Vulnerability Test Report: Conducted Using OWASP ZAP

Executive Summary

Mba Agha conducted a full web application penetration test on [http://testphp.vulnweb.com]. The goal was to analyze the security posture of the web application and suggest countermeasures for all the findings requiring remediation.

The application penetration test on [http://testphp.vulnweb.com] was conducted on 29 August 2024.

As a result of the engagement, I managed to find lots of high-risk vulnerabilities, which confirmed that the security posture of the application is very low and proper security countermeasures have not been implemented in the environment.

This report contains a detailed analysis of the vulnerabilities that I found during the engagement, along with a remediation report, which would help the organization to improve the overall security posture of the application. The report also contains a detailed explanation about every vulnerability found along with detailed countermeasures to fix the vulnerability.

The overall risk of compromise was analyzed to be 80%. Addressing the security issues that are present inside the report would significantly improve the overall risk of compromise.

Remediation

The security control environment of [http://testphp.vulnweb.com] was found very poor, as a result of which there are certain security countermeasures I would like to suggest. With the goal of protecting the web application's infrastructure, I would recommend the organization perform the following actions:.

A perfect plan for fixing critical, high, medium, and low-risk vulnerabilities should be designed and implemented. The vulnerabilities should be fixed in the descending order of priority.

Cross-Site Scripting (XSS)

- **Input validation:**
  - **Sanitize user input:** Remove or encode special characters that could be interpreted as HTML or JavaScript.

    - **Use prepared statements:** For dynamic SQL queries, use prepared statements to prevent the injection of malicious code.
    - **Output encoding:** Encode output to prevent malicious code from being executed.
- **Content Security Policy (CSP):**
    - Define rules to restrict the resources that a webpage can load.
    - Use CSP to prevent the execution of inline scripts or scripts from unauthorized sources.

SQL Injection

- **Prepared Statements:**
    - Use parameterized queries to separate the SQL statement from the data values.
    - This prevents the injection of malicious SQL code.
- **Input validation:**
    - Validate user input to ensure it adheres to expected formats and data types.
    - Prevent the injection of unexpected characters or control sequences.
- **Least Privilege Principle:**
    - Grant database users only the minimum necessary privileges to perform their tasks.
    - This limits the potential damage if an attacker gains unauthorized access.

SQL Injection Specific to MySQL

- **Escape Special Characters:**
    - Use MySQL's escape() function to escape special characters within strings.
    - This prevents the injection of malicious SQL code.
- **Stored Procedures:**
    - Encapsulate SQL queries within stored procedures to centralize security and improve performance.
- **MySQL Strict Mode:**
    - Enable strict mode to enforce data type compatibility and prevent implicit conversions.
    - This can help catch potential SQL injection errors.

Clickjacking

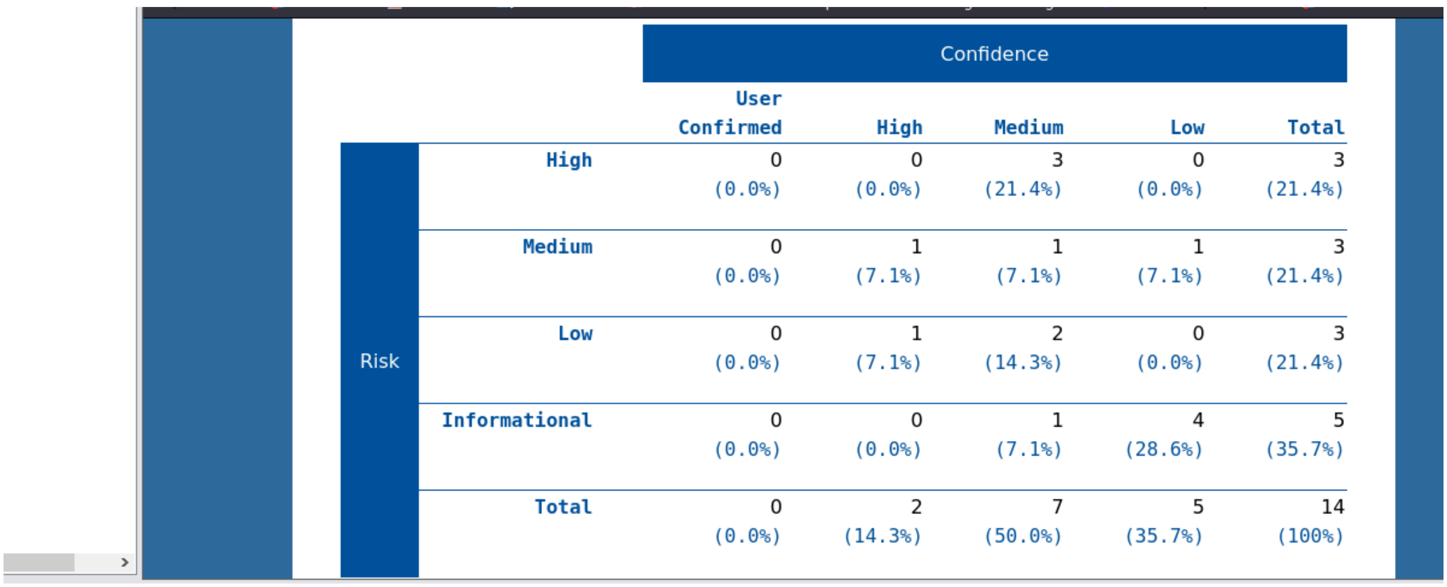
- **Framebusting:**
    - Use techniques like HTTP headers (X-Frame-Option) or JavaScript to prevent a webpage from being embedded in an iframe.
    - This prevents clickjacking attacks.
- **User Interface Design:**
    - Design user interfaces to minimize the risk of accidental clicks or confusion.
    - Use clear visual indicators to distinguish between legitimate and malicious content.

