

Dynamic Security Analysis with OWASP ZAP

Dynamic Security Analysis.

Dynamic testing is done when the code is in operation mode. Dynamic testing is performed in runtime environment. When the code being executed is input with a value, the result or the output of the code is checked and compared with the [2]. expected output. With this we can observe the functional behavior of the software, monitor the system memory, CPU response time, performance of the system. Dynamic testing is also known as validation testing, evaluating the finished product. Dynamic testing is of two types: Functional Testing and Nonfunctional testing





Dynamic Testing Techniques.

The Dynamic testing techniques can be classified into two categories. They are:

1. Functional Testing.
2. Non-Functional Testing.

Levels of Dynamic Testing.

There are various levels of Dynamic Testing Techniques. They are:

-  Unit Testing
-  Integration Testing
-  System Testing
-  Acceptance Testing

To do this analysis you can use any dynamic security analysis tool which are existing, here it is used OWASP ZAP (OWASP Zed Attack Proxy) tool.

What is OWASP ZAP?

OWASP ZAP is an open-source web application security scanner. It is intended to be used by both those new to application security as well as professional penetration testers. It is one of the most active OWASP projects and has been given Flagship status. When used as a proxy server it allows the user to manipulate all of the traffic that passes through it, including traffic using https. It can also run in a 'daemon' mode which is then controlled via a REST Application programming interface. This cross-platform tool is written in Java and is available in all of the popular operating systems including Microsoft Windows, Linux and Mac OS [1].

You can download Zap using this link address:

<https://github.com/zaproxy/zaproxy/wiki/Downloads>

Demonstration of OWASP ZAP.

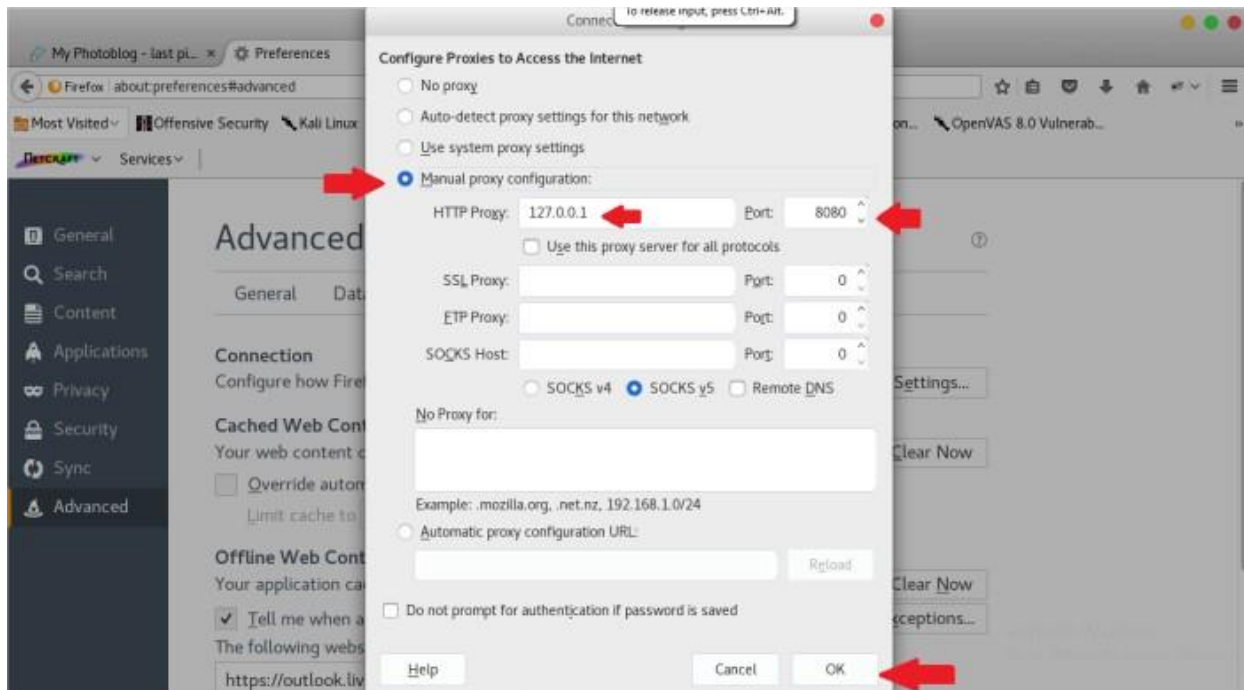
Vulnerable image that has used in this practical can be download the link below:

