Baruch COLLEGE
The City University of New York

NYC Tech Talent Pipeline

Exposing and Securing Web Application Vulnerabilities

OWASP Juice Shop Case Study

TTPR Capstone Report

August 2025

Team#10

Aiden Yeung

Kairos Liang

Adrian Davis

Table of Contents

Figures

List of Acronyms & Abbreviations

API: Application Programming Interface

CWE: Common Weakness Enumeration

CVSS: Common Vulnerability Scoring System

DAST: Dynamic Application Security Testing

HTML: HyperText Markup Language

IDOR: Insecure Direct Object Reference

NLP: Natural Language Processing

OWASP: Open Web Application Security Project

SQL: Structured Query Language

SRI: Subresource Integrity

VDI: VirtualBox Disk Image

VM: Virtual Machine

XSS: Cross-Site Scripting

ZAP: Zed Attack Proxy

Executive Summary

This capstone project, “Web Application Assessment," focused on identifying and analyzing security weaknesses within a modern, deliberately vulnerable web application. Using a structured methodology, a virtual lab environment was established with OWASP Juice Shop as the target and a Kali Linux VM equipped with industry standard tools the OWASP ZAP, Burp Suite, and Nikto as the attack platform.

The assessment successfully uncovered a range of vulnerabilities, which were categorized according to the OWASP Top 10 (2021). Key findings included Security Misconfigurations (A05) such as missing HTTP security headers and publicly browsable backup files. The project also identified Cryptographic Failures (A02) and Broken Access Control (A01) through the exploitation of Insecure Direct Object References (IDORs). In a simulated authentication scenario, Identification and Authentication Failures (A07) were discovered, as the login API returned a specific error code.

The project not only provides a detailed analysis of these vulnerabilities but also outlines a comprehensive remediation guide with actionable steps and authoritative references from OWASP. The findings demonstrate the critical importance of proactive vulnerability assessment and secure coding practices to protect web applications from common and damaging cyber threats. The project's success proves our ability to apply theoretical knowledge to a practical, real-world scenario, a key skill for any cybersecurity professional.

As part of the remediation phase, secure coding practices were mapped to each confirmed vulnerability. For every finding, examples of insecure versus secure JavaScript/Node.js code were developed, along with explanations of the underlying root causes and references to OWASP best practice guidelines. This approach bridged the gap between vulnerability detection and practical prevention.

Problem Statement

Business Context

These days, web applications are a prime target for cyberattacks, and hackers are always on the lookout for security weaknesses. When they find vulnerabilities like SQL injection or weak access controls, the results can be pretty bad, we're talking about data breaches, lost money, and a company's reputation getting damaged. The real challenge is that a lot of organizations just don't have a good way to do thorough security checks. They often don't have a solid plan or the hands-on experience, which leaves them wide open to common flaws that are easy to exploit.

This project is designed to tackle that problem head-on. By running a practical, step-by-step vulnerability assessment on a well-known insecure web application, the project shows exactly how to find and report security risks as well as provide secure coding practices to prevent future explorations. Our main goal is to demonstrate how to use cybersecurity skills in a real-world setting, so we can proactively find these threats and figure out how to fix them, making a company's security much stronger.

Methodology

This project was conducted within a controlled cybersecurity testing environment utilizing two VMs alongside specialized vulnerability assessment tools. The target VM hosted the vulnerable web application, while the attacker VM contained the necessary testing tools and served as the penetration testing platform. This homelab configuration provided an isolated and secure environment, allowing for realistic attack-and-defense simulations without the risk of affecting external systems. The attacker VM was equipped with OWASP ZAP, Burp Suite, and Nikto, each chosen for their unique strengths in web application security testing. OWASP ZAP was primarily used for automated scanning to detect common vulnerabilities such as cross-site scripting (XSS) and security misconfigurations. Burp Suite offered both manual and semi-automated testing capabilities, enabling in-depth analysis through request and response manipulation. Nikto focused on identifying outdated components, misconfigurations, and known server-side issues. These tools were systematically run against the target web application to produce comprehensive vulnerability reports.