General Site Security

- **Regular Updates:**
    - Keep software and libraries up-to-date to address known vulnerabilities.
    - Apply security patches promptly.
- **Secure Configuration:**

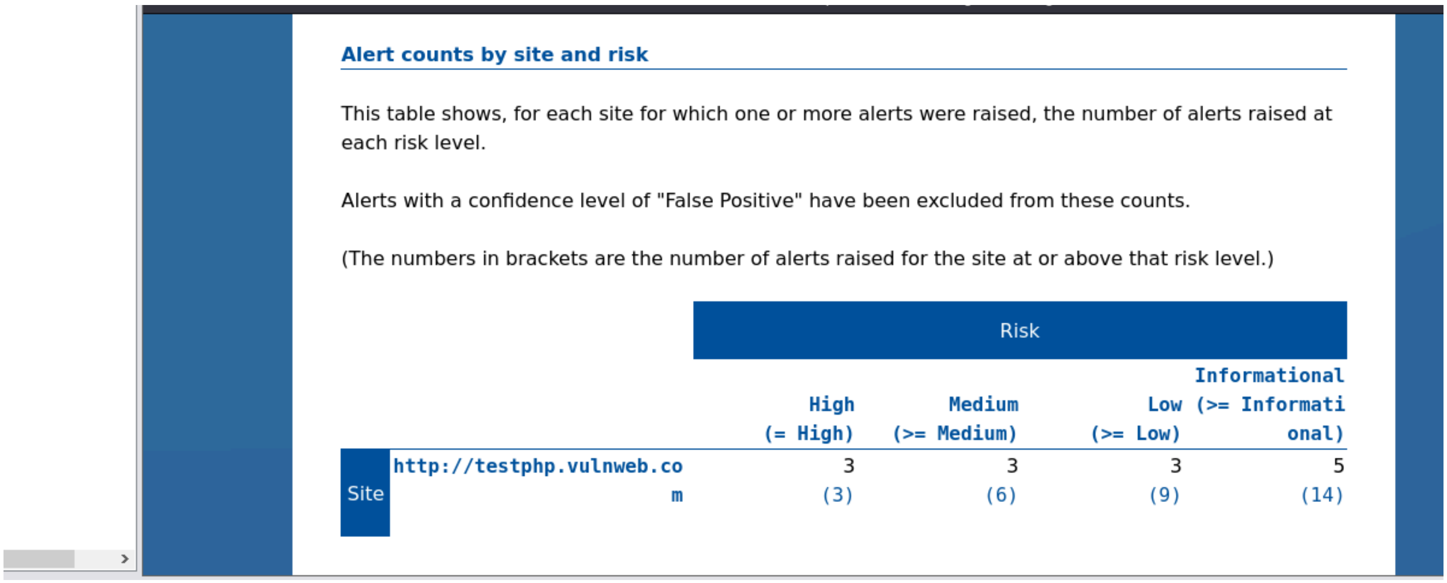
  - Follow best practices for configuring web servers, databases, and other components.
  - Disable unnecessary features and services.
- **Strong authentication:**
  - Implement strong password policies and multi-factor authentication.
  - Protect user credentials.
- **Secure Communication:**
  - Use HTTPS to encrypt data transmitted between the server and client.
  - Implement SSL/TLS certificates.
- **Regular Testing:**
  - Conduct vulnerability assessments and penetration testing to identify and address security weaknesses.
- **Security Awareness Training:**
  - Educate employees about security best practices and the risks of phishing, social engineering, and other attacks.