https://pentesterlab.com/exercises/from_sqli_to_shell

After download and install the above software and applications, Configure the proxy setting in browser and the ZAP. Open zap by typing command in terminal "zaproxy", it's easy than navigating through the menu.

```
root@kali:~# zaproxy
Found Java version 1.8.0 102
Available memory: 2319 MB
Setting jvm heap size: -Xmx512m
0 [main] INFO org.zaproxy.zap.GuiBootstrap - OWASP ZAP 2.5.0 started.
11927 [AWT-EventQueue-1] INFO org.parosproxy.paros.network.SSLConnector - Reading supported SSL/TLS protocols...
11927 [AWT-EventQueue-1] INFO org.parosproxy.paros.network.SSLConnector - Using a SSLEngine...
13134 [AWT-EventQueue-1] INFO org.parosproxy.paros.network.SSLConnector - Done reading supported SSL/TLS protocols: [SSLv2Hello, SSLv3, TLSv1, TLSv1.1, TLSv1.2]
13138 [AWT-EventQueue-1] INFO org.parosproxy.paros.extension.option.OptionsParam
```

Then configure browser Manual proxy setting as follows.

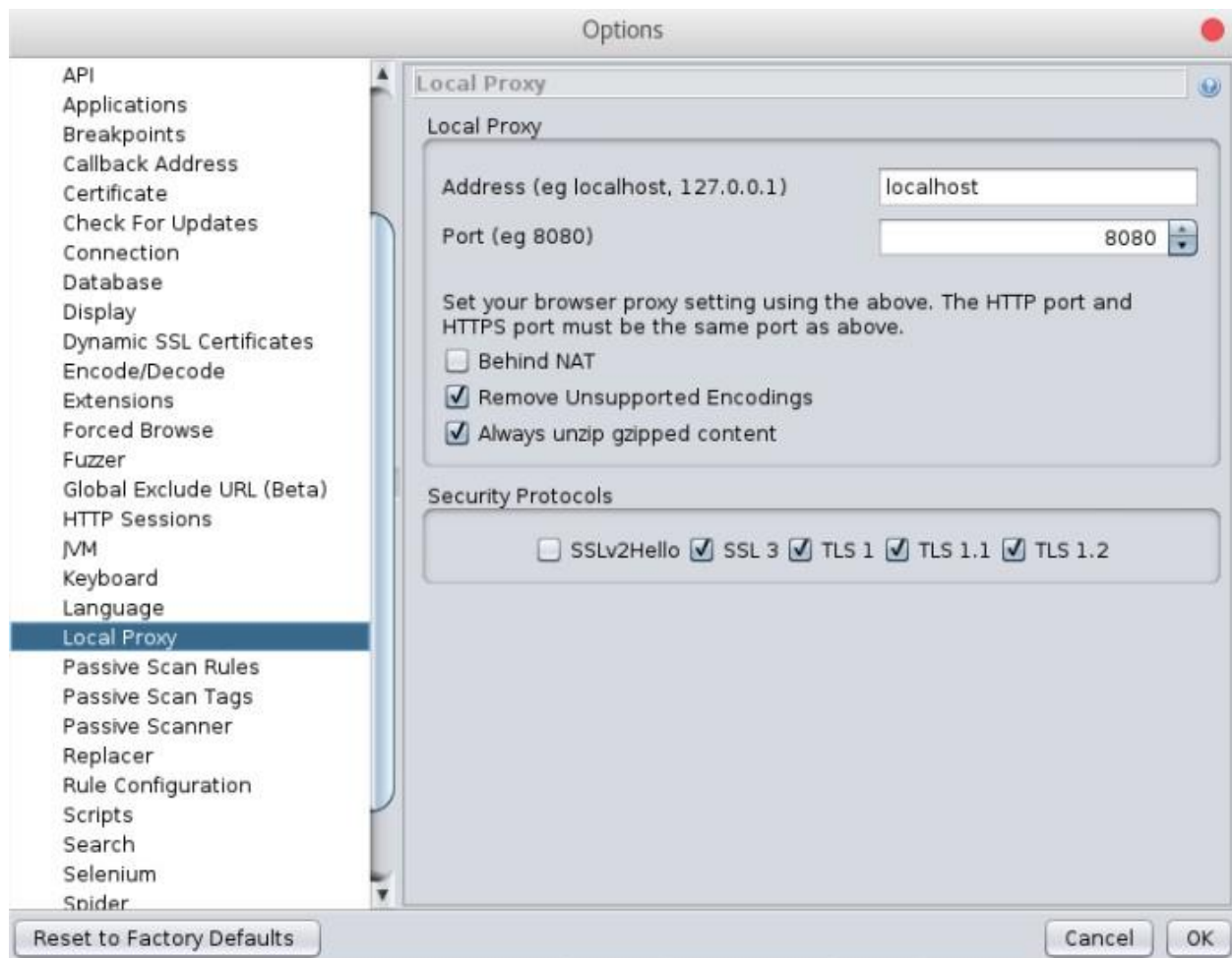


- ❖ Port:-8080
- ❖ HTTP Proxy: - 127.0.0.1

Then we need to setup local proxy setting in ZAP as same as browser proxy settings. To do that go to ZAP,

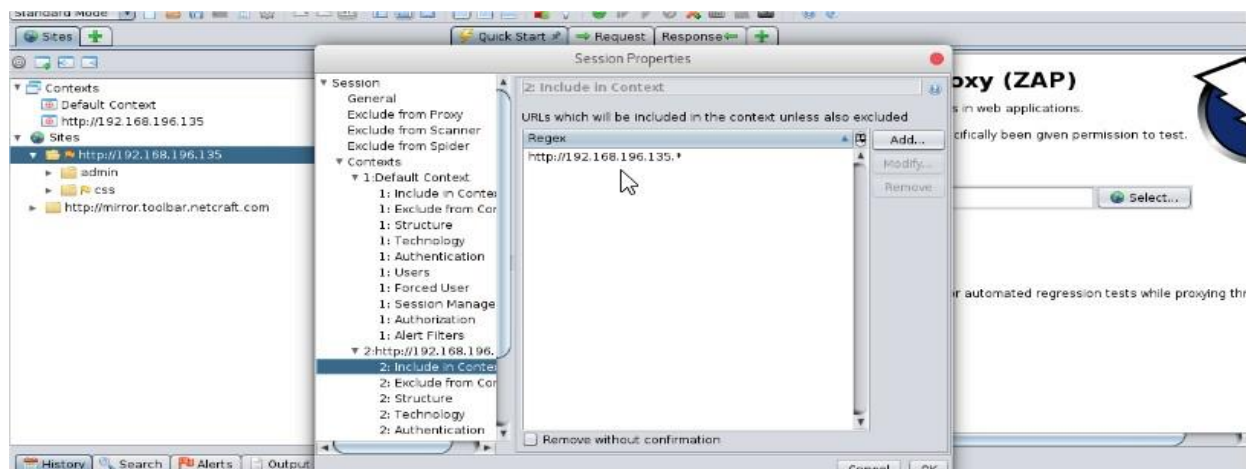


tools->Options->Local Proxy



Now following screenshots explains how the scanning can be done through the ZAP.

Right click on site link -> include in context -> new context

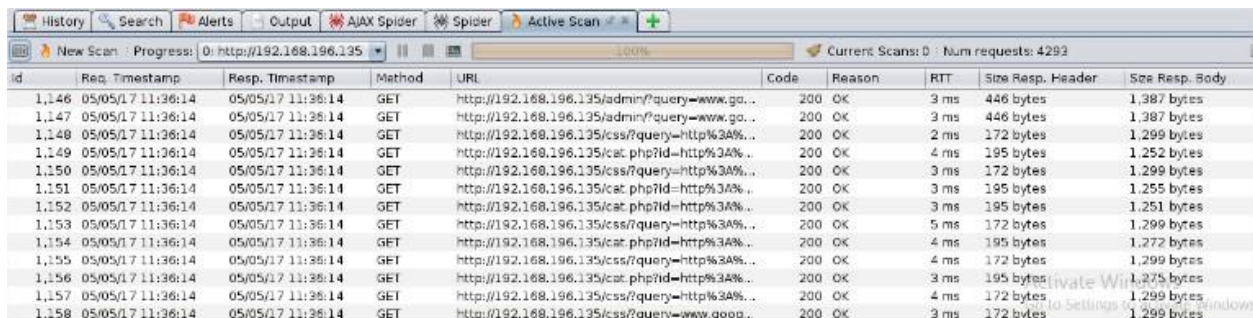


Then the following steps should be completed for retrieving a ZAP report.

