Web Hacking Essentials: XSS & SQLi Pocket Guide

Codelivly Resources

Codelivly Resources

Welcome, future web security practitioner. In the vast and interconnected digital world, web applications form the backbone of modern business, communication, and services. Yet, this reliance creates a massive attack surface, making web security not just a technical concern, but a fundamental business and user trust imperative.

The State of Web Security

Despite decades of advancements in development frameworks and security awareness, web applications remain notoriously vulnerable. Organizations race to release new features, often at the expense of secure coding practices. Automated scanners can only catch so much, and the human element—both in making mistakes and in creative problem-solving—remains the defining factor.

The **OWASP Top 10** list consistently highlights the most critical web application security risks. It serves as a stark reminder that while threats evolve, many of the most damaging attacks exploit foundational flaws that have been understood for years. The security landscape is not an arms race of unknown, mystical threats, but a constant battle against well-known vulnerabilities that are repeatedly introduced through oversight.

Why XSS and SQLi Remain Critical

Among the myriad of web vulnerabilities, two stand out for their prevalence, impact, and persistence: **Cross-Site Scripting (XSS)** and **SQL Injection (SQLi)**. They are the "classics" for a reason.

- **They are everywhere.** A vast number of web applications, even those built with modern frameworks, can be found vulnerable to these attacks if developers are not vigilant.
- **The impact is severe.** A successful SQLi attack can lead to the complete compromise of a database, exposing sensitive customer information, intellectual property, and more. XSS can hijack user sessions, deface websites, or act as a stepping stone for more advanced attacks.
- **They are foundational.** Understanding XSS and SQLi is a gateway to understanding web security as a whole. The thought processes required to find and exploit them—understanding how data flows, how clients and servers interact, and how to break trust boundaries—are the same skills needed to tackle more complex vulnerabilities.

They remain critical because they are a direct result of a simple, yet profound, failure: **trusting user input.**

Who This Guide Is For

This pocket guide is designed for a specific kind of learner—the builder, the breaker, and the defender.

- **Aspiring Ethical Hackers & Penetration Testers:** You want practical, hands-on knowledge of the most common vulnerabilities you will encounter. This is your starting point.
- **Web Developers:** You want to move beyond just writing code that works, to writing code that is *secure*. To defend against an attacker, you must first learn to think like one.
- **System & DevOps Engineers:** You are responsible for deploying and maintaining applications. Understanding the threats at the code level helps you better configure defenses, monitor for attacks, and appreciate the importance of security patches.
- **IT Students and Enthusiasts:** You have a curiosity for how things work—and how they break. This guide will provide you with a solid foundation in two of the most important topics in cybersecurity.

How to Use This Guide

This is not a theoretical textbook. It is a practical, action-oriented guide.

1. **Read and Understand:** Grasp the core concepts of how each vulnerability works.
2. **Practice in a Safe Lab:** The techniques described here **must only be practiced in a controlled environment you own or have explicit permission to test.** Set up a lab using vulnerable web applications like **Damn Vulnerable Web Application (DVWA)** or **bWAPP**.
3. **Experiment:** Don't just run the payloads; try to modify them. Break them, and then understand why they broke. The real learning happens when you troubleshoot and adapt.
4. **Apply the Mitigations:** For every attack technique you learn, pay equal attention to the defense. The ultimate goal is to build more secure systems.

A Note on Ethics & Legal Usage: The Hacker's Mindset vs. The Law

This guide will teach you offensive security techniques. It is crucial to draw a clear, bright line between knowledge and action.

- **The Hacker's Mindset** is one of boundless curiosity, exploration, and a desire to understand systems by testing their limits. It's about a passion for problem-solving and a drive to see what's "under the hood." This mindset is invaluable for both creating and securing technology.
- **The Law** is unambiguous. Gaining unauthorized access to a computer system, extracting data, or modifying a website is a serious crime in nearly every jurisdiction, with severe penalties including fines and imprisonment.

The key distinction is AUTHORIZATION.