Vulnerability Assessment Summary



		Confidence				
		User Confirmed	High	Medium	Low	Total
Risk	High	0 (0.0%)	0 (0.0%)	3 (21.4%)	0 (0.0%)	3 (21.4%)
	Medium	0 (0.0%)	1 (7.1%)	1 (7.1%)	1 (7.1%)	3 (21.4%)
	Low	0 (0.0%)	1 (7.1%)	2 (14.3%)	0 (0.0%)	3 (21.4%)
	Informational	0 (0.0%)	0 (0.0%)	1 (7.1%)	4 (28.6%)	5 (35.7%)
	Total	0 (0.0%)	2 (14.3%)	7 (50.0%)	5 (35.7%)	14 (100%)

Tabular Summary



Alert counts by site and risk

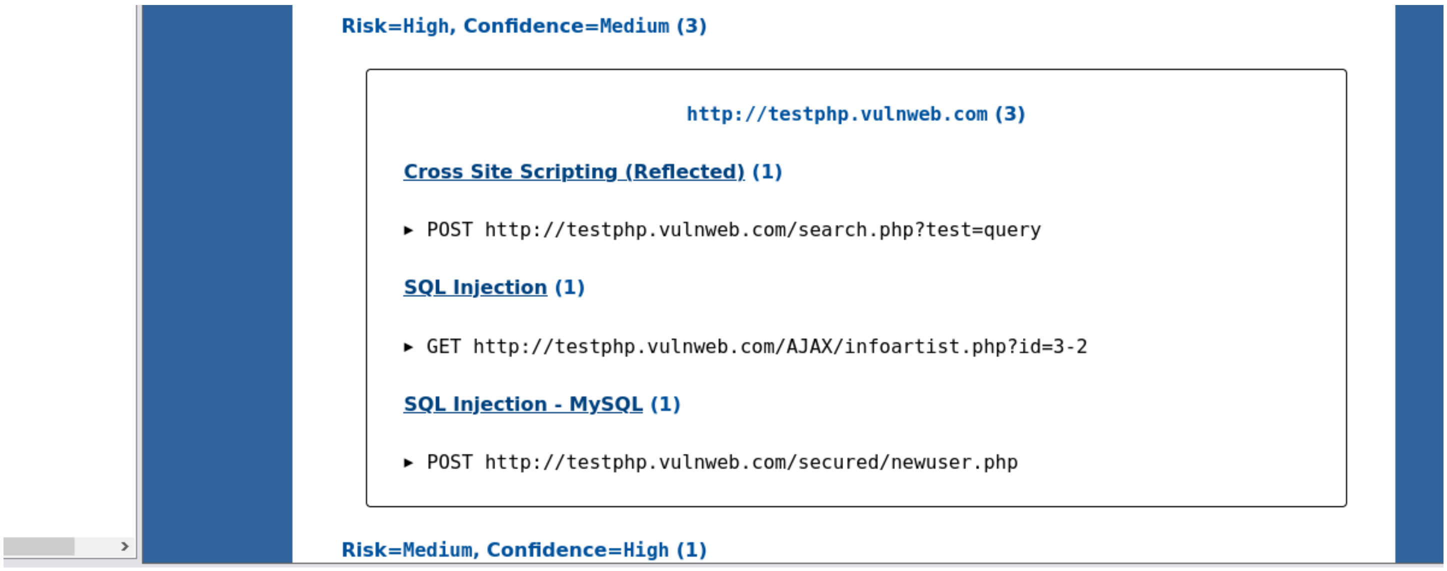
This table shows, for each site for which one or more alerts were raised, the number of alerts raised at each risk level.

Alerts with a confidence level of "False Positive" have been excluded from these counts.

(The numbers in brackets are the number of alerts raised for the site at or above that risk level.)