The raw outputs from these scans were then consolidated into a structured dataset containing relevant fields such as vulnerability name, cause, severity, OWASP category, CWE ID, detection method, impact score, remediation severity, and recommended remediation actions. This dataset was cleaned and standardized to ensure consistency and readiness for analysis. Based on the findings, the vulnerable code sections within the application were reviewed in detail, and specific insecure coding practices were identified. Secure code alternatives were then drafted as proposed solutions, incorporating industry best practices such as input validation, secure session handling, proper configuration of security headers, and the replacement or updating of outdated components. These secure code implementations were designed as theoretical

improvements and were not deployed or tested in the live environment, but rather presented as a reference for how the application's security posture could be enhanced.

Finally, the processed dataset was imported into Power BI to create an interactive vulnerability dashboard. The dashboard visualized the severity distribution of vulnerabilities using pie charts, displayed the frequency of issues by OWASP category through bar charts, illustrated the relationship between causes and impact scores using a heatmap, and highlighted the most critical vulnerabilities by impact in a ranked table. This methodology ensured a complete workflow from vulnerability identification and the development of proposed remediation strategies to visualization, offering a clear, data-driven assessment of the web application's potential for improved security.

Cybersecurity Components

- **Security Domains:** Application Security, Vulnerability Assessment, Penetration Testing

- **Tools & Technologies:** OWASP ZAP, Burp Suite, Nikto, OWASP Juice Shop (vulnerable web application), Virtual Machines (VMware/VirtualBox), Web Server Hosting Environment,VS Code, TypeScript, JavaScript (Node.js), SQL, HTML

- **Frameworks:** Helmet – Middleware for setting secure HTTP headers, bcrypt – Library for password hashing and secure authentication
- **Security Standards & References:** OWASP Top 10 (2021), OWASP Cheat Sheets

Data Analytics Components

- **Data Sources:** Vulnerability reports generated from OWASP ZAP, Burp Suite, and Nikto during penetration testing on OWASP Juice Shop; compiled into a structured CSV dataset containing vulnerability names, causes, severity ratings,

OWASP categories, CWE IDs, detection methods, impact scores, remediation severity levels, and recommended remediation actions.

- **Analytics Techniques:** Data cleaning and standardization, descriptive analytics, visual analytics (severity distribution, category frequency, cause-impact correlation), and risk prioritization based on impact and severity.

- **Tools & Technologies:** Power BI for dashboard creation and visualization, CSV processing, data transformation using Power Query.

Visualizations:

