Alright 👍 let’s do **WSTG notes for “Testing for JavaScript Execution” (WSTG-CLNT-02)**.

**📝 Notes: Testing for JavaScript Execution**

**🎯 Purpose**

To check if an application improperly allows **user-supplied input** to be interpreted as **JavaScript code**.  
This often leads to **XSS** or other client-side attacks, since injected JavaScript can be executed in the victim’s browser.

**⚡ Common Attack Scenarios**

1. **Inline Event Handlers**
2. <img src=x onerror=alert(1)>
3. <button onclick="alert(1)">Click</button>
4. **Script Injection**
5. <script>alert(1)</script>
6. **Malformed HTML Contexts**
7. <svg/onload=alert(1)>
8. <iframe src=javascript:alert(1)>
9. **JSON / JS Context Injection**
   * Application outputs user input directly inside a JavaScript block.
10. var username = "<payload>";
11. **DOM-Based JS Execution**
    * Input flows into dangerous sinks:
      + eval(), Function(), setTimeout(), document.write(), innerHTML.

**🔍 How to Test**

1. **Inject Payloads into Inputs**
   * Try <script>alert(1)</script>.
   * If blocked, try **event handler attributes**: <img src=x onerror=alert(1)>.
   * If still blocked, test **tagless injections** in JS contexts: ";alert(1);//.
2. **Check Different Contexts**
   * HTML body, attributes, JavaScript blocks, CSS, JSON.
   * Use browser DevTools to inspect DOM after injection.
3. **Fuzz with Polyglot Payloads**
   * Examples:
   * <svg/onload=alert(1)>
   * "><script>alert(1)</script>
   * javascript:alert(1)
4. **Use Tools**
   * **Burp Suite (Active Scanner)** for XSS injection attempts.
   * **OWASP ZAP** → XSS plugin.
   * Manual testing is still crucial.

**🛡️ Mitigation**

* Apply **contextual output encoding** (HTML, attribute, JS, CSS).
* Use **security libraries** for sanitization (DOMPurify).
* Avoid dangerous functions (eval, innerHTML, document.write).
* Enforce **Content Security Policy (CSP)**.
* Validate all inputs before rendering them.

✅ **Key Takeaway:**  
Testing for JavaScript Execution ensures that user input **cannot be executed as code** in the browser. It’s basically checking for **XSS vulnerabilities** in multiple contexts (HTML, attributes, JS, DOM).

👉 Do you want me to create a **quick “context-based injection cheat sheet”** (HTML context payloads, attribute payloads, JS block payloads, etc.) so you can test faster without memorizing all variations?