		Risk			
		High (= High)	Medium (>= Medium)	Low (>= Low)	Informational (>= Informational)
Site	http://testphp.vulnweb.com	3 (3)	3 (6)	3 (9)	5 (14)

Risk Assessment



Risk=High, Confidence=Medium (3)

http://testphp.vulnweb.com (3)

Cross Site Scripting (Reflected) (1)

▸ POST http://testphp.vulnweb.com/search.php?test=query

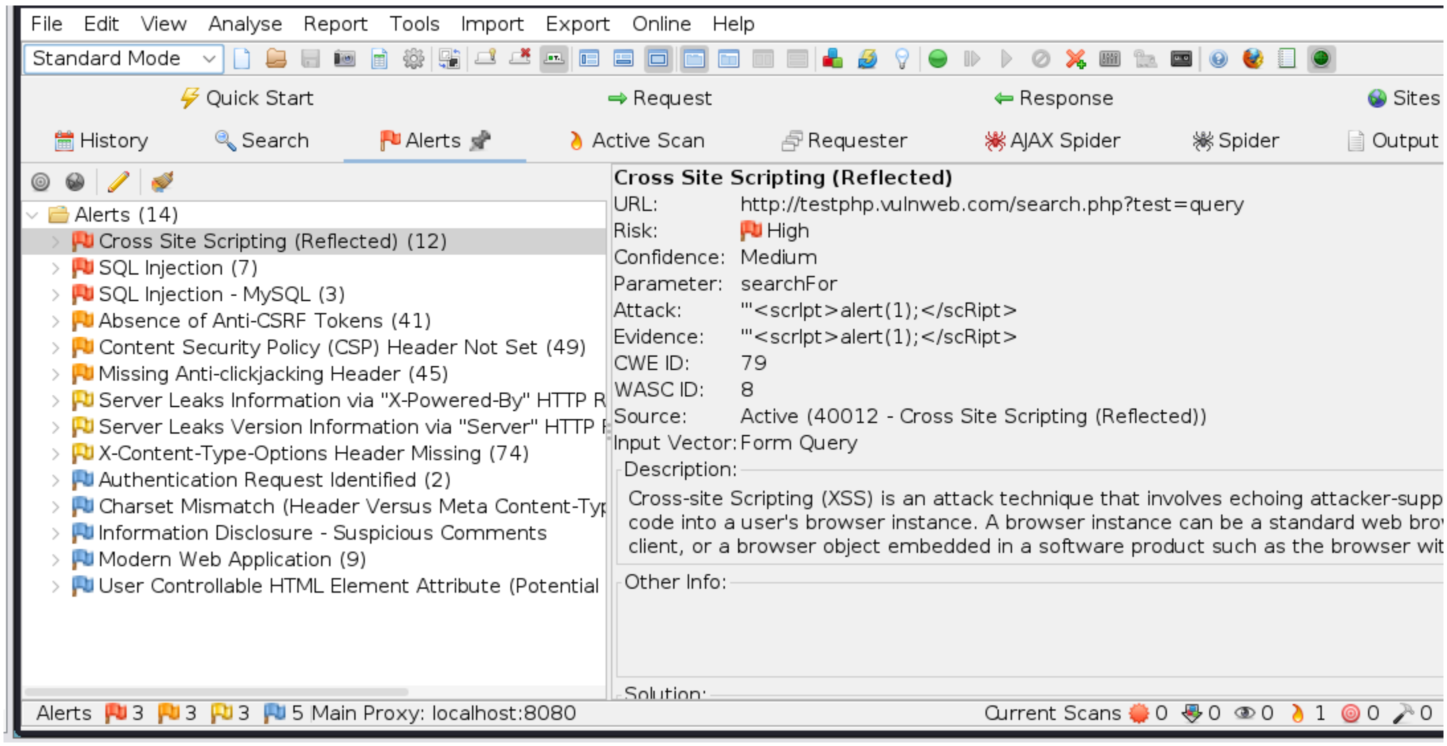
SQL Injection (1)

▸ GET http://testphp.vulnweb.com/AJAX/infoartist.php?id=3-2

SQL Injection - MySQL (1)

▸ POST http://testphp.vulnweb.com/secured/newuser.php

Risk=Medium, Confidence=High (1)



```
Alerts (14)
  Cross Site Scripting (Reflected) (12)
  SQL Injection (7)
  SQL Injection - MySQL (3)
  Absence of Anti-CSRF Tokens (41)
  Content Security Policy (CSP) Header Not Set (49)
  Missing Anti-clickjacking Header (45)
  Server Leaks Information via "X-Powered-By" HTTP R
  Server Leaks Version Information via "Server" HTTP R
  X-Content-Type-Options Header Missing (74)
  Authentication Request Identified (2)
  Charset Mismatch (Header Versus Meta Content-Typ
  Information Disclosure - Suspicious Comments
  Modern Web Application (9)
  User Controllable HTML Element Attribute (Potential
```

Cross Site Scripting (Reflected)