Now your site will appear in the context category. Now Select scan mode into Protected Mode. Then go to **tools->Ajax Spider**, in that opened window, click select and select our site in contexts category next select the browser as HtmlUnit (if we are use firefox we need to import some plug-in to do the analysis, HtmlUnit is in-build part in zap). Then click the start scan button.

After complete the Ajax spider, Again Go to **tools->Spider** in that opened window click select button in starting point and select our site and click start scan button.

To perform final scan, go to **tools->Active Scan**, in that opened menu select the site and start the scan.



id	Req. Timestamp	Resp. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Header	Size Resp. Body
1,146	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/admin/?query=www.go...	200	OK	3 ms	446 bytes	1,387 bytes
1,147	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/admin/?query=www.go...	200	OK	3 ms	446 bytes	1,387 bytes
1,148	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/css/?query=http%3A%...	200	OK	2 ms	172 bytes	1,299 bytes
1,149	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/cat.php?id=http%3A%...	200	OK	4 ms	195 bytes	1,252 bytes
1,150	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/css/?query=http%3A%...	200	OK	3 ms	172 bytes	1,299 bytes
1,151	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/cat.php?id=http%3A%...	200	OK	3 ms	195 bytes	1,255 bytes
1,152	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/cat.php?id=http%3A%...	200	OK	3 ms	195 bytes	1,251 bytes
1,153	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/css/?query=http%3A%...	200	OK	5 ms	172 bytes	1,299 bytes
1,154	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/cat.php?id=http%3A%...	200	OK	4 ms	195 bytes	1,272 bytes
1,155	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/css/?query=http%3A%...	200	OK	4 ms	172 bytes	1,299 bytes
1,156	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/cat.php?id=http%3A%...	200	OK	3 ms	195 bytes	1,275 bytes
1,157	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/css/?query=http%3A%...	200	OK	4 ms	172 bytes	1,299 bytes
1,158	05/05/17 11:36:14	05/05/17 11:36:14	GET	http://192.168.196.135/css/?query=www.goog...	200	OK	3 ms	172 bytes	1,299 bytes



Alerts (9)

- Cross Site Scripting (Reflected)**
 - GET: http://192.168.196.135/cat.php?id=%3C%2Fdiv%3E%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E%3Cdiv%3E
- SQL Injection
 - GET: http://192.168.196.135/cat.php?id=4-2
- Application Error Disclosure (27)
- Directory Browsing (3)
- X-Frame-Options Header Not Set (37)
- Cookie No HttpOnly Flag (4)
- Password Autocomplete in Browser
- Web Browser XSS Protection Not Enabled (40)
- X-Content-Type-Options Header Missing (44)

Cross Site Scripting (Reflected)

URL: http://192.168.196.135/cat.php?id=%3C%2Fdiv%3E%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E%3Cdiv%3E

Risk: High

Confidence: Medium

Parameter: id

Attack: </div><script>alert(1)</script><div>

Evidence: </div><script>alert(1)</script><div>

CWE ID: 79

WASC ID: 8

Source: Active

Description:

Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within Winamp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also

After finished all above scans we can get a report. To Get the report go to **Report->Generate HTML Report** and give a name for report, set the path and click Save.

ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	2
Medium	5
Low	11
Informational	0

Alert Detail

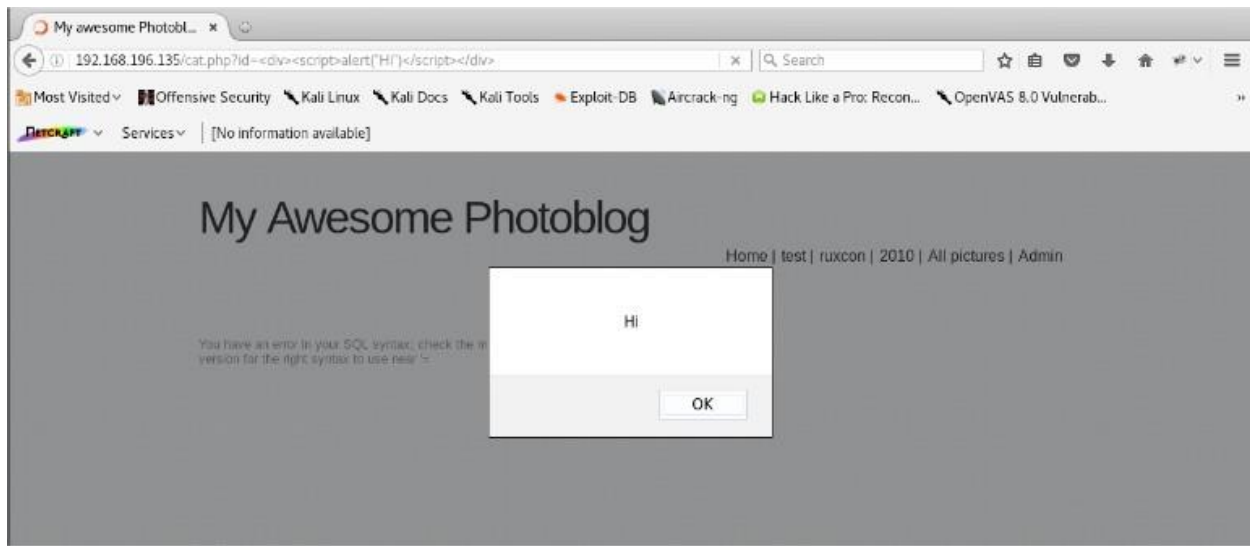
High (Medium)	Cross Site Scripting (Reflected)
Description	<p>Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.</p> <p>When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.</p> <p>There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.</p> <p>Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.</p> <p>Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.</p>
URL	http://localhost:8080/webgot-container-7.1/attack?Screen=165745535&menu=2000
Parameter	sql
Attack	><script>alert(1);</script>
Evidence	><script>alert(1);</script>
Instances	1
Solution	<p>Phase: Architecture and Design</p> <p>Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.</p> <p>Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.</p> <p>Phases: Implementation; Architecture and Design</p> <p>Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.</p> <p>For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.</p> <p>Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.</p> <p>Phase: Architecture and Design</p> <p>For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side. In order to avoid CVE-402, Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.</p> <p>If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide</p>
Reference	<p>Phase: Implementation</p> <p>For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CVE-116 for more mitigations related to encoding/escaping.</p> <p>To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHttpRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.</p> <p>Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.</p> <p>When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "bad" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue".</p> <p>Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.</p> <p>http://projects.webappsec.org/Cross-Site-Scripting</p>
CVE Id	http://cve.mitre.org/data/definitions/79.html
WASC Id	79
	8
High (Medium)	SQL Injection
Description	SQL injection may be possible.
URL	http://localhost:8080/examples/servlet/servlet/SessionExample;sessionId=020B9FABA402B4956AFAE3055E590B7
Parameter	dataname
Attack	ZAP AND {=1 --
Instances	1
Solution	<p>Do not trust client side input, even if there is client side validation in place.</p> <p>In general, type check all data on the server side.</p> <p>If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by "?".</p> <p>If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.</p> <p>If database Stored Procedures can be used, use them.</p> <p>Do "not" concatenate strings into queries in the stored procedure, or use "exec", "exec immediate", or equivalent functionality!</p> <p>Do not create dynamic SQL queries using simple string concatenation.</p> <p>Escape all data received from the client.</p> <p>Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input.</p> <p>Apply the principle of least privilege by using the least privileged database user possible.</p> <p>In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.</p>
Other information	<p>Grant the minimum database access that is necessary for the application.</p> <p>The page results were successfully manipulated using the boolean conditions [ZAP AND {=1 --}] and [ZAP AND {=2 --}]</p>

According to the Report, There is 2 high risk level vulnerabilities.

- XSS (Cross Site Scripting)
- SQL Injection

Now it is going to check, these vulnerabilities are really exists in this application.

XSS exploit



Solutions for XSS vulnerability.