- **Pie Chart** showing the distribution of vulnerabilities by severity level

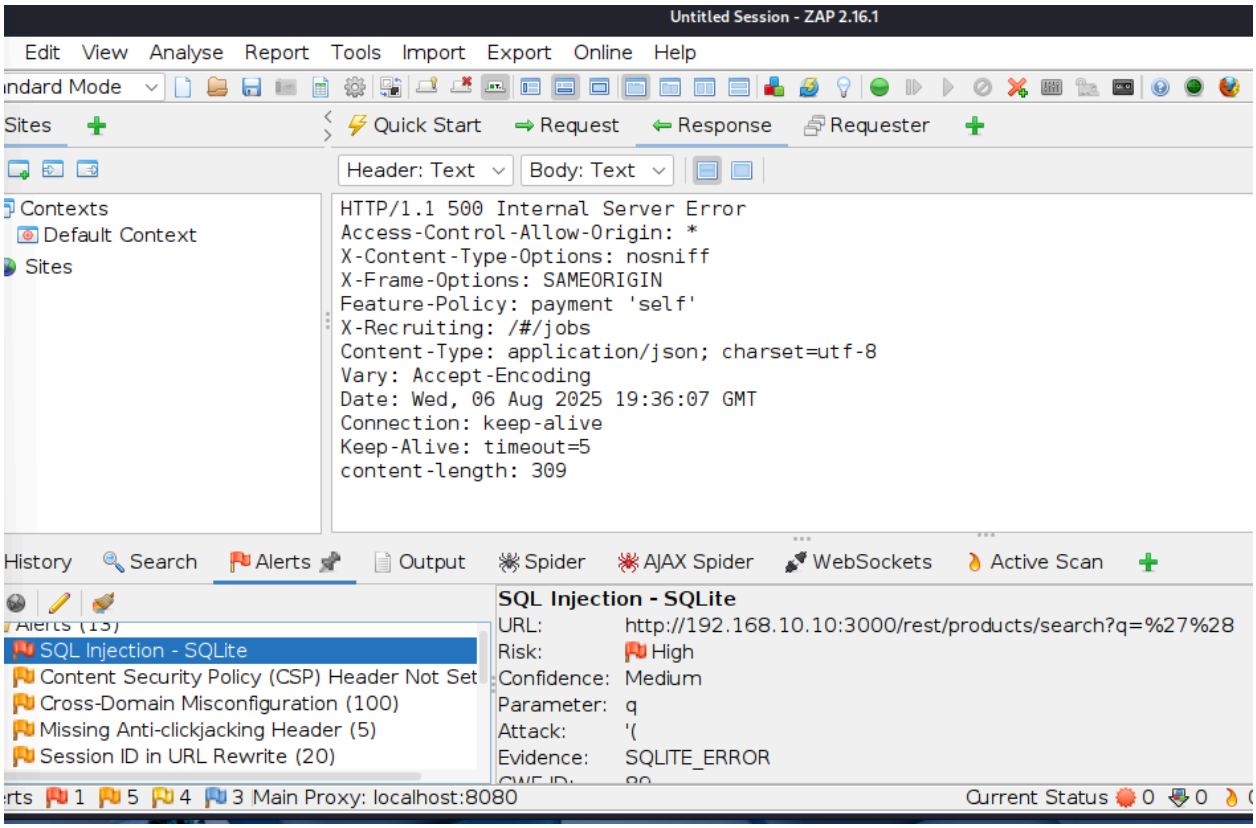
- **Bar Chart** displaying counts of vulnerabilities by OWASP category

- **Heatmap** mapping vulnerability causes to impact scores

- **Ranked Table** highlighting top vulnerabilities by impact score

Tools & Technologies: Power BI for dashboard creation and interactive visualization, CSV processing, and data transformation using Power Query.

Simulations



```
GET /rest/basket/6 HTTP/1.1
```


```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 523
ETag: W/"20b-dNhAfswkbS2pHWBqetQnymhaSuo"
Vary: Accept-Encoding
Date: Sun, 03 Aug 2025 16:40:02 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
    "status":"success",
    "data":{
        "id":6,
        "coupon":null,
        "UserId":23,
        "createdAt":"2025-08-03T16:31:42.342Z",
        "updatedAt":"2025-08-03T16:31:42.342Z",
        "Products":[
            {
                "id":1,
                "name":"Apple Juice (1000ml)",
                "description":"The all-time classic.",
                "price":1.99,
                "deluxePrice":0.99,
                "image":"apple_juice.jpg",
                "createdAt":"2025-08-03T15:28:55.274Z",
                "updatedAt":"2025-08-03T15:28:55.274Z",
                "deletedAt":null,
                "BasketItem":{
                    "ProductId":1,
                    "BasketId":6,
                    "id":9,
                    "quantity":1,
                    "createdAt":
                    "2025-08-03T16:33:34.735Z",
                    "updatedAt":
                    "2025-08-03T16:33:34.735Z"
                }
            }
        ]
    }
}
```