URL:	http://testphp.vulnweb.com/search.php?test=query
Risk:	High
Confidence:	Medium
Parameter:	searchFor
Attack:	'"<scrIpt>alert(1);</scRipt>
Evidence:	'"<scrIpt>alert(1);</scRipt>
CWE ID:	79
WASC ID:	8
Source:	Active (40012 - Cross Site Scripting (Reflected))
Input Vector:	Form Query

Description:

Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supp code into a user's browser instance. A browser instance can be a standard web brow client, or a browser object embedded in a software product such as the browser wit

Other Info:

Solution:

Alerts 3 3 3 5 Main Proxy: localhost:8080 Current Scans 0 0 0 1 0 0

Methodology

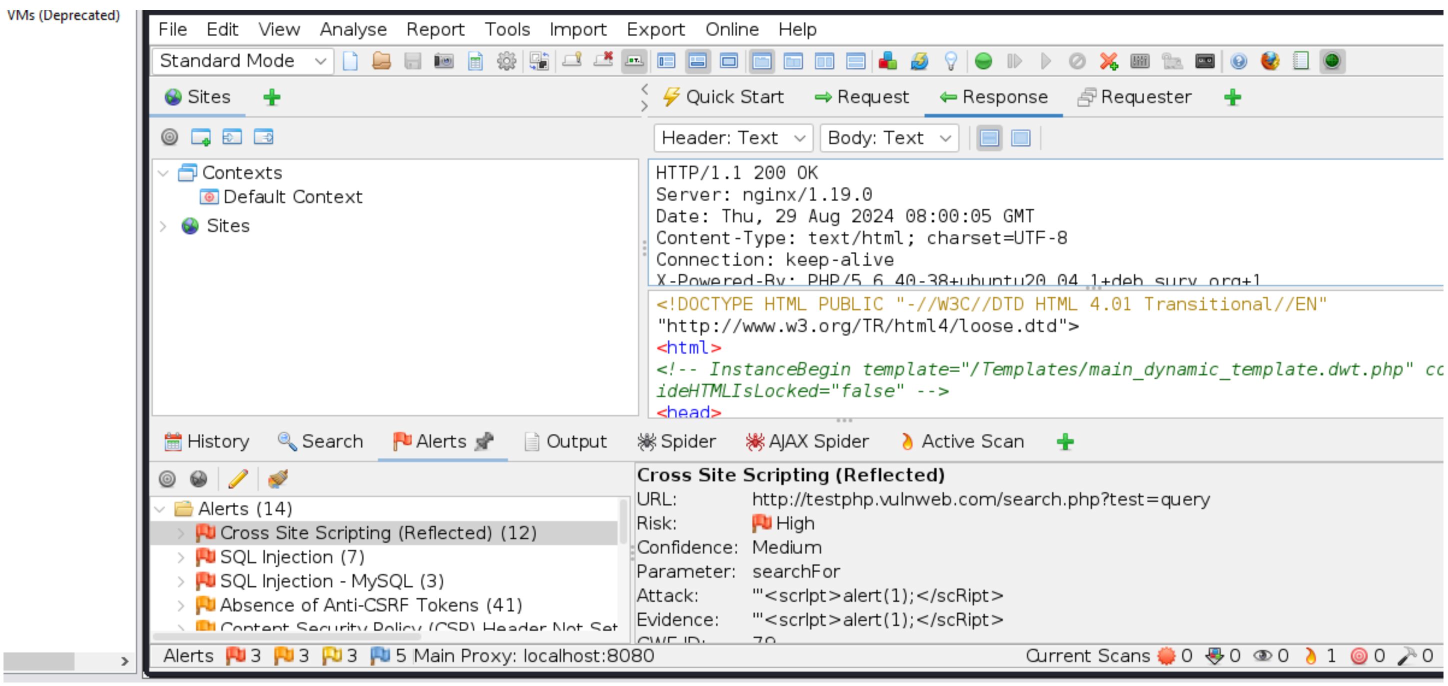
I utilized the NIST methodology in this engagement against the targets within the [http://testphp.vulnweb.com]. The methodology focuses on assessing the security posture of the target network in order to create an effective and better security posture.

