

## **Foundational Web Application Security & Vulnerability Scanning**

Nicholas Massei

Cybersecurity Project/Lab

July 12, 2025

## Introduction

During this project, I wanted to focus on identifying and exploiting web application vulnerabilities – as mentioned in the objectives, focusing on SQL Injection and Cross-Site Scripting (XSS) using tools like bWAPP, WebGoat, and ZAP. My goal was to understand these vulnerabilities in practice and learn mitigation techniques.

## WebGoat Environment Setup

To begin, I installed Docker to run WebGoat on my host machine – and later just installed WebGoat on my Kali Linux VM. WebGoat is an intentionally vulnerable web application including its own lessons for practicing techniques, and I accessed it via browser at <http://localhost:8080/WebGoat>.

The platform offers guided modules that simulate real attack scenarios, which made it super easy to engage with and test techniques like SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF) – all are techniques I have become familiar with beforehand.

Logging in and exploring these modules on WebGoat provided a controlled space to break things, learn from mistakes, and better understand how attackers think. So far, I am just thinking about how incredible this tool is for learning and sandboxing.

## Background Information

I then used an incredibly helpful YouTube tutorial titled “*Automated Hacking Tool?! | OWASP ZAP Tutorial*” to deepen my understanding of ZAP and how to properly interpret the results it was giving me. This resource helped me move beyond just passively running scans -- it taught me how to read alerts, understand their risk levels, and even explore follow-up manual testing based on what was discovered. With this video, I was able to take more **intentional** steps in using ZAP as an actual penetration testing tool, not just an automated scanner.

The tutorial specifically connected to vulnerabilities like Cross-Site Scripting (**XSS**), **SQL Injection**, and Cross-Site Request Forgery (**CSRF**) by demonstrating how ZAP detects these issues and what manual techniques can be used to exploit and confirm them. This made it much easier to grasp the practical implications of the vulnerabilities I encountered in WebGoat and understand how attackers might leverage them. It also showed me that ZAP definitely pulls false negatives (which should be expected) so it is good to verify every vulnerability ZAP finds.

**Reference:** [https://youtu.be/QJ5u\\_dHwoAk?si=1si6gh2HH\\_V62lAZ](https://youtu.be/QJ5u_dHwoAk?si=1si6gh2HH_V62lAZ)

### **Basic Vulnerability Analysis**

Now to start getting hands-on practice. I have everything set up and am ready to manually explore WebGoat's lessons and the site itself, actively investigating and exploiting vulnerabilities while using ZAP as my primary tool for identifying security weaknesses. While going through the lessons for SQL injection, XSS, and CSRF on WebGoat I had ZAP manually configured to log, spider, and further vulnerability scan using, and it obviously logged more than one example of each of these vulnerabilities – there were many more to explore but I will focus on these for this task. The alerts listed gave me solid proof of each vulnerability.

- **SQL Injection** and **SQL Injection - Hypersonic SQL** – SQL Injection
- **Vulnerable JS Library** – XSS
- **Absence of Anti-CSRF Tokens** – CSRF

Evidence screenshotted and included on the last page(s).

## Exploring Vulnerabilities

After reviewing ZAP's alert descriptions and with the lessons I went through on WebGoat, I deepened my understanding of how each vulnerability functions – not just in theory, but even in the context of a real (insecure) application.

- For **SQL Injection** (Windows – WebGoat), I used inputs in the lessons such as the first one where I used “SELECT department FROM employees WHERE first\_name = 'Bob'” to retrieve data without authentication. ZAP's alert for "SQL Injection – Hypersonic SQL" confirmed the potential impact of insecure query handling and helped reinforce how backend logic can be manipulated with crafted input.

- For **XSS** (Windows – WebGoat), I explored where simple JavaScript payloads like `<script>alert('XSS')</script>` could be submitted — particularly in comment or input fields. While WebGoat provides safe examples, ZAP identified the usage of an outdated version of Underscore.js, linked to a known CVE (CVE-2021-23358, as listed in the description), which supports the potential for script injection attacks.

