Great question 👍 Let’s go deep into **Host Header Injection** in the same WSTG-style approach.

**🌐 Testing for Host Header Injection**

**🔹 1. What is Host Header Injection?**

* Web applications often rely on the **HTTP Host header** to:
  + Route requests (virtual hosting).
  + Build **absolute URLs** in responses (password reset links, email confirmations).
  + Perform access control.
* If an attacker manipulates the Host header and the application trusts it **without validation**, it can lead to:
  + **Web cache poisoning**
  + **Password reset poisoning** (reset link sent to attacker’s domain)
  + **Bypass of access controls**
  + **SSRF or internal routing attacks**

**🔹 2. Example of a Vulnerable Request**

Legitimate request:

GET / HTTP/1.1

Host: victim.com

Injected request:

GET / HTTP/1.1

Host: attacker.com

If the server uses Host directly in email templates, the victim might receive:

https://attacker.com/reset?token=abc123

**🔹 3. Attack Scenarios**

1. **Password Reset Poisoning**
   * Victim requests password reset.
   * Application uses attacker-supplied Host header → malicious link emailed.
2. **Web Cache Poisoning**
   * Cache servers (e.g., Varnish, CDN) may cache based on Host header.
   * Poisoning one request can impact many users.
3. **Access Control Bypass**
   * Some apps restrict functionality based on host (e.g., admin.victim.com).
   * Manipulated Host header may trick the app into exposing admin features.
4. **SSRF via Host Header**
   * Application might make server-to-server calls using Host header → attacker controls backend target.

**🔹 4. Manual Testing**

1. **Send modified Host headers**:
2. Host: attacker.com
3. Host: localhost
4. Host: 127.0.0.1
5. Host: internal-service
6. **Check application behavior**:
   * Does it reflect the injected host in responses?
   * Are absolute URLs in HTML/email poisoned?
   * Does it affect redirections (Location: headers)?
7. **Add duplicate Host headers** (some servers handle them differently):
8. Host: victim.com
9. Host: attacker.com

or

Host: attacker.com

X-Forwarded-Host: victim.com

1. **Inject into password reset flow**
   * Initiate password reset.
   * See if email contains attacker’s host.

**🔹 5. Testing with Burp Suite**

* **Proxy** → Intercept all requests, modify Host.
* **Repeater** → Send variations (localhost, 127.0.0.1, evil.com).
* **Intruder** → Automate payloads with common hosts.
* **Collaborator** → Detect if external requests are made to attacker-controlled hosts.

**🔹 6. Useful Payloads**

| **Purpose** | **Payload** |
| --- | --- |
| External injection | Host: attacker.com |
| Local SSRF | Host: 127.0.0.1 |
| Admin access | Host: admin.victim.com |
| Mixed headers | Host: victim.com\nX-Forwarded-Host: attacker.com |
| Injection attempt | Host: evil.com\r\nX-Forwarded-For: attacker.com |

**🔹 7. Mitigation Strategies**

✅ **Best Practices**

1. Always **validate the Host header** against a strict allowlist (e.g., victim.com, www.victim.com).
2. Use server-side configuration (ServerName, UseCanonicalName in Apache).
3. Avoid relying on the Host header for critical functionality like:
   * Password reset links.
   * Security decisions.
   * Internal routing.
4. In frameworks → use trusted base\_url settings.
5. For caching layers (CDNs, reverse proxies) → normalize Host headers and enforce strict canonicalization.

**🔹 8. Quick Detection Checklist**

* ✅ Does the app reflect the Host header anywhere (HTML, JSON, emails)?
* ✅ Can duplicate Host headers or X-Forwarded-Host override behavior?
* ✅ Does password reset or email confirmation link include attacker’s host?
* ✅ Does it poison cache or cause redirection?

👉 That’s the **full methodology for Host Header Injection**: manual testing, Burp workflow, payloads, and mitigations.

Would you like me to also create a **Burp Intruder payload list for Host Header testing** (with common hostnames + cloud/local IPs) so you can plug it directly into Burp?