0.1 Authenticated Scans Using ZAP

Authenticated scanning broadens the test surface by including pages and features that are visible only after login. To perform an authenticated scan with ZAP use the `Manual Explore` option, provide the application URL, enable the Heads Up Display (HUD), and launch the browser through ZAP. The browser that ZAP launches may display a security warning due to ZAP's interception certificate. Proceed to the application and authenticate with an account that has relevant privileges, for example an administrative user. This allows ZAP to record traffic while exercising authenticated workflows.

For the `BookShelf` example, create a small file on the testing machine and upload it through the administrative interface to exercise file upload handling. Use this command to create a file:

touchzap.txt

Upload the file through the admin page and then use the application's verification feature. In this example the admin page's `Verify Results` action checks whether a file named `books.xml` exists in the admin directory and returns a confirmation in the browser.

After exercising authenticated features, log out of the application and close the browser window. In the ZAP interface, locate the target under the `Sites` tree, right click the application node, select the Attack menu, and choose `Active Scan`. Use the default policy and start the scan by clicking `Start Scan`. Authenticated scans often take longer than unauthenticated scans because there are more pages and parameters to evaluate.

Monitor the `Alerts` tab as the scan proceeds. Authenticated testing commonly reveals issues that are not visible to unauthenticated users, including problems on administration pages, file handling flaws, and logic errors.

Following is a sample image of the `Alerts` tab showing the vulnerabilities discovered during the authenticated scan:

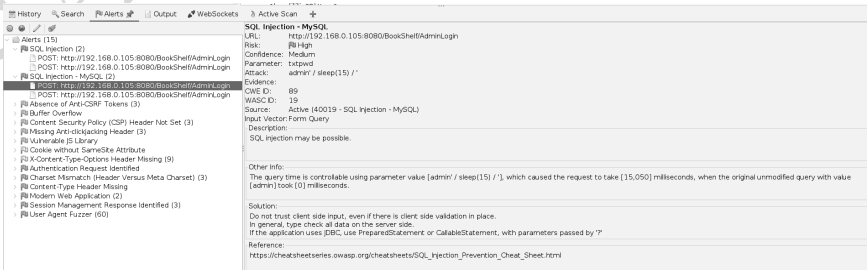

Figure 1: Authenticated Vulnerability Scan Results

In this authenticated example, ZAP identified a time based blind SQL injection affecting the `txtpwd` parameter on the login form. The scanner used a payload of the form:

```
admin' / sleep(15) / '
```

When the server's response time increased by more than the expected threshold, ZAP interpreted the delay as evidence that the injected sleep function executed in the database. Time based blind injection is confirmed when a clear and repeatable delay appears only when the payload is present. In this test the response returned `Invalid Username or Password`, which is a typical result from a login attempt. The true indicator of injection was the timing behaviour rather than a distinct error message.

This issue maps to CWE-89, improper neutralization of special elements used in an SQL command. The root cause is the inclusion of user input directly within SQL statements. To mitigate this problem developers should use parameterized queries or prepared statements and avoid dynamic SQL composition using untrusted input. Authenticated scanning therefore plays an important role because it can reveal critical flaws in privileged functionality that an unauthenticated scan would miss.

0.2 Can Automated Scanners Discover all Vulnerabilities?

Automated scanners are powerful for finding common vulnerabilities but they cannot provide a complete examination. These tools operate by applying rules, signatures, and predefined payloads. That makes them effective for known classes of issues, but they struggle with context dependent flaws and logic errors. Vulnerabilities such as insecure deserialization, complex authorization bypasses, multi step workflow problems, and race conditions often require human reasoning and bespoke testing.

For example, the `BookShelf` application contains a Java deserialization vulnerability in its `Contact Us` functionality that was not detected by ZAP. Detecting such an issue requires an understanding of the application workflow and carefully crafted tests that go beyond standard tool checks. Human testers can consider business logic, design nuanced inputs, and sequence actions in ways that automated scanners cannot.

It is also critical to perform scanning responsibly. Running automated scans without explicit permission can harm systems. Scanners generate traffic that may overwhelm servers, trigger monitoring alerts, or even cause crashes. Testing should be conducted in controlled environments such as staging servers and must always be authorized by the application owner. Purchasing a tool license does not grant the right to scan any system at will.