NIST penetration test methodology

1. Planning
2. Discovery
3. Attack
4. Additional discovery
5. Reporting

Detailed Findings

Cross-Site Scripting



```
HTTP/1.1 200 OK
Server: nginx/1.19.0
Date: Thu, 29 Aug 2024 08:00:05 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
X-Powered-By: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<!-- InstanceBegin template="/Templates/main_dynamic_template.dwt.php" cc
ideHTMLIsLocked="false" -->
<head>
```

Alerts (14)
- Cross Site Scripting (Reflected) (12)
- SQL Injection (7)
- SQL Injection - MySQL (3)
- Absence of Anti-CSRF Tokens (41)
- Content Security Policy (CSP) Header Not Set

Cross Site Scripting (Reflected)

URL:	http://testphp.vulnweb.com/search.php?test=query
Risk:	High
Confidence:	Medium
Parameter:	searchFor
Attack:	'"<scrIpt>alert(1);</scRipt>
Evidence:	'"<scrIpt>alert(1);</scRipt>
CWE ID:	79

Alerts 3 3 3 5 Main Proxy: localhost:8080 Current Scans 0 0 0 1 0 0

HTTP/1.1 200 OK

Server: nginx/1.19.0

Date: Thu, 29 Aug 2024 08:00:05 GMT

Content-Type: text/html; charset=UTF-8

Connection: keep-alive

X-Powered-By: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1

content-length: 4797

Descriptions

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.

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.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

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 executed. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

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.

Solutions

Phase: Architecture and Design

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

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.

Phases: Implementation; Architecture and Design

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.

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.

Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design

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 CWE-602. 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.

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 this capability at every point where output is generated.

Phase: Implementation

For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoder is not specified, the web browser may choose a different encoder by guessing which encoder 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 CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be Http Only. In browsers that support the Http Only 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 cookies. This is not a complete solution, since Http Only is not supported by all browsers. More importantly, XMLHTTP

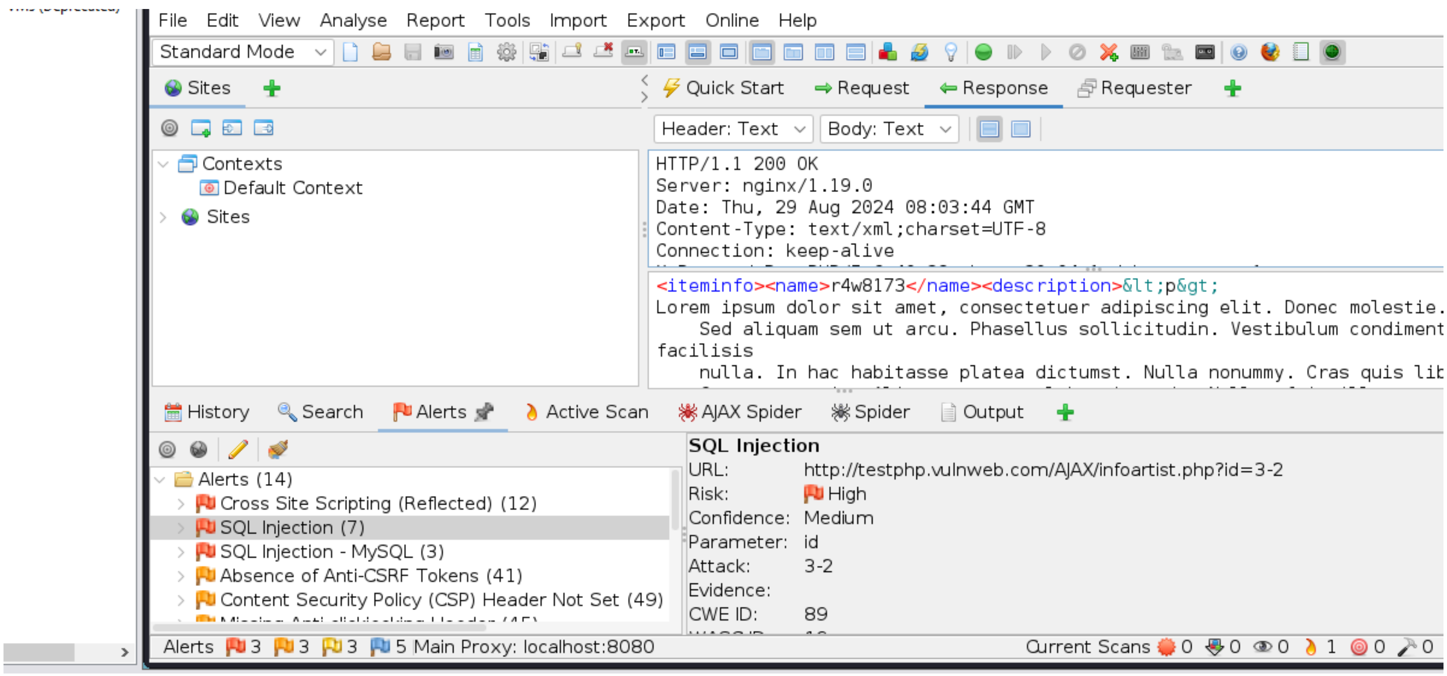
Request and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the Http Only flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list 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 deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

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, "boat" 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."