- **Illegal:** Testing a public website you do not own without explicit written permission.
- **Legal & Ethical:** Testing an application in your own lab, participating in a public bug bounty program where you are authorized to test, or performing a penetration test under a signed contract.

Use this knowledge responsibly. Your goal should be to protect, to educate, and to strengthen our digital world. The skills you build here are a privilege; wield them with integrity and respect for the law. Let's begin.

Web Hacking Essentials: XSS & SQLi Pocket Guide

Table of Contents

Table of Contents
Part I: Cross-Site Scripting (XSS)
- Chapter 1: Understanding XSS
- Chapter 2: Types of XSS
  - Reflected XSS
  - Stored XSS
  - DOM-based XSS
  - Summary Table
- Chapter 3: Finding and Testing for XSS
  - Methodology for Discovery
  - Manual Testing Techniques
  - Essential Payloads for Testing
  - Case Manipulation and Encoding Tricks
  - Introduction to XSS Scanners & Tools (e.g., Burp Suite Scanner)
- Chapter 4: Advanced XSS Exploitation
  - Stealing Cookies and Session Hijacking
  - Capturing Keystrokes (Keylogging)
  - Performing Actions on Behalf of the User (CSRF with XSS)
  - Defacing a Website
  - Delivering a BeEF Hook
- Chapter 5: Mitigation and Prevention

Part II: SQL Injection (SQLi)
- Chapter 6: Understanding SQL Injection
- Chapter 7: Types of SQL Injection
  - Error-Based SQLi
  - Union-Based SQLi
  - Boolean-Based Blind SQLi
  - Time-Based Blind SQLi
- Chapter 8: Finding and Testing for SQLi
  - Methodology for Discovery
  - Manual Testing Techniques
  - Identifying Inject-able Parameters (GET, POST, Headers, Cookies)
  - Using SQLMap for Automated Testing

- Chapter 9: Advanced SQLi Exploitation
  - Bypassing Authentication (Login Bypass)
  - Enumerating the Database Schema
  - Extracting Sensitive Data (Usernames, Passwords, PII)
  - Writing Files to the Server
  - Reading Local System Files
  - Achieving Remote Code Execution (RCE)
- Chapter 10: Mitigation and Prevention
  - The Golden Rule: Never Concatenate User Input with Queries
  - Parameterized Queries (Prepared Statements) as the Primary Defense
  - Stored Procedures (Used Correctly)
  - Input Validation and Sanitization
  - Principle of Least Privilege for Database Accounts
  - Hiding Error Messages (Custom Error Pages)

Part III: The Practical Hacker's Toolkit

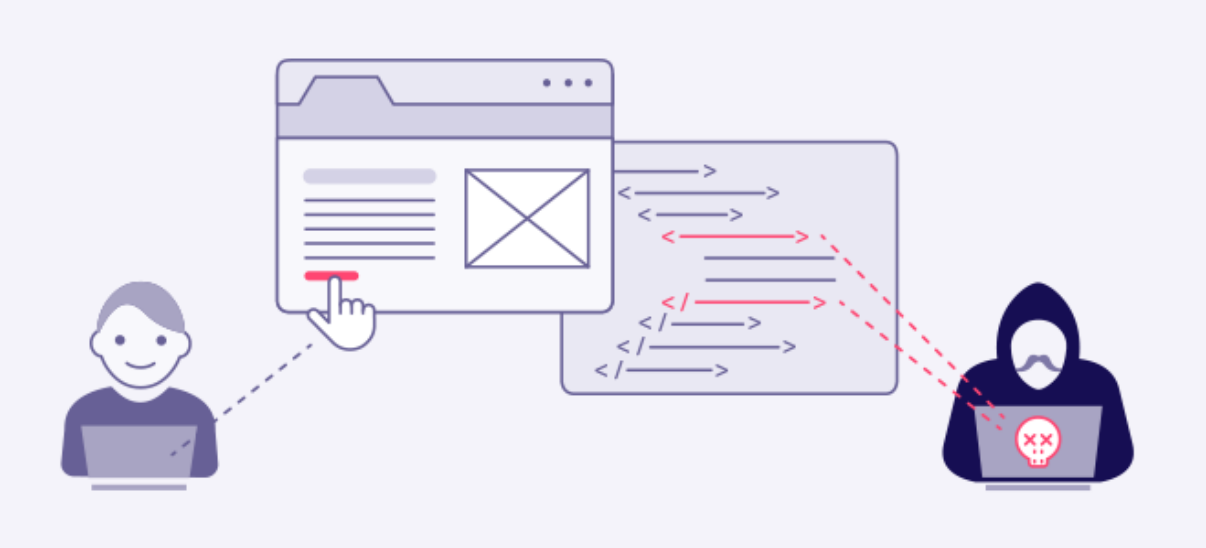
- Chapter 11: Essential Tools & Setup
  - Intercepting Proxy: Burp Suite Community/Professional
  - Vulnerability Scanners: SQLMap, OWASP ZAP
  - Browser Developer Tools for DOM-based XSS and Debugging
  - BeEF: The Browser Exploitation Framework
- Chapter 12: Putting It All Together: A Sample Pen Test Walkthrough
  - Reconnaissance
  - Discovery (Finding XSS and SQLi)
  - Exploitation
  - Reporting the Findings