- For **CSRF** (Windows – WebGoat), ZAP detected the absence of anti-CSRF tokens in many requests. Without these tokens, an attacker could forge state-changing requests from a victim's browser. This aligns closely with what I learned when going through WebGoat's CSRF lessons. Each of these findings was confirmed through both ZAP's logging, automated scanning, and manual interaction with the WebGoat interface.

## Using Kali Linux VM with ZAP in bWAPP to explore SQL Injection and XSS further

Attempting to manually exploit a site (with consent - bWAPP) using techniques I have learned.

**bWAPP** (buggy web application) is an intentionally vulnerable app -- like WebGoat, but different.

- **SQL Injection:** For SQL Injection (Search/GET – bWAPP on Kali), I used payloads like ' OR 1=1 # to bypass search filters and retrieve the entire dataset from the backend without proper validation. The success of this injection demonstrated how vulnerable queries can be exploited to expose all records. This further reinforced the importance of sanitizing input and using parameterized queries to defend against unauthorized data access.
- **Cross-Site Scripting XSS:** For XSS testing on bWAPP, I used payload '><script>alert('You got hacked!')</script>' and injected it. This successfully triggered an alert, demonstrating that the application failed to properly sanitize user input before rendering it on the page. This vulnerability highlights the risk of attackers injecting malicious scripts that can steal user data or perform unauthorized actions. It further emphasizes the need for input validation to prevent such attacks.

Evidence screenshotted and included on the last page(s) as always.

### **Challenges Faced**

Most tasks were clear and manageable, though I faced some formatting issues with documentation and had to familiarize myself with new security tools and set ups. I addressed these by researching best practices and searching for help when needed.

### **Report**

**SQL Injection:** Bypassed input validation to access unauthorized data.

**Cross-Site Scripting (XSS):** Executed malicious scripts via un-sanitized user input.

**Cross-Site Request Forgery (CSRF):** Missing anti-CSRF tokens would allow unauthorized actions.

## Possible Simple Mitigations

**Input Sanitization & Parameterized Queries:** Prevent SQL injection by using prepared statements and validating user inputs rigorously.

**Output Encoding & Content Security Policy (CSP):** Mitigate XSS by encoding output, sanitizing input, and implementing CSP headers to restrict script execution.

**Implement Anti-CSRF Tokens:** Include unique tokens in state-changing requests to verify legitimate user actions and block forged requests.

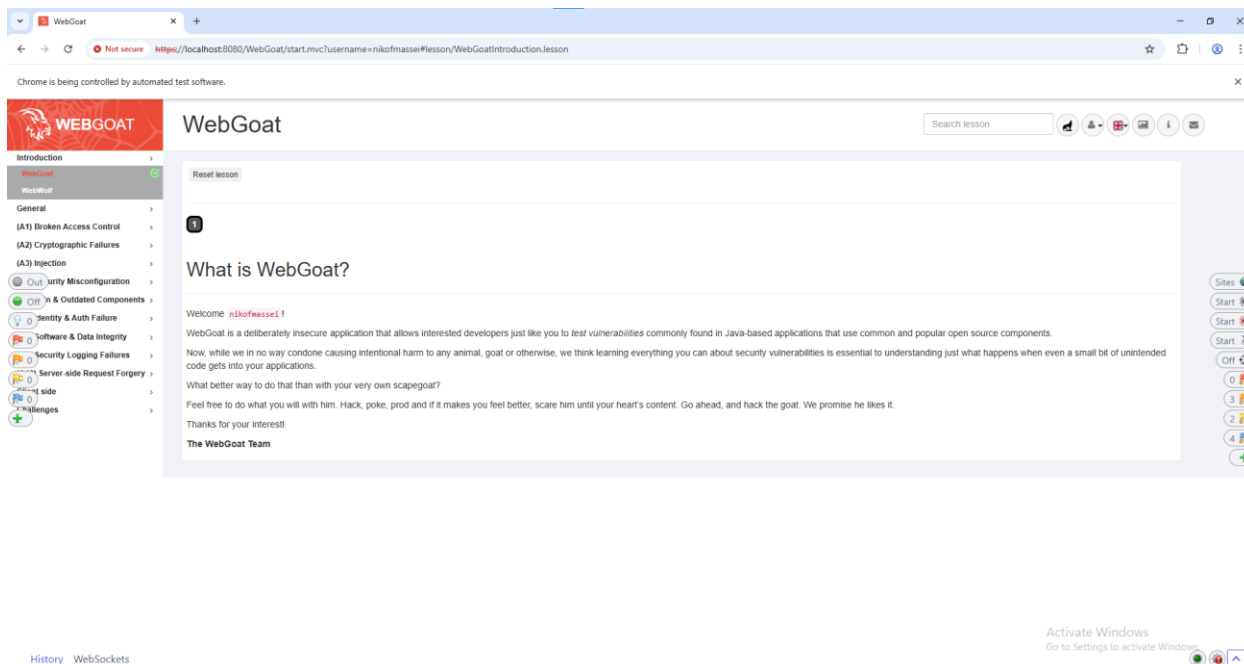
Reference:

OWASP Foundation. *OWASP Cheat Sheet Series*. <https://cheatsheetseries.owasp.org/index.html>

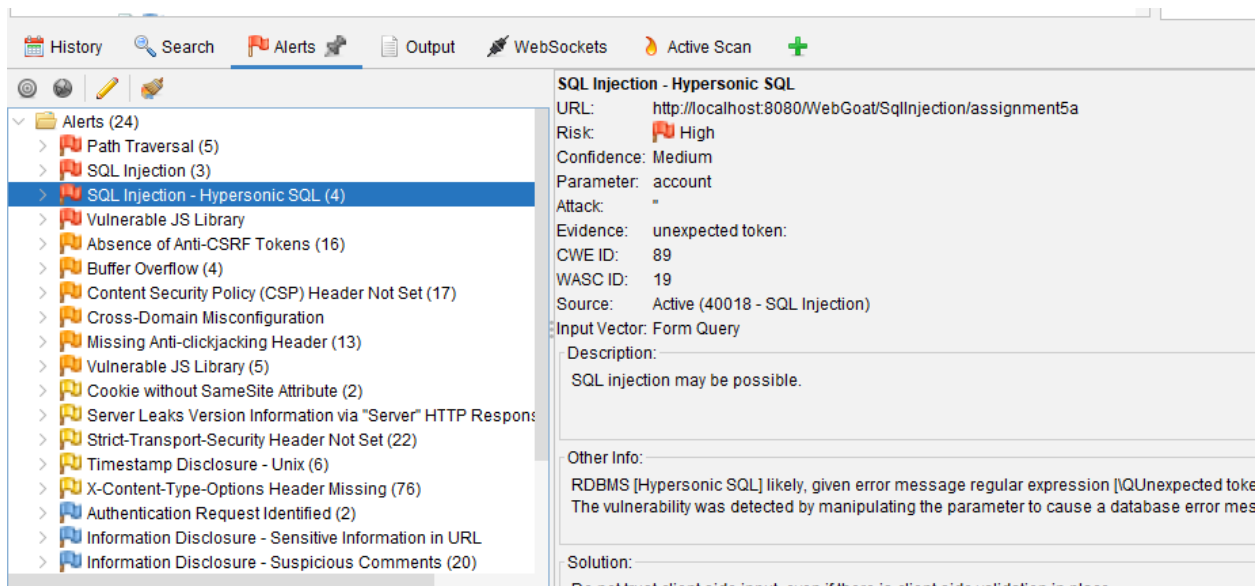
## Conclusion

This project has been a great learning experience that took my web vulnerability skills from theory to hands-on application. Working through real vulnerabilities like SQL injection, cross-site scripting, and CSRF while using tools such as WebGoat, ZAP, and Wireshark allowed me to better understand both offensive **and** defensive techniques. From network traffic monitoring to vulnerability scanning and safe configuration practices, every task showed the importance of layered security and its implementation.

