Got it 👍 Here are **precise WSTG notes for “Testing for Client-side Resource Manipulation” (WSTG-CLNT-06)**.

**📝 Testing for Client-side Resource Manipulation**

**🎯 Purpose**

To verify if an attacker can **modify or replace client-side resources** (JavaScript, CSS, images, config files) to:

* Inject malicious code (e.g., JS → XSS, CSS → data leaks).
* Alter application behavior (bypass logic, disable security checks).
* Exploit **insecure client-side trust** in static resources.

**⚡ Common Attack Scenarios**

1. **Modifying Local Resource References**
   * App loads scripts like:
   * <script src="/js/app.js"></script>

If attacker can tamper with /js/app.js (via LFI, upload, or cache poisoning), they gain full control.

1. **Parameter-based Resource Loading**
   * Example:
   * https://victim.com/load?file=app.js

Can be changed to:

https://victim.com/load?file=evil.js

1. **Third-Party Resource Abuse (CDN/External)**
   * If app uses scripts from http://cdn.example.com/lib.js without integrity checks → attacker intercepts (MITM, DNS poisoning).
2. **Manipulating JSON/Config Resources**
   * Changing .json / .xml files loaded by JS to modify app behavior or access rights.

**🔍 How to Test**

1. **Identify Resources**
   * Check browser DevTools → **Network tab** for all loaded resources (JS, CSS, fonts, JSON).
2. **Manipulate Resource References**
   * Intercept requests with **Burp/ZAP** → modify file, path, or url params.
   * Try replacing with attacker-controlled files.
3. **Check for Caching Weaknesses**
   * Test **HTTP cache poisoning** by injecting malicious payloads in cacheable responses.
4. **Evaluate Integrity Protections**
   * See if resources have **SRI (Subresource Integrity)** hashes.
   * If not → app trusts resources blindly.
5. **Test Dependency Tampering**
   * If open-source libraries (jQuery, React, etc.) are loaded via insecure channels → try replacing with malicious builds.

**🛡️ Mitigation**

* **Never trust user input** to determine which client-side resource loads.
* Use **SRI (Subresource Integrity)** when loading external JS/CSS.
* Enforce **HTTPS + CSP** to prevent MITM injection.
* Host critical scripts **locally** instead of third-party.
* Validate + sanitize filenames/paths if dynamic resource loading is required.
* Set proper cache-control headers to prevent poisoning.

✅ **Key Takeaway:**  
Client-side resource manipulation lets attackers **inject malicious code or change application behavior** by tampering with scripts, configs, or styles. Always use **SRI, HTTPS, CSP, and validation** to protect resources.

👉 Do you also want me to prepare a **quick checklist of common parameters (file=, load=, script=, etc.) attackers look for** in this type of testing? That would help you spot them fast during assessments.