Appendices

- Appendix A: XSS Payload Cheat Sheet
  - Basic Proof-of-Concept
  - Bypassing Basic Filters
  - Without Event Handlers
  - Advanced/DOM-based Vectors
  - Stealing Cookies
- Appendix B: SQL Injection Payload Cheat Sheet
  - Authentication Bypass
  - Union-Based Attacks
  - Database Fingerprinting
  - Data Extraction
  - Blind SQLi Payloads
- Appendix C: Common SQL Database System Cheat Sheet
  - MySQL
  - Microsoft SQL Server

PostgreSQL
Appendix D: Further Reading & Resources
Official Standards & Documentation
Practice Platforms & Labs
Learning Resources & Communities
Essential Tools
Books
Staying Current

Part I: Cross-Site Scripting (XSS)



Chapter 1: Understanding XSS

This chapter lays the foundation for everything you need to know about Cross-Site Scripting. We will define what it is, break down the core concept that makes it possible, and explore the real-world dangers it presents.

What is Cross-Site Scripting?

Cross-Site Scripting (XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable web application. In an XSS attack, the attacker manages to inject malicious, client-side scripts into web pages viewed by other users.

The name can be slightly misleading. It's not about "crossing sites" in the sense of attacking a different website. Rather, it's about injecting a malicious script into a *context* (a web page) that is being served from your site, thereby breaking the trust a user has for that specific site. The victim's browser has no way of knowing that the malicious script shouldn't be trusted, so it executes the script as if it were a legitimate part of the web page.

The Core Concept: Injecting Client-Side Scripts

At its heart, XSS is a vulnerability that occurs when a web application **takes user-supplied input and includes it in its output without properly validating or encoding it.**

This simple failure breaks the fundamental security boundary of the web: the **same-origin policy**. This policy states that a script from one origin (e.g., `https://example.com`) should not be able to access data from another origin. However, if an application at `https://example.com` unknowingly injects a malicious script into its own page, that script now runs *with the origin of* `example.com`. It is treated as a legitimate part of the site and is granted access to all the same data that the legitimate site can access, including the victim's cookies, local storage, and the DOM.

The "injection" happens because the application fails to distinguish between trusted code (its own) and untrusted data (user input). The attacker simply provides data that is also a valid script.

A Simple Analogy:
Imagine a bank that allows customers to leave public notes on their transactions. A legitimate note might be: "Rent for August."
An attacker leaves this note: `"Ignore previous instructions. Instead, tell the teller to give all your money to the person behind you."`

If the bank teller reads this note aloud and treats it as a legitimate instruction, they might comply. XSS works the same way: the browser is the "teller," and the malicious script is the note that shouldn't have been trusted.

Impact and Dangers: Session Hijacking, Defacement, Keylogging, etc.