# Windows - WebGoat



*Logged in & running WebGoat successfully. Manual explore active via ZAP (with HUD enabled)*



*Alerts were found after manual interaction with all WebGoat lessons and active scan in the ZAP GUI. Used the HUD in WebGoat to spider and scan.*

# SQL Injection & XSS through WebGoat

The screenshot shows the 'SQL Injection (intro)' lesson page. The left sidebar contains a navigation menu with items like 'SQL Injection (intro)', 'SQL Injection (advanced)', 'Cross Site Scripting', etc. The main content area is titled 'Data Manipulation Language (DML)'. It explains that DML statements include SELECT, INSERT, UPDATE, and DELETE. An example query is provided: `SELECT phone FROM employees WHERE userid = 96134;`. Below the text, a 'Submit' button is shown, and a success message reads: 'Congratulations. You have successfully completed the assignment.' A table displays the result of the query:

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN	PHONE
89752	Tobi	Barnett	Sales	77000	TAS9LL1	null

At the bottom right, there is a watermark for 'Activate Windows'.

*A later example of using SQL Injection to manipulate a database in WebGoat via host machine*

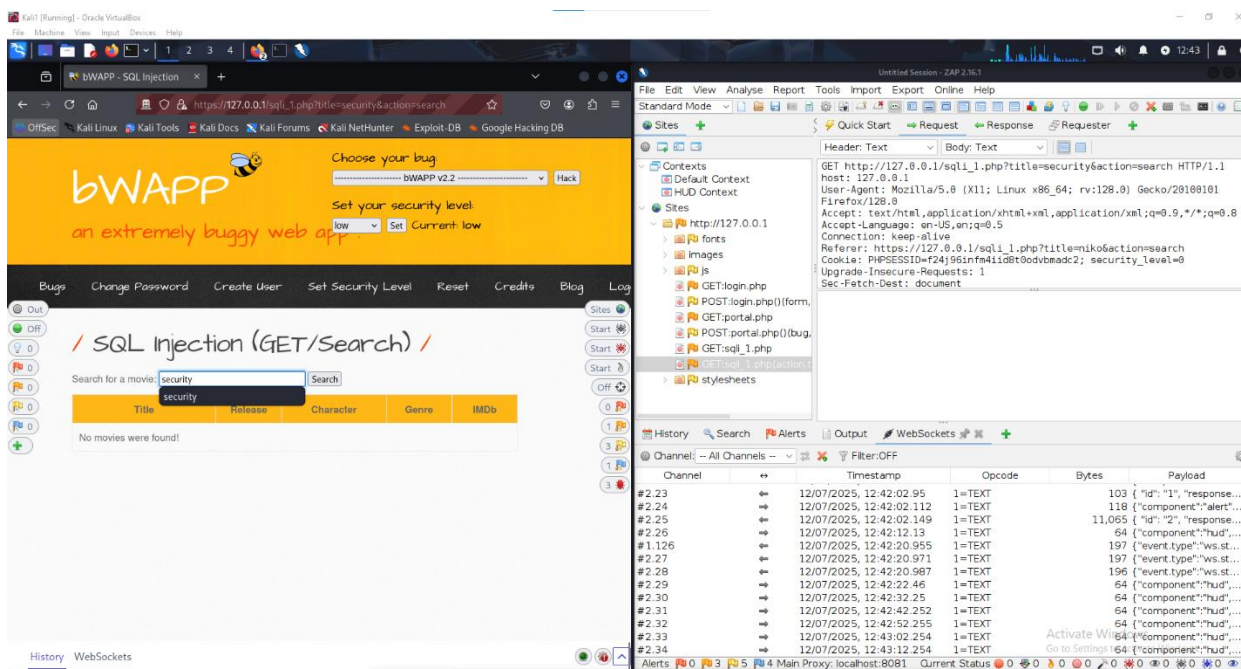
The screenshot shows the 'Try It! Reflected XSS' lesson page. The left sidebar contains a navigation menu with items like 'Introduction', 'General', '(A1) Broken Access Control', etc. The main content area is titled 'Try It! Reflected XSS'. It explains that XSS occurs when unvalidated user input gets used in an HTTP response. Below the text, a 'Shopping Cart' section is shown with a table of items:

Shopping Cart Items -- To Buy Now	Price	Quantity	Total
Studio RTA - Laptop/Reading Cart with Tilling Surface - Cherry	69.99	1	\$0.00
Dynex - Traditional Notebook Case	27.99	1	\$0.00
Hewlett-Packard - Pavilion Notebook with Intel Centrino	1599.99	1	\$0.00
3 - Year Performance Service Plan \$1000 and Over	299.99	1	\$0.00