Figure 1: Broken Access Control

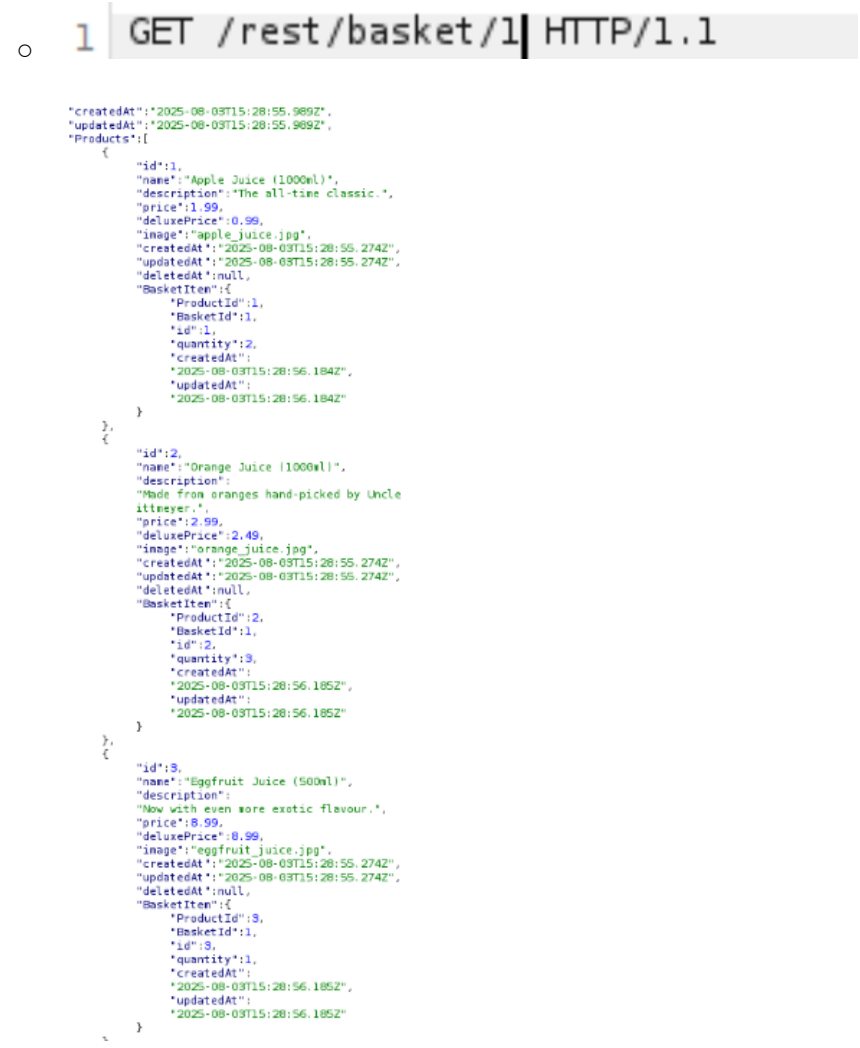
- 

Figure 2: Broken Access Control Manipulation

Broken Access Control involved the manipulation of the GET statement to return someone else's basket. This proves to be a vulnerability in that we can view someone else's account information or in this case, their shopping basket.

Figure 1 illustrates a normal log with the GET statement providing information about our basket of goods. **Figure 2** illustrates the manipulation of the GET changing the

number from a 6 to a 1 and the results of the alteration. We are now able to view the basket of someone else's shopping cart.

Results

Cause	Critical	High	Medium-High	**Total**
Vulnerable and Outdated Components			1	**1**
Software and Data Integrity Failures		1		**1**
Security Misconfiguration			3	**3**
Injection	1			**1**
Identification & Authentication Failures		1		**1**
Cryptographic Failures			1	**1**
Broken Access Control		1	5	**6**
Total	**1**	**3**	**10**	**14**

This section of the report outlines how the project was tested or validated to ensure its accuracy and effectiveness. It presents the results in a clear and organized manner, often supported by tables, figures, or graphs for better understanding. The findings are then interpreted to explain their significance, demonstrating how they address the original problem statement and highlighting the overall impact and value of the project's outcomes.