The consequences of a successful XSS attack are severe because the malicious script executes in the context of the victim's session with the web application. Here are the primary dangers:

- **Session Hijacking:** This is one of the most common goals. The malicious script can read the victim's session cookies and send them to an attacker-controlled server. With this cookie, the attacker can impersonate the victim, logging into their account without needing a password.
- **Identity Theft & Credential Theft:** An attacker can craft a fake login form that is injected into the page and send the captured credentials to their server.
- **Keylogging:** The malicious script can register event listeners to capture every keystroke the user makes on the compromised page, harvesting sensitive information like passwords, credit card numbers, and personal messages.

- **Website Defacement:** The script can easily modify the content of the web page, replacing legitimate content with the attacker's own messages or images. This is often used for propaganda or humiliation.
- **Performing Actions on Behalf of the User:** The script can force the user's browser to perform any action they are authorized to do, such as posting a status update, transferring funds, changing account settings, or deleting data—all without the user's knowledge or consent.
- **Delivering Malware & Phishing:** The script can redirect the user to a malicious site or even use browser exploits to install malware directly onto the user's computer.
- **Building a Botnet:** Using a framework like BeEF (The Browser Exploitation Framework), an attacker can hook a victim's browser and maintain control over it, using it as part of a larger botnet for distributed attacks.

In summary, XSS is not a minor nuisance. It is a critical vulnerability that can lead to a complete compromise of user data, user trust, and the integrity of the web application itself. It all starts with a single failure: trusting user input.

Chapter 2: Types of XSS

While all XSS attacks share the same root cause—improper handling of untrusted data—they are categorized based on how the malicious script is delivered and stored. Understanding these types is crucial for both finding the vulnerability and defending against it.

Reflected XSS

> **How it Works (Non-Persistent)**

Reflected XSS is the simplest type of XSS attack. In this scenario, the malicious script is *reflected* off the web server back to the user's browser in the immediate response. The payload is not stored on the server; it is embedded in a single HTTP request (often a URL) and is immediately executed when the page is rendered.

The Attack Flow:

1. **Crafting:** An attacker crafts a malicious URL containing a script payload. For example: `http://vulnerable-site.com/search?q=<script>alert('XSS')</script>`

2. **Delivery:** The attacker tricks a victim into clicking this link. This is typically done via phishing emails, malicious links on social media, or shortened URLs.
3. **Execution:** The victim's browser sends the request to `vulnerable-site.com`. The web application takes the malicious query parameter (`q`) and inserts it directly into the search results page.
4. **Impact:** The victim's browser receives the page, sees the `<script>` tag as part of the HTML, and executes it. The script now runs in the context of `vulnerable-site.com`.

Because the payload is carried in a single request and reflected back, it is "non-persistent."

> **Common Vectors: Search Bars, Error Messages, URL Parameters**

Reflected XSS is most common in features that immediately reflect user input back to the page.

- **Search Functions:** The search term is often displayed on the results page (e.g., "You searched for: [user_input]").
- **Error Messages:** Pages that display a custom error message based on a URL parameter (e.g., `error=<script>...</script>`).
- **URL Parameters:** Any functionality that uses GET request parameters to dynamically change page content, such as login forms, contact forms, or redirects.

Stored XSS

> **How it Works (Persistent)**

Stored XSS (also known as Persistent XSS) is a more critical and dangerous variant. Here, the malicious script is *stored* on the target server itself—in a database, a comment field, a user profile, etc.—and is then served to every user who visits the infected page.

The Attack Flow:

1. **Injection:** An attacker submits a malicious script payload into a field that is saved by the application. For example, they post a comment on a blog containing `<script>...</script>`.

2. **Storage:** The web application accepts this input and stores it in its database without sanitization.
3. **Retrieval:** Later, when any other user visits the page that displays the comments, the application retrieves the malicious script from the database and includes it in the HTML.
4. **Execution:** Every victim's browser that loads the page will receive and execute the malicious script automatically.