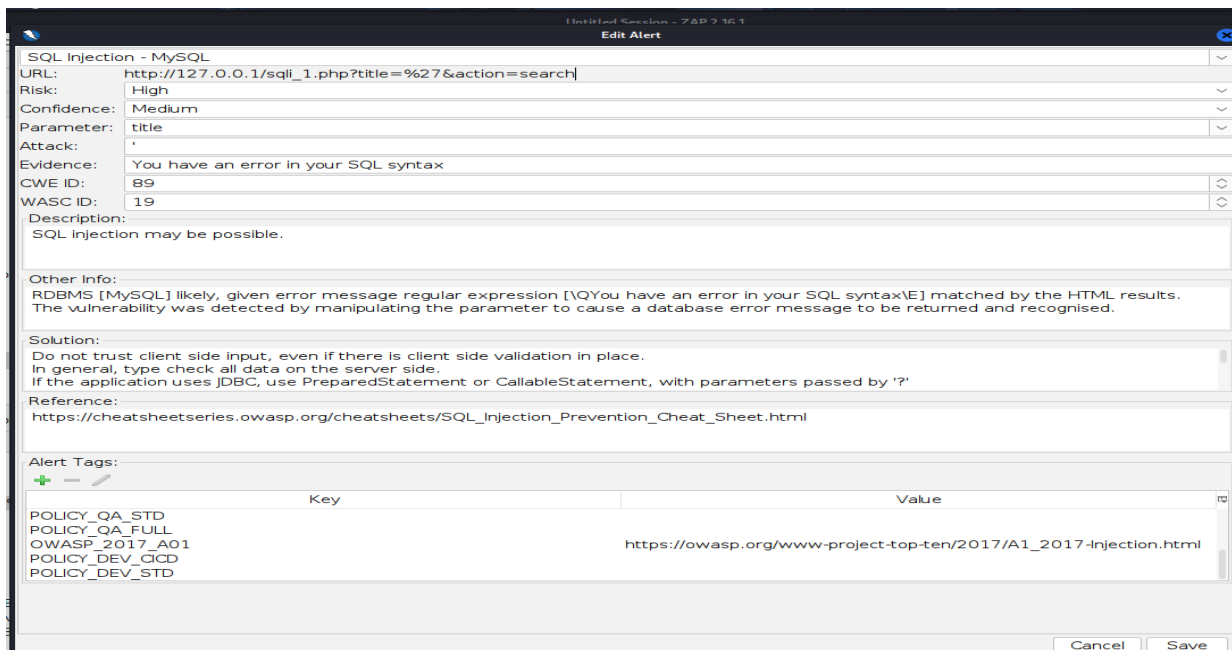
Below the table, there is a form with two input fields: 'Enter your credit card number:' and 'Enter your three digit access code:'. The 'credit card number' field contains the payload `<script>alert('XSS')</script>`. The 'access code' field contains '111'. A 'Purchase' button is visible. Below the form, a success message reads: 'We have charged credit card 4128 3214 0002 1999 \$1997.96'. At the bottom right, there is a watermark for 'Activate Windows'.

*XSS attack successful in a later lesson in WebGoat via host machine*

# Kali – bWAPP



*Normal input through bWAPP to log in ZAP for automated attacks and alerts – as seen in the HTTP request*



*Possible SQL Injection details listed from active scan*

# SQL Injection

The screenshot shows the bwAPP interface with a search bar containing a SQL injection payload. The search results table is empty, and an error message is displayed below it.

Search for a movie:  Search

Title	Release	Character	Genre	IMDb
Error: 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				

I followed the HTTP address for the possible injection, and this confirms that SQL Injection can be exploited

The screenshot shows the bwAPP interface with a search bar containing a SQL injection payload. The search results table displays a list of movies.