1. A01 – Broken Access Control

🔓 **Insecure Code (TypeScript):**

ts

```
import { Request, Response } from 'express';

app.get('/user/:id', (req: Request, res: Response) => {
  const userId: string = req.params.id;
  db.query('SELECT * FROM users WHERE id = ?', [userId], (err:
any, results: any[]) => {
```

```
    res.json(results);
  });
});
```

✅ **Secure Code (TypeScript):**

```
import { Request, Response } from 'express';

interface AuthUser {
  id: number;
  role?: string;
}

// Assume a preceding auth middleware populates req.user
app.get('/user/:id', (req: Request & { user: AuthUser }, res:
Response) => {
  const requestedId: number = parseInt(req.params.id, 10);
  const loggedInUserId: number = req.user.id;

  if (Number.isNaN(requestedId) || requestedId !==
loggedInUserId) {
    return res.status(403).json({ message: 'Access denied' });
  }

  db.query('SELECT * FROM users WHERE id = ?', [requestedId],
(err: any, results: any[]) => {
    if (err) return res.status(500).json({ message: 'Server
error' });
    res.json(results);
  });
});
```

Figure 3: Broken Access Control Remediation Strategy

Insecure Code Walkthrough:

- The route `/user/:id` takes a user ID from the URL.
- It runs a SQL query directly using that ID without checking if the current user is authorized.

- This means any logged-in or even unauthenticated user could access someone else's data by changing the URL ID.

Secure Code Walkthrough:

- The route still accepts `/user/:id`, but also retrieves the currently logged-in user's ID from `req.user`.
- It checks: if the requested ID is not the same as the logged-in user's ID, it returns a `403 Forbidden`.
- Only if the IDs match does it query the database for that user's data.
- This prevents Insecure Direct Object Reference by enforcing server-side authorization.

2. A03 – SQL Injection (Login Bypass)

🔓 **Insecure Code (TypeScript):**

```
import { Request, Response } from 'express';

app.post('/login', (req: Request, res: Response) => {
  const query = `SELECT * FROM users WHERE email =
'${req.body.email}' AND password = '${req.body.password}'`;
  db.query(query, (err: any, results: any[]) => {
    // ...
  });
});
```

✅ **Secure Code (TypeScript with parameterized queries & hashing):**

ts

```
import { Request, Response } from 'express';
import bcrypt from 'bcrypt';

app.post('/login', (req: Request, res: Response) => {
  const { email, password } = req.body as { email: string;
password: string };

  const query = 'SELECT id, email, hashed_password FROM users
WHERE email = ?';
  db.query(query, [email], async (err: any, results: any[]) => {
    if (err) return res.status(500).json({ message: 'Server
error' });
```

```
    const user = results?.[0];
    const ok = user && await bcrypt.compare(password,
user.hashed_password);
    if (!ok) {
      // Generic response to avoid user enumeration
      return res.status(200).json({ success: false, message:
'Invalid credentials' });
    }

    // Issue session/JWT here…
    return res.json({ success: true, userId: user.id });
  });
});
```

Figure 4: SQL Injection (Login Bypass) Remediation Strategy

Insecure Code Walkthrough:

- Takes `email` and `password` from the request body.
- Builds a SQL query by concatenating the user's input directly into the query string.
- If an attacker enters `' OR 1=1--`, the query will return all rows, effectively bypassing authentication.
- This is a classic SQL Injection vulnerability.

Secure Code Walkthrough:

- Uses a parameterized query: `SELECT * FROM users WHERE email = ?`.
- Passes the input values separately so the database treats them as data, not executable SQL.
- Compare the provided password to the stored hashed password using `bcrypt`.
- Returns a generic "Invalid credentials" message to prevent username enumeration.
- This fully blocks injection attempts because the input can't alter the SQL logic.