In practice, automated scanning should be treated as one component of a broader testing strategy. Use automated tools to quickly identify likely issues and to maintain continuous checks, and complement those results with careful manual testing to discover vulnerabilities that require human ingenuity. This combination yields the most accurate and actionable view of an application's security posture.

1. Web Security in the age of Artificial Intelligence (AI)

1.1 Introduction

Over the last two decades, the web has evolved from a collection of static documents to a complex ecosystem of highly interactive applications. This journey has been shaped by shifts in architecture, from server-rendered pages to client-heavy single-page applications, from relational databases to polyglot persistence, where applications combine relational, document, graph, and even vector stores depending on the use case, and from monolithic systems to microservices and containerized deployments. Each of these transitions has unlocked new possibilities for user experience and scalability, but each has also introduced new risks and vulnerabilities.

We now stand at the beginning of another transformation: the integration of artificial intelligence, specifically Large Language Models (LLMs) into everyday web applications. What makes LLMs different from earlier software modules is that they are not deterministic. Traditional software components operate under strict logical rules, executing predictable instructions. LLMs, by contrast, generate probabilistic outputs based on statistical patterns learned from massive datasets. This unpredictability is the source of their power, but also the root of their security challenges.

With just a few lines of code, a developer can give an application the ability to understand natural language, to synthesize text, to translate between formal and informal representations, and even to generate executable code. This capability has enabled novel features such as conversational search, intelligent assistants, and workflow automation. Yet the very flexibility that delights users also empowers attackers. When a system willingly accepts instructions in natural language, it becomes vulnerable to manipulation in ways that traditional software never was.

In this chapter, we explore the intersection of AI integration and web application security. We examine how LLMs fit into application architecture, the kinds of vulnerabilities they introduce, and how existing frameworks like OWASP's LLM Top 10 and MITRE ATLAS can help us reason about them. To ground the discussion, we use a deliberately vulnerable demonstration system, `BookShelf AI`, as our case study. Through its benign features and intentional flaws, `BookShelf AI` illustrates how AI-driven risks converge with classical web vulnerabilities like SQL injection. This chapter is not merely about cataloging problems. It is also about showing how these vulnerabilities can be exploited in practice. By the end, the reader should have a deep appreciation for both the opportunities and the dangers of embedding LLMs into web applications, and a clear understanding of why vigilance is essential as AI becomes a permanent layer in the web stack.

1.2 How LLMs work in the modern web stack

To understand where vulnerabilities arise, we must first understand how LLMs are typically integrated into web applications. In most cases, the model is not the front-facing component. Instead, it is embedded into the application's architecture as a service. A typical stack might look like this:

- **A front-end interface**, often built in React or Vue, where users enter natural language queries.
- **A backend API**, usually implemented in a web framework such as Flask, Express, or Django, which orchestrates requests.
- **A model service**, either a hosted API (e.g., OpenAI, Anthropic) or a local container running an open-source model.
- **A database or other knowledge store**, which may be accessed through SQL or via retrieval-augmented generation (RAG).

The promise of this architecture is flexibility. The front end can remain simple, the backend can scale horizontally, and the model can act as a universal translator between natural language and structured operations.

In `BookShelf AI`, this design takes a concrete form: a React-based user interface communicates with a Flask backend. The backend calls a locally containerized LLM to interpret user queries, generating SQL commands that are executed against a SQLite database containing book and user records.

The appeal is obvious. Users can simply type `Show me all science fiction books after 1980` instead of constructing complex filters. The LLM parses intent, generates a query, and the results are returned seamlessly. But this architecture is also fragile. Every layer in this chain: input, model, output, and execution can be manipulated.

1.3 The Risks of AI-Powered Applications

Traditional web applications are already vulnerable to a variety of attacks: SQL injection, cross-site scripting, authentication bypass, and so on. Security over the past two decades has largely been about building defenses against these known patterns. Developers have learned (sometimes painfully) that unvalidated inputs, unsanitized outputs, and poor session management can be catastrophic.

LLM-powered applications inherit all of these risks and add new ones. What makes the new risks particularly challenging is that they blur boundaries. In a traditional system, data and instructions are separate. A text field is supposed to contain only data, while code is meant to be executed separately. In an LLM-driven system, that distinction collapses. A user's natural language input is simultaneously data and instruction.