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.

SQL Injection



```
HTTP/1.1 200 OK
Server: nginx/1.19.0
Date: Thu, 29 Aug 2024 08:03:44 GMT
Content-Type: text/xml;charset=UTF-8
Connection: keep-alive
```

```
<iteminfo><name>r4w8173</name><description>&lt;p&gt;
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec molestie.
    Sed aliquam sem ut arcu. Phasellus sollicitudin. Vestibulum condiment
facilisis
    nulla. In hac habitasse platea dictumst. Nulla nonummy. Cras quis lib
```

Alerts (14)
- Cross Site Scripting (Reflected) (12)
- SQL Injection (7)
- SQL Injection - MySQL (3)
- Absence of Anti-CSRF Tokens (41)
- Content Security Policy (CSP) Header Not Set (49)

SQL Injection

URL:	http://testphp.vulnweb.com/AJAX/infoartist.php?id=3-2
Risk:	High
Confidence:	Medium
Parameter:	id
Attack:	3-2
Evidence:	
CWE ID:	89

Alerts 3 3 3 5 Main Proxy: localhost:8080 Current Scans 0 0 0 1 0 0

HTTP/1.1 200 OK

Server: nginx/1.19.0

Date: Thu, 29 Aug 2024 08:03:44 GMT

Content-Type: text/xml;charset=UTF-8

Connection: keep-alive

```
X-Powered-By: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1
content-length: 1343

<iteminfo><name>r4w8173</name><description>&lt;p&gt;
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec molestie.
    Sed aliquam sem ut arcu. Phasellus sollicitudin. Vestibulum condimentum facilisis
    nulla. In hac habitasse platea dictumst. Nulla nonummy. Cras quis libero.
    Cras venenatis. Aliquam possessore lobortis pede. Nullam fringilla urna id leo.
    Praesent aliquet pretium erat. Praesent non odio. Pellentesque a magna a
    mauris vulputate lacinia. Aenean viverra. Class aptent taciti sociosqu ad
    litora torquent per conubia nostra, per inceptos hymenaeos. Aliquam lacus.
    Mauris magna eros, semper a, tempor et, rutrum et, tortor.
&lt;/p&gt;
&lt;p&gt;
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec molestie.
    Sed aliquam sem ut arcu. Phasellus sollicitudin. Vestibulum condimentum facilisis
    nulla. In hac habitasse platea dictumst. Nulla nonummy. Cras quis libero.
    Cras venenatis. Aliquam posuere lobortis pede. Nullam fringilla urna id leo.
    Praesent aliquet pretium erat. Praesent non odio. Pellentesque a magna a
    mauris vulputate lacinia. Aenean viverra. Class aptent taciti sociosqu ad
    litora torquent per conubia nostra, per inceptos hymenaeos. Aliquam lacus.
    Mauris magna eros, semper a, tempor et, rutrum et, tortor.
&lt;/p&gt;</description></iteminfo>
```

Description

SQL injection may be possible

The original page results were successfully replicated using the expression [3-2] as the parameter value

The parameter value being modified was stripped from the HTML output for the purposes of the comparison

Solution

Do not trust client side input, even if there is client side validation in place.

In general, type check all data on the server side.

If the application uses JDBC, use Prepared Statement or Callable Statement, with parameters passed by '?'

If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.

If database Stored Procedures can be used, use them.

Do not concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!

Do not create dynamic SQL queries using simple string concatenation.

Escape all data received from the client.