Post-Project Considerations

Looking ahead, this project has several potential possibilities for future enhancement and application. For one, the vulnerability assessment process could integrate more security mechanisms such as real time patches to vulnerabilities found. Another consideration would be expanding the project's scope to include authenticated scanning, which would provide deeper insights into vulnerabilities that are only accessible to logged-in users. Finally, the methodology developed in this project could serve as a foundation for assessing the security of other types of web applications and APIs, making it a valuable tool for future security audits.

Future Enhancements

Automated Scanning in a CI/CD Pipeline: Integrate tools like OWASP ZAP's API into a continuous integration/continuous deployment (CI/CD) pipeline. This would enable automated security scans on every code change, allowing for the early detection of vulnerabilities.

Authenticated Scanning: Expand the project's scope to perform authenticated scans. This involves configuring tools to log in with different user roles (e.g., admin, standard

user) to assess vulnerabilities in areas of the application that are only accessible after authentication.

Source Code Analysis (SAST): Supplement the dynamic analysis (DAST) by incorporating static analysis (SAST) tools. This would allow you to scan the application's source code to find a different class of vulnerabilities that dynamic scanners might miss.

Career Applications

Vulnerability Assessment Portfolio Piece: The entire project serves as a strong, tangible portfolio piece demonstrating your ability to perform a professional-grade vulnerability assessment from start to finish.

Resume Skills: The project highlights key resume skills such as **DAST**, **vulnerability analysis**, **risk assessment**, and **technical reporting**. It also shows proficiency with specific tools like **OWASP ZAP** and **Burp Suite**.

Networking Opportunities: The project can be a powerful talking point in interviews and for networking. It shows practical experience and a proactive approach to learning, which are highly valued in the cybersecurity field.

Knowledge Sharing

Create a Blog Post or Article: Write a detailed blog post or article summarizing your project's methodology, key findings, and remediation advice. This is an excellent way to showcase your expertise and build a professional online presence.

Presentation at a School Event: Prepare a short presentation for a school or local cybersecurity event. A live demo of your findings is a great way to share your knowledge and engage with other students and professionals.

Open-Source Contributions: Consider contributing to the documentation of the tools you used, such as OWASP ZAP or Burp Suite. This demonstrates a commitment to the open-source community and a deeper understanding of the tools.

Conclusion

This capstone project successfully demonstrated a comprehensive and practical approach to web application security, bridging the gap between theoretical knowledge and real-world application. By establishing a controlled testing environment and leveraging industry-standard tools like OWASP ZAP, Burp Suite, and Nikto, we were able to conduct a thorough vulnerability assessment of a deliberately insecure web application. The findings confirmed the presence of critical security weaknesses, including Broken Access Control, SQL Injection, and Security Misconfigurations, all of which were mapped directly to the OWASP Top 10 (2021).

Our work went beyond mere detection. For each identified vulnerability, we developed detailed remediation strategies, providing clear examples of insecure and secure code implementations. This focus on practical, actionable solutions such as using parameterized queries to prevent SQL Injection and implementing server-side access checks to block Insecure Direct Object References not only addressed the immediate threats but also reinforced the importance of secure coding practices.

The project's success validates our ability to apply a systematic methodology for vulnerability assessment and highlights the critical need for proactive security measures in modern web development. The data-driven approach, powered by visualizations in Power BI, provided a clear, compelling overview of the risks and their potential impact. Ultimately, this project serves as a strong testament to our technical skills and our commitment to building a more secure digital landscape.

References

· Jagatic, T. N., Johnson, N. A., Jakobsson, M., & Menczer, F. (2007). Social phishing. *Communications of the ACM, 50*(10), 94–100.