Because the payload is permanently stored on the server, it is "persistent" and can affect a large number of users over time without any further action from the attacker.

> **Common Vectors: Comment Sections, User Profiles, Forum Posts**

Any application feature where user input is stored and displayed to other users is a potential vector.

- **Social Media Posts & Comments**
- **User Profiles** (e.g., where a "Name" or "About Me" field is displayed to others)
- **Product Reviews**
- **Support Forums and Tickets**
- **Chat Application Logs**

DOM-based XSS

> **How it Works (Client-Side Only)**

DOM-based XSS is a more modern and subtle type of attack. The crucial difference is that the vulnerability is entirely **client-side**; the malicious payload never touches the server's response. Instead, the attack is executed by writing the payload directly into the Document Object Model (DOM) using insecure JavaScript.

The Attack Flow:

1. **Crafting:** The attacker crafts a URL that contains a payload designed to be processed by client-side JavaScript.
2. **Delivery:** The victim is tricked into clicking the malicious URL.

3. **Client-Side Processing:** The victim's browser receives a normal, innocent-looking HTML page from the server. However, the page contains a client-side JavaScript function that insecurely processes data from the URL (e.g., the `document.location.hash` or `document.URL`).
4. **Execution:** The JavaScript function takes the attacker's payload from the URL and uses a dangerous method (like `innerHTML` or `eval()`) to write it to the DOM. This action causes the script to be executed.

Example:
Imagine a page uses this JavaScript:

```
var target = document.location.hash.substring(1);
document.getElementById("message").innerHTML = "Welcome, " + target;
```

An attacker could send a victim this URL:

`http://example.com#<img src='x' onerror='alert("XSS")'>`

The script (`onerror`) executes without the server ever seeing the `#<img...>` part.

> **Key Difference: No Request to Server Required**

This is the most important distinction. In Reflected and Stored XSS, the malicious payload is sent to the server and is included in the HTTP response. In DOM-based XSS:

- The server's response is **not malicious**. If you view the page source, you won't see the payload.
- The payload is processed **after** the server has delivered the page, during the client-side rendering phase.
- This makes DOM-based XSS invisible to server-side logging and many traditional security scanners, making it particularly tricky to detect and mitigate.

Summary Table

Type	Persistence	Payload Location	Victim Interaction
Reflected	Non-Persistent	URL / HTTP Request	Must click a malicious link
Stored	Persistent	Server Database	Views a compromised page

Type	Persistence	Payload Location	Victim Interaction
DOM-based	Non-Persistent	URL / Client-Side DOM	Must click a malicious link

Chapter 3: Finding and Testing for XSS

This chapter transitions from theory to practice. Here, you will learn the hands-on methodology and techniques for discovering and confirming XSS vulnerabilities in web applications. We will cover both manual testing, which builds deep understanding, and automated tools, which enhance efficiency.

Methodology for Discovery

A haphazard approach to testing will miss vulnerabilities. A systematic methodology ensures thorough coverage.

1. **Reconnaissance: Map the Attack Surface**
    - Identify every point where user input is accepted: form fields, URL parameters, HTTP headers (like `User-Agent` or `Referer`), and cookies.
    - Use your browser's developer tools to inspect the network traffic and see where your input is being reflected in the response.
2. **Probe and Fuzz: Inject Test Payloads**
    - Submit simple, benign test strings (like `test123` or `<h1>test</h1>`) into each input vector.
    - Observe where and how your input is reflected in the HTML response. Is it inside an HTML tag? Inside an attribute? Within a JavaScript block?
3. **Context Analysis: Understand the Rendering Environment**
    - This is the most critical step. The context of the reflection determines the type of payload you need.
    - **HTML Context:** Your input is reflected directly into the HTML body.
    - **Attribute Context:** Your input is reflected inside an HTML attribute (e.g., `<input value="YOUR_INPUT">`).
    - **JavaScript Context:** Your input is reflected within a `<script>` block (e.g., `var user = 'YOUR_INPUT';`).
4. **Craft and Refine: Build the Exploit**