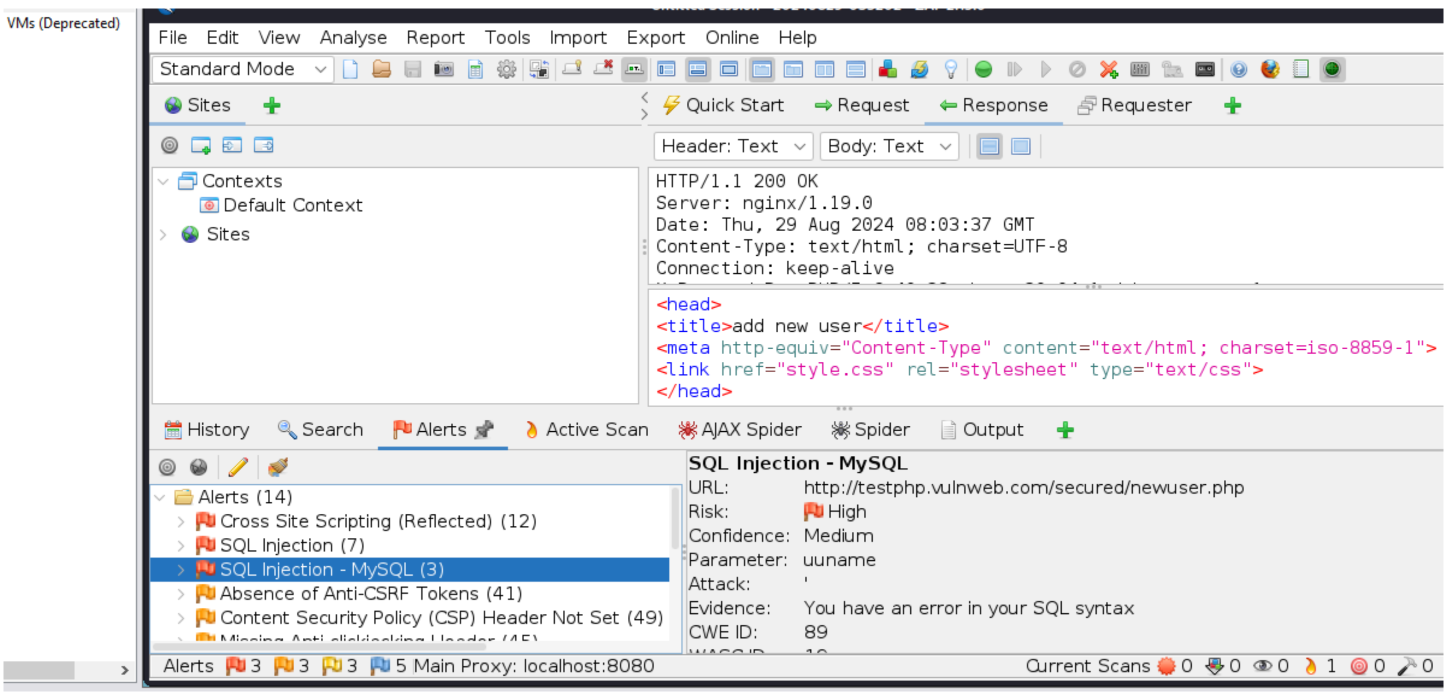
Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input.

Apply the principle of least privilege by using the least privileged database user possible.

In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.

Grant the minimum database access that is necessary for the application.

SQL Injection - MySQL



```
HTTP/1.1 200 OK
Server: nginx/1.19.0
Date: Thu, 29 Aug 2024 08:03:37 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
```

```
<head>
<title>add new user</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link href="style.css" rel="stylesheet" type="text/css">
</head>
```

Alerts (14)
- Cross Site Scripting (Reflected) (12)
- SQL Injection (7)
- SQL Injection - MySQL (3)
- Absence of Anti-CSRF Tokens (41)
- Content Security Policy (CSP) Header Not Set (49)

SQL Injection - MySQL

URL:	http://testphp.vulnweb.com/secured/newuser.php
Risk:	High
Confidence:	Medium
Parameter:	uuname
Attack:	'
Evidence:	You have an error in your SQL syntax
CWE ID:	89

Alerts 3 3 3 5 Main Proxy: localhost:8080 Current Scans 0 0 0 1 0 0

HTTP/1.1 200 OK

Server: nginx/1.19.0

Date: Thu, 29 Aug 2024 08:03:37 GMT

Content-Type: text/html; charset=UTF-8

Connection: keep-alive

X-Powered-By: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1

content-length: 570

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<title>add new user</title>

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">

```
<link href="style.css" rel="stylesheet" type="text/css">

</head>

<body>

<div id="masthead">

  <h1 id="siteName">ACUNETIX ART</h1>

</div>

<div id="content">

    Unable to access user database: You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use near '''' at line 1
```

Description

SQL injection may be possible.

RDBMS [MySQL] likely, given error message regular expression [\QYou have an error in your SQL syntax\E] matched by the HTML results.

The vulnerability was detected by manipulating the parameter to cause a database error message to be returned and recognised

Solution

Do not trust client side input, even if there is client side validation in place.

In general, type check all data on the server side.

If the application uses JDBC, use Prepared Statement or Callable Statement, with parameters passed by '?'

If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.

If database Stored Procedures can be used, use them.

Do not concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!

Do not create dynamic SQL queries using simple string concatenation.

Escape all data received from the client.

Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input.

Apply the principle of least privilege by using the least privileged database user possible.

In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.

Grant the minimum database access that is necessary for the application.

Prepared By: Mba Agha

Date: 29th, August 2024