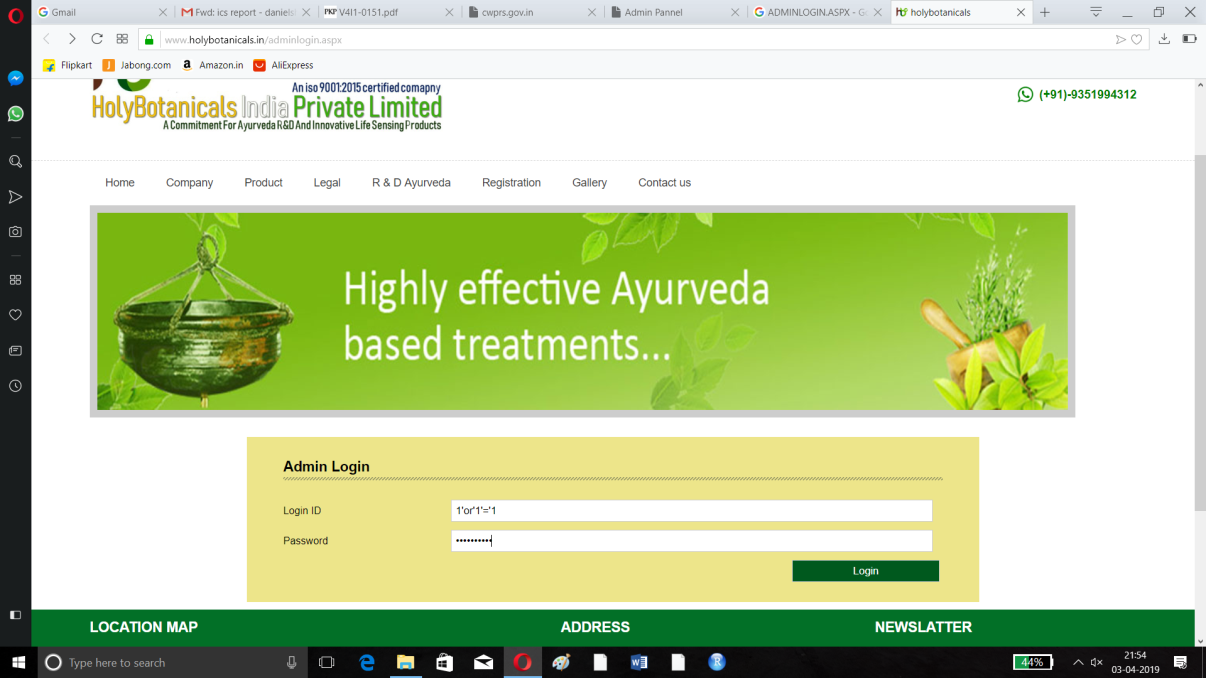
**Conclusion:**

Hence, we have learned SQL Injection attacks and Cross -Site Scripting attacks

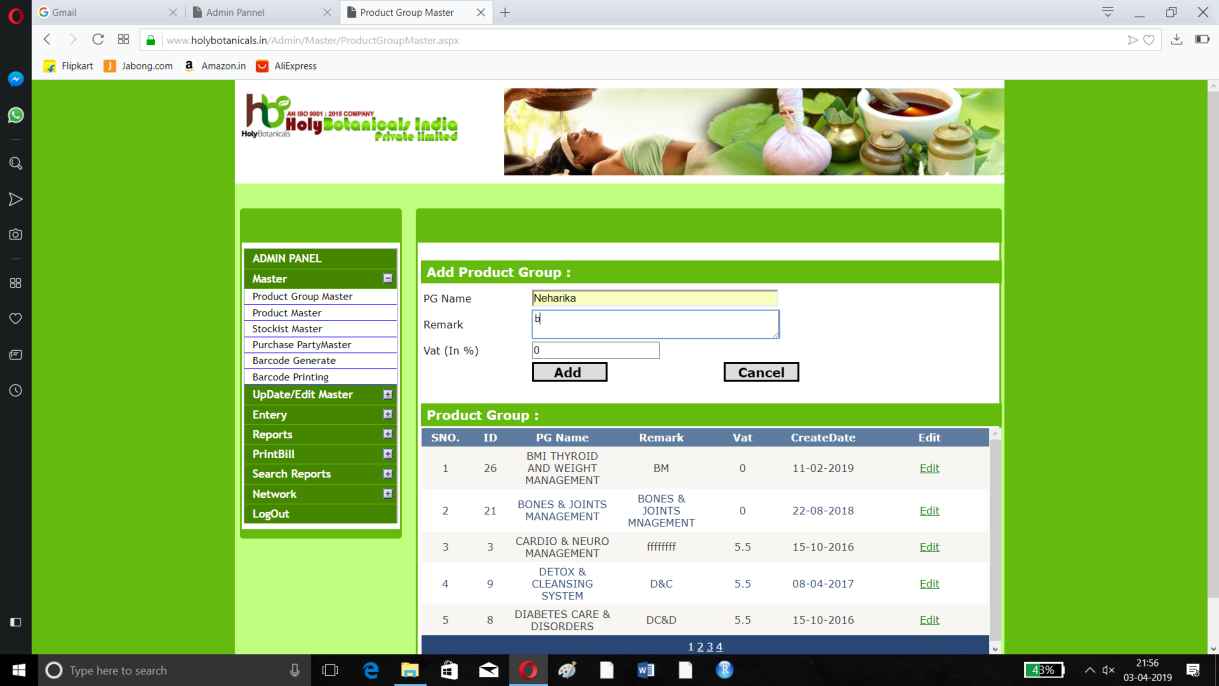
.

**Output**

**1. Sql Injection Attack**



**Fig 1. Sql Injection Attack**



**Fig 2. Sql Injection Attack**