Search for a movie:  Search

Title	Release	Character	Genre	IMDb
G.I. Joe: Retaliation	2013	Cobra Commander	action	<a href="#">Link</a>
Iron Man	2008	Tony Stark	action	<a href="#">Link</a>
Man of Steel	2013	Clark Kent	action	<a href="#">Link</a>
Terminator Salvation	2009	John Connor	sci-fi	<a href="#">Link</a>
The Amazing Spider-Man	2012	Peter Parker	action	<a href="#">Link</a>
The Cabin in the Woods	2011	Some zombies	horror	<a href="#">Link</a>
The Dark Knight Rises	2012	Bruce Wayne	action	<a href="#">Link</a>
The Fast and the Furious	2001	Brian O'Connor	action	<a href="#">Link</a>

After attempting multiple common SQL injections, "' OR 1=1 #' worked!

# XSS

**Edit Alert**

S

URL: `http://127.0.0.1/sqli_1.php?title=%27%22%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E&action=search`

Risk: High

Confidence: Medium

Parameter: title

Attack: `"<script>alert(1);</script>`

Evidence: `"<script>alert(1);</script>`

CWE ID: 79

WASC ID: 8

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

Other Info:

Solution:  
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.

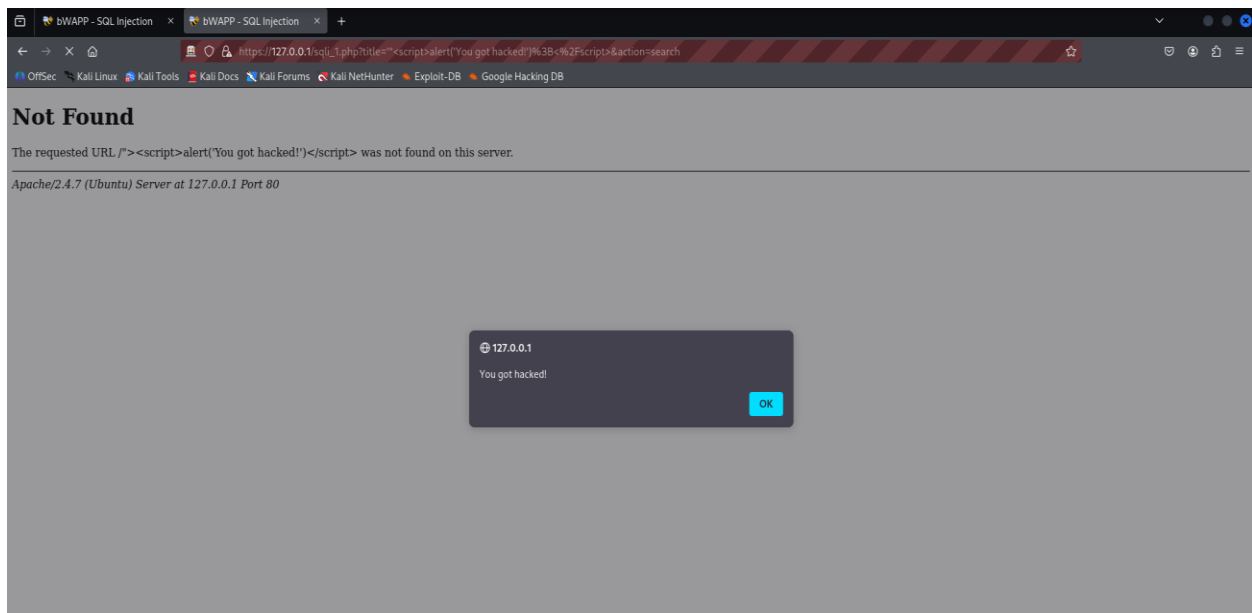
Reference:  
<https://owasp.org/www-community/attacks/xss/>  
<https://cwe.mitre.org/data/definitions/79.html>

Alert Tags:

Key	Value
POLICY_QA_FULL	
WSTG-V42-INPV-01	<a href="https://owasp.org/www-project-web-security-testing-guide...">https://owasp.org/www-project-web-security-testing-guide...</a>

Cancel Save

*XSS (Reflected) details form auto scan. Following HTTP and confirming vulnerability.*



*Vulnerability is confirmed. Changed payload from displaying – as seen in the ZAP details – “1” to “You got hacked!”*