· Symantec. (2023). *Internet Security Threat Report*. Retrieved from https://www.symantec.com/

OWASP Foundation. (2025). *OWASP Top 10: 2021*. Retrieved from https://owasp.org/www-project-top-ten/

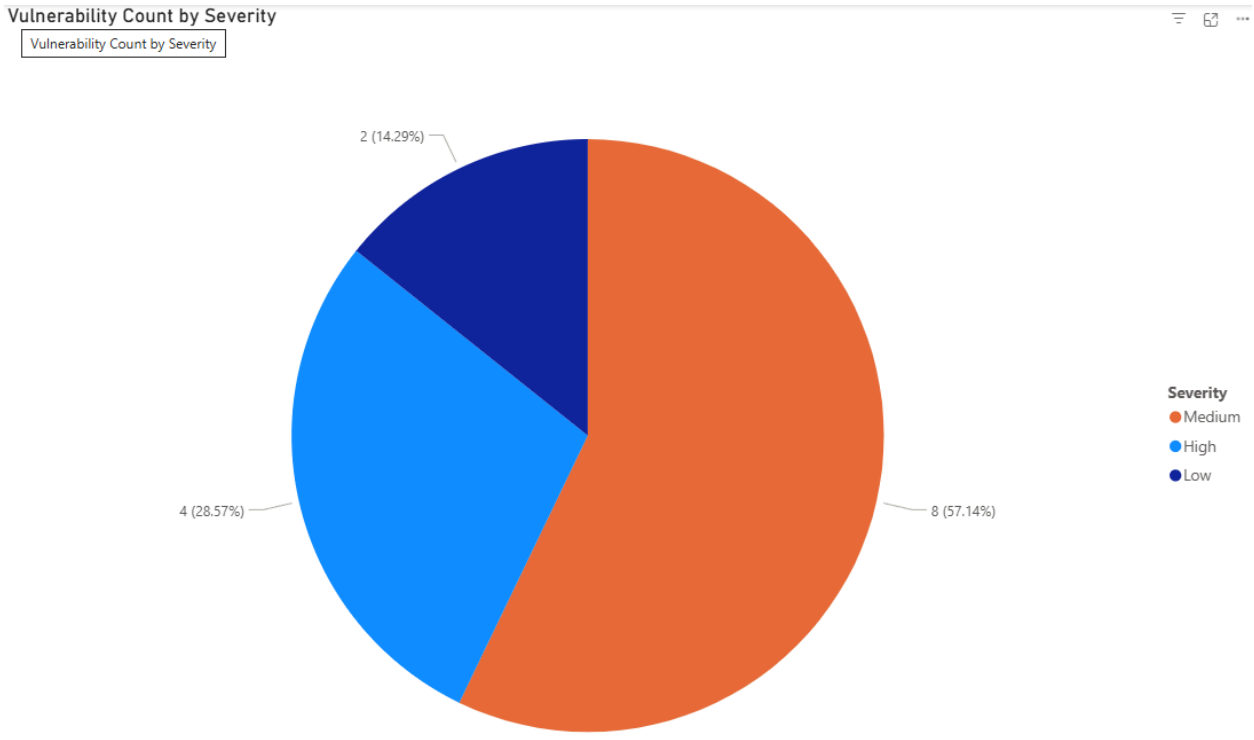
OWASP Foundation. (2025). *OWASP Zed Attack Proxy (ZAP)*. Retrieved from https://www.zaproxy.org/

PortSwigger Web Security. (2025). *Burp Suite Community Edition*. Retrieved from https://portswigger.net/burp/communitydownload

Tenable. (2025). *Nikto Web Server Scanner*. Retrieved from https://www.tenable.com/downloads/nikto

Appendix A

The appendix may include supplementary materials such as a full dataset description, additional charts, raw test results, or code snippets that support the main content of the report.


Vulnerability Count by Severity
Vulnerability Count by Severity
2 (14.29%)
4 (28.57%)
8 (57.14%)
Severity
Medium
High
Low