Great pick 👍 — **Server-Side Template Injection (SSTI)** is one of the most powerful injection flaws because it often leads directly to **Remote Code Execution (RCE)**. Let’s go step by step, in the same detailed style as before.

**🧩 Testing for Server-Side Template Injection (SSTI)**

**🔹 1. What is SSTI?**

* Many modern web frameworks use **templating engines** (e.g., Jinja2 for Python/Flask, Twig for PHP, Freemarker for Java, Smarty for PHP).
* Templates render dynamic HTML by embedding variables like {{username}}.
* If **user input is directly placed into the template**, an attacker can inject **malicious template expressions**.
* This can escalate to **RCE** depending on the template engine.

**🔹 2. Example of Vulnerable Code**

**Python (Flask + Jinja2):**

@app.route("/hello")

def hello():

name = request.args.get("name")

return render\_template\_string("Hello " + name)

* If attacker sends:

http://site.com/hello?name={{7\*7}}

* Output:

Hello 49

→ This confirms SSTI.

**🔹 3. Exploitation Flow**

1. **Detection Payloads** (testing for template engine behavior):
   * {{7\*7}} → Expect 49 (Jinja2, Twig).
   * ${7\*7} → Expression Language (EL) injection in Java.
   * <%= 7\*7 %> → ERB (Ruby).
2. **Fingerprint the Template Engine**:
   * Jinja2 (Python/Flask/Django): {{config}}
   * Twig (Symfony/PHP): {{dump(app)}}
   * Freemarker (Java): ${"freemarker"?api}
3. **Escalate to RCE** (if sandbox escapes work):
   * Jinja2 RCE Example:
   * {{ self.\_\_init\_\_.\_\_globals\_\_.\_\_builtins\_\_.open('/etc/passwd').read() }}

**🔹 4. Manual Testing Steps**

* Identify **dynamic reflections** in server output.
* Insert arithmetic payloads ({{2\*3}}) and see if evaluated.
* Try object access ({{config.items()}}) for leaks.
* If response changes based on evaluation → SSTI confirmed.

**🔹 5. Testing with Tools**

**🔸 Burp Suite**

* Intercept the request with a suspicious parameter.
* Replace input with SSTI payloads ({{7\*7}}, ${7\*7} etc.).
* Observe output.
* Use **Burp Intruder** to fuzz with template payload wordlists (SecLists has one).

**🔸 OWASP ZAP**

* Active Scan → Injection rules may detect template behavior.

**🔸 tplmap (specialized SSTI tool)**

* Automates SSTI detection & exploitation.
* Usage:

python tplmap.py -u "http://target/hello?name="

**🔹 6. Impact of Successful SSTI**

* **Information Disclosure** (view config, environment variables).
* **Privilege Escalation** inside app.
* **Remote Code Execution** on the server.
* **Pivoting** → from web app compromise to full infrastructure compromise.

**🔹 7. Mitigation Strategies**

✅ **Best Practices**:

1. Never concatenate user input directly into templates.
2. Use **template sandboxing** (many engines support restricted execution).
3. Escape user input:
   * Jinja2: {{ user\_input | e }}
   * Twig: Auto-escaping enabled by default.
4. Apply **allowlist for variables** (only pass specific safe variables to the template).
5. Regularly update template engines (sandbox escapes are discovered frequently).

**🔹 8. Quick Testing Payloads Reference**

| **Engine** | **Test Payload** | **Expected Output** |
| --- | --- | --- |
| Jinja2 (Python) | {{7\*7}} | 49 |
| Twig (PHP) | {{7\*7}} | 49 |
| ERB (Ruby) | <%= 7\*7 %> | 49 |
| Freemarker (Java) | ${7\*7} | 49 |
| Velocity (Java) | #set($x=7\*7) $x | 49 |

✅ That’s the **complete SSTI testing methodology** in WSTG style: Overview → Examples → Manual → Tools → Impact → Mitigation.

Do you want me to now also write you a **one-pager cheat sheet of SSTI payloads per template engine** (Jinja2, Twig, Freemarker, ERB, etc.) so you can use it directly during pentests?