We can validate the text boxes,

```
<script type="text/javascript">
```

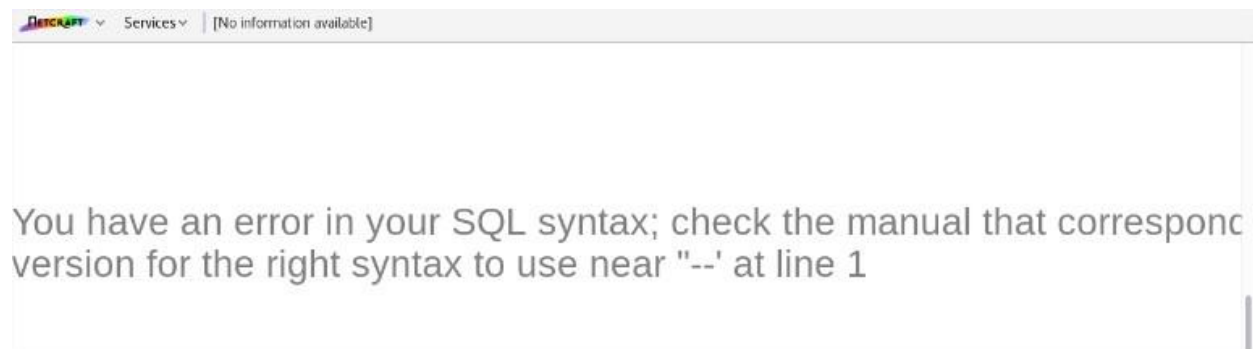
```
$(document).ready(function () {  
    $('#ctl00_topNavigation_txtSearch').keyup(function () {  
        var $th = $(this);  
        $th.val($th.val().replace(/^[^%a-zA-Z0-9 ]/g,  
        function (str) {  
            alert('Special characters not allowed except %');  
            return "";  
        }  
    }));  
});  
});
```

Following are some other solutions for XSS

- HTML escape questionable input characters like <>"'& to their equivalents (e.g., < goes to < ;). Additionally, if you needed to allow some formatting (e.g., users can submit links, insert bold text) use a safe subset of a lightweight markup language so you convert user input like [Google link] (http://www.google.com) to Google link.
- JavaScript Escape before Inserting Untrusted Data into JavaScript Data Values. [3]
- HTML escape JSON values in an HTML context and read the data with JSON.parse. [3]
- HTML Escape before Inserting Untrusted Data into HTML Element Content [3]
- CSS Escape and Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values. [3]
- Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes [3]
- URL Escape before Inserting Untrusted Data into HTML URL Parameter Values. [3]
- Sanitize HTML Markup with a Library Designed for the Job. [3]
- Prevent DOM-based XSS. [3]

Sql Injection Vulnerability verification

SQL injection is guessing or fuzzing the SQL query used in the backend server script and sending some SQL input string to the script for processing so it leads to manipulating the SQL query in the backend and hence generating suitable response. It is the topmost web application vulnerability in OWASP Top 10



Solution for These type of SQL injection.

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

➤ **Build parameterized queries**

```
txtUserId = getRequestString("UserId");  
sql = "SELECT * FROM Customers WHERE CustomerId = @0";  
command = new SqlCommand(sql);  
command.Parameters.AddWithValue("@0",txtUserID);  
command.ExecuteReader();
```

➤ **Insert Statement using php using parameterized queries**

```
$stmt = $dbh->prepare("INSERT INTO Customers (CustomerName,Address,City)  
VALUES (:nam, :add, :cit)");  
$stmt->bindParam(':nam', $txtNam);  
$stmt->bindParam(':add', $txtAdd);  
$stmt->bindParam(':cit', $txtCit);  
$stmt->execute();
```

Following are some general solutions for sql injections.

- Prepared Statements (with Parameterized Queries)Language specific recommendations:
 - Java EE – use PreparedStatement() with bind variables
 - .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables

- PHP – use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate – use createQuery() with bind variables (called named parameters in Hibernate)
- SQLite – use sqlite3_prepare() to create a statement object
- Stored Procedures.
- White List Input Validation.
- Escaping All User Supplied Input.

References

- [1] https://en.wikipedia.org/wiki/OWASP_ZAP
- [2] <http://www.guru99.com/static-dynamic-testing.html>
- [3] [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)